3.4 does so too. The actual inductive hypothesis we need is that a tuple added to some IDB relation P in round j by Algorithm 3.3, not having been placed in that relation on any prior round, will be placed in both P and ΔP on round j by Algorithm 3.4. The basis, round 1, is immediate, since the same formulas, given by EVAL, are used by both algorithms.

For the induction, one has only to notice that if a tuple μ is added to some IDB relation P on round i, and μ was not previously in P, then there must be some rule r for predicate p (the predicate corresponding to relation P) and tuples in the relations for all the subgoals of r such that

1. The tuples for the subgoals together yield μ, and
2. At least one of these tuples, say ν, was added to its relation, say T, on round $i-1$.

By the inductive hypothesis with $j = i-1$ and observation (2) above, ν is in ΔT when we start round i of Algorithm 3.4. Therefore the term of EVAL-INCR that uses ΔT (or rather its copy into some ΔQ_j) will produce μ, since that term uses full relations for subgoals other than the one that supplies ν, and ν will be supplied by ΔT. □

3.6 NEGATIONS IN RULE BODIES

There are frequent situations where we would like to use negation of a predicate to help express a relationship by logical rules. Technically, rules with negated subgoals are not Horn clauses, but we shall see that many of the ideas developed so far apply to this broader class of rules. In general the intuitive meaning of a rule with one or more negated subgoals is that we should complement the relations for the negated subgoals, and then compute the relation of the rule exactly as we did in Algorithm 3.1.

Unfortunately, the "complement" of a relation is not a well-defined term. We have to specify the relation or domain of possible values with respect to which the complement is taken. That is why relational algebra uses a set-difference operator, but not a complementation operator. But even if we specify the universe of possible tuples with respect to which we compute the complement of a relation, we are still faced with the fact that this complement will normally be an infinite relation. We cannot, therefore, apply operations like selection or join to the complement, and we cannot perform Algorithm 3.1 on a rule with negation in a straightforward manner.

It turns out that one critical issue we face when trying to define the meaning of rules with negated subgoals is whether the variables appearing in the negated subgoals also appear in nonnegated, ordinary (non-built-in) subgoals. In the next example, we see what happens when things work right, and then we see where problems arise when variables appear only in negated subgoals. Later, we examine another problem that comes up when some subgoals are negated:

there is not necessarily a least fixed point for a logic program. Furthermore, since we have no mechanism for proving negated facts, the proof-theoretic point of view does not help us, and we are forced to select one of the minimal models as the "meaning" of the logic program.

Example 3.14: Suppose we want to define "true cousins" to be individuals who are related by the *cousin* predicate of Figure 3.1 but who are not also related by the *sibling* relationship. We might write

```
trueCousin(X,Y) :- cousin(X,Y) & ¬sibling(X,Y).
```

This rule is very much like an application of the difference operator of relational algebra, and indeed we can compute $T = C - S$, where T is the relation for *trueCousin*, and C and S are the relations for *cousin* and *sibling*, computed as in the previous section.

The formula $T = C - S$ is easily seen to give the same relation as

$$C(X,Y) \bowtie \bar{S}(X,Y)$$

where $\bar{S}$ is the "complement" of S with respect to some universe U that includes at least the tuples of C.[11] For example, we might let U be the set of individuals that appear in one or more tuples of the *parent* relation, i.e., those individuals mentioned in the genealogy. Then, $\bar{S}$ would be $U \times U - S$, and surely C is a subset of $U \times U$. □

Unfortunately, not all uses of negation are as straightforward as the one in Example 3.14. We shall investigate some progressively harder problems concerning what rules with negation mean, and then develop a set of restraints on the use of negation that allow datalog rules with this limited form of negation to be given a sensible meaning. The first problem we encounter is what happens when variables appear only in negated subgoals.

Example 3.15: Consider the following rule:

```
bachelor(X) :- male(X) & ¬married(X,Y).        (3.6)
```

Here, we suppose that *male* is an EDB relation with the obvious meaning, and *married*(X,Y) is an EDB relation with the meaning that X is the husband of Y.

One plausible interpretation of (3.6) is that X is a bachelor if he is male and there does not exist a Y such that Y is married to X. However, if we computed the relation for this rule by joining the relation *male*(X) with the "complement" of *married*, that is, with the set of (X,Y) pairs such that X is not married to Y, we would get the set of pairs (X,Y) such that X is male and Y is not married to X. If we then project this set onto X, we find that "bachelors" are

[11] Notice that the natural join is an intersection when the sets of attributes are the same, and intersection with the complement is the same as set difference.

males who are not married to absolutely everybody in the universe; that is, there exists some Y such that Y is not married to X.

To avoid this apparent divergence between what we intuitively expect a rule should mean and what answer we would get if we interpreted negation in the obvious way (complement the relation), we shall forbid the use of a variable in a negated subgoal if that variable does not also appear in another subgoal, and that subgoal is neither negated nor a built-in predicate. This restriction is not a severe one, since we can always rewrite the rule so that such variables do not appear.[12] For example, to make the attributes of the two relations involved in (3.6) be the same, we need to project out Y from *married*; that is, we rewrite the rules as:

```
husband(X) :- married(X,Y).
bachelor(X) :- male(X) & ¬husband(X).
```

These rules can then have their meaning expressed by:

$$husband(X) = \pi_X\big(married(X,Y)\big)$$
$$bachelor(X) = male(X) - husband(X)$$

or just:

$$bachelor(X) = male(X) - \pi_X\big(married(X,Y)\big)$$

□

While we shall forbid variables that appear only in negated subgoals, the condition found in Example 3.14 and in the rewritten rules of Example 3.15, which is that the set of variables in a negated subgoal exactly match the variables of a nonnegated subgoal, is not essential. The next example gives the idea of what can be done in cases when there are "too few" variables in a negated subgoal.

Example 3.16: Consider:

```
canBuy(X,Y) :- likes(X,Y) & ¬broke(X).
```

Here, *likes* and *broke* are presumed EDB relations. The intention of this rule evidently is that X can buy Y if X likes Y and X is not broke. Recall the relation for this rule is a join involving the "complement" of *broke*, which we might call *notBroke*. The above rule can then be expressed by the equivalent relational algebra equation:

$$canBuy(X,Y) = likes(X,Y) \bowtie notBroke(X) \tag{3.7}$$

The fact that *notBroke* may be infinite does not prevent us from computing

[12] Provided, of course, that we take the interpretation of $\neg q(X_1,\ldots,X_n)$ to be that used implicitly in (3.6): "there do not exist values of those variables among $X_1,\ldots,X_n$ that appear only in negated subgoals such that these values make $q(X_1,\ldots,X_n)$ true."

the right side of (3.7), because we can start with all the $likes(X,Y)$ tuples and then check that each one has an X-component that is a member of *notBroke*, or equivalently, is not a member of *broke*.

As we did in the previous two examples, we can express (3.7) as a set difference of finite relations if we "pad" the *broke* tuples with all possible objects that could be liked. But there is no way to say "all objects" in relational algebra, nor should there be, since that is an infinite set.

We have to realize that we do not need all pairs (X,Z) such that X is broke and Z is anything whatsoever, since all but a finite number of the possible Z's will not appear as a second component of a *likes* tuple, and therefore could not possibly be in the relation *canBuy* anyway. The set of possible Z's is expressed in relational algebra as $\pi_2(likes)$, or equivalently, $\pi_Y\big(likes(X,Y)\big)$. We may then express *canBuy* in relational algebra as:

$$canBuy(X,Y) = likes(X,Y) - \Big(broke(X) \times \pi_Y\big(likes(X,Y)\big)\Big)$$

Finally, we can derive from the above expression a way to express *canBuy* with rules where the only negated literal appears in a rule with a positive literal that has exactly the same set of variables, as we derived in Example 3.15. Such rules can naturally be interpreted as straightforward set differences. The general idea is to use one rule to obtain the projection onto the needed set of values, $\pi_2(likes)$ in this case, then use another rule to pad the tuples in the negated relation. The rules for the case at hand are:

```
liked(Y) :- likes(X,Y).
brokePair(X,Y) :- broke(X) & liked(Y).
canBuy(X,Y) :- likes(X,Y) & ¬brokePair(X,Y).
```

□

Nonuniqueness of Minimal Fixed Points

Adjusting the attribute sets in differences of relations is important, but it does not solve all the potential problems of negated subgoals. If $\mathcal{S}_1$ and $\mathcal{S}_2$ are two solutions to a logic program, with respect to a given set of EDB relations, we say $\mathcal{S}_1 < \mathcal{S}_2$ if $\mathcal{S}_1 \leq \mathcal{S}_2$ and $\mathcal{S}_1 \neq \mathcal{S}_2$. Recall that fixed point $\mathcal{S}_1$ is said to be minimal if there is no fixed point $\mathcal{S}$ such that $\mathcal{S} < \mathcal{S}_1$. Also, $\mathcal{S}_1$ is said to be a least fixed point if $\mathcal{S}_1 \leq \mathcal{S}$ for all fixed points $\mathcal{S}$. When rules with negation are allowed, there might not be a least fixed point, but several minimal fixed points. If there is no unique least fixed point, what does a logic program mean?

Example 3.17: Consider the rules:

```
(1) p(X) :- r(X) & ¬q(X).
(2) q(X) :- r(X) & ¬p(X).
```

Let P, Q, and R be the relations for IDB predicates p and q, and EDB predicate r, respectively. Suppose R consists of the single tuple 1; i.e., $R = \{1\}$. Let $\mathcal{S}_1$ be the solution $P = \emptyset$ and $Q = \{1\}$; let $\mathcal{S}_2$ have $P = \{1\}$ and $Q = \emptyset$. Both $\mathcal{S}_1$ and $\mathcal{S}_2$ are solutions to the equations $P = R - Q$ and $Q = R - P$.[13]

Observe that $\mathcal{S}_1 \leq \mathcal{S}_2$ is false, because of the respective values of Q, and $\mathcal{S}_2 \leq \mathcal{S}_1$ is false because of P. Moreover, there is no solution $\mathcal{S}$ such that $\mathcal{S} < \mathcal{S}_1$ or $\mathcal{S} < \mathcal{S}_2$. The reason is that such an $\mathcal{S}$ would have to assign $\emptyset$ to both P and Q. But then $P = R - Q$ would not hold.

We conclude that both $\mathcal{S}_1$ and $\mathcal{S}_2$ are fixed points, and that they are both minimal. Thus, the set of rules above has no least fixed point, because if there were a least fixed point $\mathcal{S}$, we would have $\mathcal{S} < \mathcal{S}_1$ and $\mathcal{S} < \mathcal{S}_2$. □

Stratified Negation

To help deal with the problem of many minimal fixed points, we shall permit only "stratified negation." Formally, rules are *stratified* if whenever there is a rule with head predicate p and a negated subgoal with predicate q, there is no path in the dependency graph from p to q.[14] Restriction of rules to allow only stratified negation does not guarantee a least fixed point, as the next example shows. However, it does allow a rational selection from among minimal fixed points, giving us one that has become generally accepted as "the meaning" of a logic program with stratified negation.

Example 3.18: Consider the stratified rules:[15]

```
(1) p(X) :- r(X).
(2) p(X) :- p(X).
(3) q(X) :- s(X) & ¬p(X).
```

The above set of rules is stratified, since the only occurrence of a negated subgoal, $\neg p(X)$ in rule (3), has a head predicate, q, from which there is no path to p in the dependency graph. That is, although q depends on p, p does not depend on q.

Let EDB relations r and s have corresponding relations R and S, and let IDB relations p and q have relations P and Q. Suppose $R = \{1\}$ and $S = \{1, 2\}$.

[13] Note that rules (1) and (2) are logically equivalent, but these two set-valued equations are not equivalent; certain sets P, Q, and R satisfy one but not the other. This distinction between logically equivalent forms as we convert logic into computation should be seen as a "feature, not a bug." It allows us, ultimately, to develop a sensible semantics for a large class of logical rules with negation.

[14] The construction of the dependency graph does not change when we introduce negated subgoals. If $\neg q(X_1, \ldots, X_n)$ is such a subgoal, and the rule has head predicate p, we draw an arc from q to p, just as we would if the $\neg$ were not present.

[15] If one does not like the triviality of the rule (2), one can develop a more complicated example along the lines of Example 3.9 (paths in a graph) that exhibits the same problem as is illustrated here.

Then one solution is $\mathcal{S}_1$ given by $P = \{1\}$ and $Q = \{2\}$, while another is $\mathcal{S}_2$ given by $P = \{1, 2\}$ and $Q = \emptyset$. That is, both $\mathcal{S}_1$ and $\mathcal{S}_2$ are solutions to the equations $P = P \cup R$ and $Q = S - P$.[16]

One can check that both $\mathcal{S}_1$ and $\mathcal{S}_2$ are minimal. Thus, there can be no least fixed point for the rules of this example, by the same reasoning we used to conclude there is none for the rules of Example 3.17. On the other hand, $\mathcal{S}_1$ is more "natural," since its tuples each can be obtained by making substitutions of known facts in the bodies of rules and deducing the fact that appears at the head. We shall see later how the proper attribution of "meaning" to stratified rules produces $\mathcal{S}_1$ rather than $\mathcal{S}_2$. □

Finding Stratifications

Since not every logic program with negations is stratified, it is useful to have an algorithm to test for stratification. While this test is quite easy, we explain it in detail because it also gives us the *stratification* of the rules; that is, it groups the predicates into *strata*, which are the largest sets of predicates such that

1. If a predicate p is the head of a rule with a subgoal that is a negated q, then q is in a lower stratum than p.
2. If predicate p is the head of a rule with a subgoal that is a nonnegated q, then the stratum of p is at least as high as the stratum of q.

The strata give us an order in which the relations for the IDB predicates may be computed. The useful property of this order is that following it, we may treat any negated subgoals as if they were EDB relations.

Algorithm 3.5: Testing For and Finding a Stratification.

INPUT: A set of datalog rules, possibly with some negated subgoals.

OUTPUT: A decision whether the rules are stratified. If so, we also produce a stratification.

METHOD: Start with every predicate assigned to stratum 1. Repeatedly examine the rules. If a rule with head predicate p has a negated subgoal with predicate q, let p and q currently be assigned to strata i and j respectively. If $i \leq j$, reassign p to stratum $j+1$. Furthermore, if a rule with head p has a nonnegated subgoal with predicate q of stratum j, and $i < j$, reassign p to stratum j. These laws are formalized in Figure 3.7.

If we reach a condition where no strata can be changed by the algorithm of Figure 3.7, then the rules are stratified, and the current strata form the output of the algorithm. If we ever reach a condition where some predicate is assigned a stratum that is larger than the total number of predicates, then the rules are not stratified, so we halt and return "no." □

[16] If we did not have rule (2), then the first equation would be $P = R$, and there would be a unique solution to the equations.