CHAPTER 12

More About Logic

We shall now resume the study of logical rules as a way to express knowledge, which we began in Chapter 3 (Volume I). We begin by examining Horn-clause rules with function symbols in Section 12.1, and in Section 12.2 we discuss the natural least-fixed-point semantics for such rules. There, we extend the "bottom-up" approaches of Chapter 3, called "naive" and "semi-naive" evaluation, to allow function symbols in rules. In Section 12.3 we consider "top-down" evaluation of logical rules, with and without function symbols, and the advantages and disadvantages of this approach, compared with "bottom-up," are explored. The technique described here for top-down processing is the method used by Prolog, and it is sometimes referred to as "SLD-resolution" or "LUSH-resolution." Section 12.4 gives an algorithm for unification, an essential step in top-down processing of logic.

Subsequent sections lay the groundwork for Chapters 13 to 15, which cover optimization of queries that are expressed as logical rules. Sections 12.5 to 12.7 are a discussion of what is, in a sense, a "straw-man" algorithm for evaluating logical queries in a top-down fashion; it may be regarded as a relation-oriented version of SLD-resolution. We introduce some important concepts for further study of efficient algorithms: side-ways information passing and top-down passing of binding relations. However, the primary reason for studying this algorithm is so we can understand what the more efficient algorithms of Chapter 13 are doing, and see why they are no less efficient and no less general than the simple top-down, relation-based algorithm.

In Section 12.8 we introduce a useful tool for describing the interactions among rules, the "rule/goal graph." Section 12.9 shows how to use the rule/goal graph to determine the binding pattern (set of bound arguments) of each predicate as we process the rules top-down. We also introduce in Section 12.9 two constructions that are very important for the "magic sets" algorithms of Chapter 13: making binding patterns unique and "rectifying subgoals." The last section explains how to use the rule/goal graph to select a good order for the subgoals of the various rules.

Notation for Logic

The following conventions were followed in Volume I, when we discussed logic, and we adopt them here.

1. A logical rule is written

   \[ H \rightarrow G_1 \land \cdots \land G_k \]

   \( H \) is called the head, and \( G_1, \ldots, G_k \) are the subgoals. The subgoals together are called the body. The head and each of the subgoals are atomic \textit{formulas}, whose form is \( p(t_1, \ldots, t_n) \), where \( p \) is a predicate symbol, and the \( t_i \)'s are arguments. In Volume I, arguments could be only variables or constants, while here we shall allow more general expressions, to be defined in Section 12.1. The meaning of the rule is "if \( G_1, \ldots, G_k \) are true, then \( H \) is true."

2. Certain subgoals have predicates that are given a conventional interpretation, that of an arithmetic comparison operator, for example, \( X < Y \). These subgoals and their predicates are called \textit{built-in}. Other subgoals have uninterpreted symbols as predicates. These predicates represent finite relations; we call them ordinary subgoals and predicates.

3. Predicate symbols are denoted by character strings beginning with a lowercase letter, for example, \( p \) or \( foo \). Variables are denoted by character strings beginning with a capital letter.

4. Certain predicates are defined by the rules: that is, they appear as the head of one or more rules. These are called IDB (\textit{intensional database}) predicates. Other predicates are not defined by rules, but by a stored relation, and these are called EDB (\textit{extensional database}) predicates. An EDB predicate can only appear in subgoals; an IDB predicate can appear in both heads and subgoals.

5. Corresponding to each ordinary predicate is a relation—an "EDB relation" for an EDB predicate and an "IDB relation" for an IDB predicate. Whether \( p \) is an EDB or an IDB predicate, the value of the corresponding relation \( P \) is a collection of tuples. However, if \( p \) is an IDB predicate, then \( P \) is not stored, but is constructed as needed from the rules. A tuple of relation \( P \) can be denoted by any of the notations used for tuples in general, for example, \( p, abc \), or \( (a, b, c) \). We also use the notation \( p(a, b, c) \), that is, the predicate name with arguments corresponding to the components of the tuple. We may say "\( p(a, b, c) \) is true" to mean that \( abc \) is in the relation \( P \). Finally, when there is no confusion, we may refer to the relation for \( p \)
as $p$ itself.

6. A rule is safe if every variable that appears in the head also appears in the body, in a nonnegated, ordinary subgoal. Safety was introduced in Chapter 3 as a way to guarantee that finite EDB relations implied that the IDB relations were also finite. As we shall see, safety is not sufficient to guarantee finite IDB relations when there are function symbols in the logic. We shall continue to assume that rules are safe unless stated otherwise, however.

12.1 LOGIC WITH FUNCTION SYMBOLS

In Volume I we restricted ourselves to logic in the datalog style, where the arguments of predicates are only variables or constants. That is perfectly reasonable when we model relational query languages, which can be viewed as notations for nonrecursive datalog. There are, however, some situations described more easily by the use of arguments that are terms built from variables, constants, and function symbols. Terms may be defined recursively by

1. A variable is a term.
2. A constant is a term.
3. If $f$ is a function symbol, and $T_1, \ldots, T_k$ are terms, then $f(T_1, \ldots, T_k)$ is a term.

In full first-order logic, the arguments of predicates may be arbitrary terms, rather than constants or variables only. We shall continue to use the Prolog style of rules, that is, Horn clauses, with possibly negated subgoals in the body. Tuples of EDB relations will be permitted to have components that are arbitrary terms, rather than being restricted to be simple constants, as heretofore. Unless we state otherwise, we assume that terms in EDB relations do not have variables, but there are some interesting applications where these terms do have variables. Terms without variables are called ground terms. A predicate whose arguments are ground terms is a ground atom. We may view a ground atom as a tuple of a relation, but the components of these tuples are not necessarily elementarily values (for example, integers or character strings); they can be terms with nontrivial structure.

**Example 12.1:** One simple use of function symbols is to incorporate into relations some of the structure that is possible in the hierarchical model. Thus, we might store an EDB relation `emps` with the scheme

$$\text{emps} \{ \text{name}(F, L), \text{addr}(\text{street}(N, S), \text{city}(C, T, Z)) \}$$

(12.1)

Here, `emps` is a predicate symbol with a binary relation. The first argument is a term consisting of the binary function symbol `name` applied to arguments $F$ and $L$, representing the first and last name of the employee. Note that the variables $F$ and $L$ will, in each tuple of the relation, be replaced by two appropriate constants.

The second component is the binary function symbol `addr` applied to two arguments that are themselves nontrivial terms. The first is function symbol `street` applied to variables $N$ and $S$, standing for number and street name, respectively, while the second is function symbol `city` applied to variables $C$, $T$, and $Z$, standing for city name, state name, and zip code. A possible tuple of the `emps` relation is

$$\text{emps} \{ \text{name}(\text{arnold, avarice}), \text{addr}(\text{street}(23, \text{lois, jane}), \text{city}(\text{yuppie, valley, calf, 94720})) \}$$

While we suggested that (12.1) was the "scheme" for the relation `emps`, in fact, we gain power from the use of function symbols if we permit many different "schemes" and apply appropriate function symbols to help interpret the data. For example, if an employee lives in an apartment, then instead of a term `street(N, S)` in the first component of `addr`, we wish to have a term `apt(A, street(X, S))`, where $A$ is the apartment number. Thus, if Esther Eggplant lives in an apartment building next door to Arnold Avarice, we could have the `emps` "tuple"

$$\text{emps} \{ \text{name}(\text{esther, eggplant}), \text{addr}(\text{apt}(3d, street(21, \text{lois, jane})), \text{city}(\text{yuppie, valley, calf, 94720})) \}$$

Suppose we wish to print addresses in the customary format: the employee's name on line 1, the apartment number if there is one on the next line, the street address on the next line, and the city, state, and zip on the last line. Thus, the third line will contain the street if the first component of the address has an apartment number, that is, if the function symbol is `apt`. If the function symbol is `street`, then the third line contains the information in the second component of `addr`. We can express these observations as rules with function symbols. The following rules define a predicate `print3`, which we might think of as the set of third lines of the addresses of all the employees.

$$\text{print3}(X, S) :- \text{emps}(X, \text{addr}(\text{apt}(Y, \text{street}(N, S)), Z)).$$
$$\text{print3}(C, T, Z) :- \text{emps}(U, \text{addr}(\text{street}(V, W), \text{city}(C, T, Z))).$$

Notice how the structure provided by the function symbols not only allows us to deduce the meaning of different fields in a tuple, but also it allows us
to ignore the structure of parts of tuples in which we are not interested. For example, Z in the first rule stands for whatever is in the city field of the address portion of the tuple.

The reader should not be overly concerned by the fact that the two rules for \textit{print3} use different numbers of arguments. Formally, the relation for \textit{print3} is assumed to have one particular arity, and it would not make sense to place in that relation tuples of different arities. However, what is intended in this example is that all the components of a \textit{print3} tuple are printed in the order of appearance (and actually, we should have added necessary spaces and a comma after the city); there is no IDB relation for \textit{print3} constructed. To be formally correct, we could wrap the arguments of \textit{print3} in a function symbol, since function symbols can be used with different numbers of arguments. That is, we could write the heads of the rules for \textit{print3} as \textit{print3} with \textit{print3} with \textit{print3}(\textit{line}(N),S)) and \textit{print3}(\textit{line}(C,T,Z)).)

In Example 12.1, function symbols were, in effect, used to create named record structures, and to define alternative structures for the fields of records. That capability is often a convenience, although in principle it can be dispensed with. For example, we could have defined an ordinary ("flat") relation with format $\textit{empa}(P,I,A,N,S,C,T,Z)$ where the $A$ (apartment) component is null for those who do not dwell in an apartment.

When rules are recursive, however, the use of function symbols allows us to create an infinite set of potential values that may appear in relations, as the next example shows. Notice that datalog rules are not powerful enough to create values that did not already exist in the EDB relations or in the rules themselves.

Example 12.2: Addition can be defined recursively as follows. Suppose we have a predicate symbol \textit{int}(X) that is intended to be true if \(X\) is a nonnegative integer and a predicate \textit{sum}(X,Y,Z) that is to be true if and only if the sum of \(X\) and \(Y\) is \(Z\). Suppose also that we have the function symbol \textit{succ} with the intention that \textit{succ}(X) is the integer \(X + 1\). Finally, we have the constant 0, with its obvious meaning. Then the term \textit{succ}(0) is our way of saying "1," \textit{succ}(\textit{succ}(0)) is "2," and so on; the integers other than 0 do not exist in any other form. Then the rules we desire are shown in Figure 12.1.

That is, the first two rules in Figure 12.1 define 0, \textit{succ}(0), \textit{succ}(\textit{succ}(0)), and so on, to be integers. The third rule says that the sum of any integer and 0 is that integer, and the last says that if \(X + \textit{Y} + 1 = Z + 1\), if \(X + Y = Z\) put another way, \(X + (Y + 1) = (X + Y) + 1\). These rules are sufficient to generate all and only the true addition facts.

For example, \textit{int}(0) follows from rule (1). Then, from rule (3), \textit{sum}(0,0,0)

(1) \textit{int}(0).
(2) \textit{int}(\textit{succ}(X)) :\!-\! \textit{int}(X).
(3) \textit{sum}(X,0,X) :\!-\! \textit{int}(X).
(4) \textit{sum}(X,\textit{succ}(Y),\textit{succ}(Z)) :\!-\! \textit{sum}(X,Y,Z).

Figure 12.1 Rules for defining addition.

is true. Applying rule (4) to this fact we have \textit{sum}(0,\textit{succ}(0),\textit{succ}(0)), that is, \textit{sum}(0,1,1). From rule (2), we infer \textit{int}(\textit{succ}(0)), that is, \textit{int}(1). Then rule (3) gives us \textit{sum}(1,0,1) and (4) gives us \textit{sum}(1,1,2). In these facts we have used 1 to stand for \textit{succ}(0) and 2 to stand for \textit{succ}(1) = \textit{succ}(\textit{succ}(0)). Note that both \textit{int} and \textit{sum} have infinite relations, even though the rules of this example are safe.

Let us observe that although the syntax for function symbols and predicate symbols is the same (strings beginning with a lowercase letter), there is no confusion between these two types of symbols. Function symbols take domain values as arguments and return domain values. Predicate symbols take domain values as arguments but return truth values (true or false). Thus, in Example 12.2, \textit{succ} takes one integer as an argument and returns an integer, while \textit{sum} takes three integer arguments; it returns true if the third is the sum of the first and second, and it returns false otherwise. In Example 12.1 we can see a function symbol like \textit{name} as taking two domain elements, character strings in this case, and returning an "object" of type "record of two character strings."

In our next example, we see how function symbols can be used both to create an infinity of potential new objects and to give clues that let us decipher the structure of the object. It is a very trivial example of a reasoning system about a world with five states. It allows us to describe the true facts about transitions from state to state, thus letting us to plan how to get from one state to another. In a realistic situation, there might be millions of states, and the power of a knowledge-base system would be needed to deduce the necessary facts efficiently.

Example 12.3: Let us consider a simple "blocks world" with three blocks, \textit{A}, \textit{B}, and \textit{C}, which may be placed in piles only if the piles are in decreasing size order. Block \textit{A} is larger than \textit{B}, which is larger than \textit{C}. The five possible states of the system, \(a\) through \(e\), are shown in Figure 12.2.

The legal moves of the system are expressed by an EDB relation \textit{GO}(\textit{S},\textit{T}) that is intended to be true if and only if state \(S\) can become state \(T\) by one "move." Here, a move shifts the position of one block, removing it from the block or the floor on which it rests and placing it either on the floor or on a larger block that is itself uncovered. For the states of Figure 12.2, the relation
Figure 12.2 Blocks-world states.

GO consists of the ten tuples that, written as ground atoms, are

\[ \begin{align*}
  &go(a, e) \quad go(b, c) \quad go(b, d) \quad go(c, b) \quad go(c, d) \\
  &go(d, b) \quad go(d, c) \quad go(d, e) \quad go(e, a) \quad go(e, d)
\end{align*} \]

Let us use a binary function symbol \( f \), and let \( g(S, T) \) be a term representing the possibility that state \( S \) can be transformed into state \( T \) by one or more moves. We define the unary predicate \( true \) with the intention that \( true(X) \) holds exactly when term \( X \) represents a statement about state transitions that is true. For example, we would naturally have the rule

\[ true(g(S, T)) := go(S, T). \]

Here, go is presumed to be the EDB predicate that corresponds to the EDB relation GO.

We also need a rule that defines what “goes to by one or more moves” means, that is,

\[ true(g(S, T)) := true(g(S, U)) \land true(g(U, T)). \]

Next, we may wish to talk about “accessible” states. Suppose we are given a unary EDB relation ACC\((S)\) of states \( S \) that are initially accessible. For example, ACC may contain only the one state in which we initially find the blocks world. We may then use unary function symbol \( o(S) \) to represent the possibility that state \( S \) is accessible after some number of moves. The rules for when an accessibility condition is true are

\[ \begin{align*}
  &true(o(S)) := acc(S). \\
  &true(o(S)) := true(o(T)) \land true(g(T, S)).
\end{align*} \]

In rules (1) through (4), the use of function symbols is not essential; we could remove the “true” everywhere it appears, treating \( a \) and \( b \) as if they were predicates rather than function symbols, and the rules would say essentially the same thing. However, there are other things we can say with the approach taken here that are considerably more natural when we use function symbols. For example, we may wish to use terms that express conditions more complicated than “goes” or “accessible.” We could, for example, use binary function symbols \( or \) and \( and \) to represent the logical OR and AND of two conditions. We could then include in our “knowledge base” rules defining AND and OR

\[ \begin{align*}
  &true(or(X, Y)) := true(X). \\
  &true(or(X, Y)) := true(Y). \\
  &true(and(X, Y)) := true(X) \land true(Y).
\end{align*} \]

Notice that these rules do not depend on the structure of the terms \( X \) and \( Y \); they could be “goes” facts, “accessible” facts, or the conjunction or disjunction of several facts. Also notice that rules (5) and (6) are not safe, since they each have a variable that appears in the head but not the body. There are two approaches to fixing this problem.

1. Write another predicate possible\((X)\) that says \( X \) is a term that might be true, i.e., all its constants come from the EDB relations GO and ACC. We can use possible to restrict the range of the variable that otherwise would not appear in the body. For example, (5) could be rewritten

\[ true(or(X, Y)) := true(X) \land possible(Y). \]

2. We could accept the unsafe rules. That prevents us from applying a rule to infer all of the true facts, since from a fact like \( true(g(a, c)) \) follow an infinity of facts with different values of \( Y \) in rule (5). However, it would not prevent us from answering questions about particular facts, like “is \( or(g(a, b), g(c, d)) \) true?”

The exercises examine both of these possibilities. □

12.2 EVALUATING LOGIC WITH FUNCTION SYMBOLS

In this section, we shall generalize the techniques for evaluating rules without function symbols that were introduced in Sections 3.3 and 3.4 of Volume I. The reader who is not familiar with these methods and concepts is advised to reread those sections before proceeding.

In Chapter 3 we offered Algorithm 3.3, called “naive evaluation,” for com-
puting the least fixed point of a collection of rules with no negated subgoals. The idea is to start by assuming the IDB relations are all empty, then repeatedly to compute new values for the IDB predicates, by applying the rules to the EDB relations and the current values of the IDB relations. When the rules are datalog, this process eventually converges to a finite relation, which is the “meaning” of the rules, as we saw in Section 3.3. When there are function symbols, we can show that a least fixed point exists, but it may be infinite, in which case no finite number of repetitions of the loop of naive evaluation suffices to reach the least fixed point.

We also gave, in Chapter 3, a more efficient algorithm, called “semi-naive evaluation,” Algorithm 3.4. Naive evaluation, once it discovers a fact, redisCOVERS IT ON SUBSEQUENT PASSES. Semi-naive evaluation avoids discovering the same fact in the same way twice, by insisting that each time we apply a rule, at least one of the subgoals uses a fact that was just discovered on the previous round. We shall describe these algorithms in more detail later in the section.

To begin, we shall offer some informal examples of the operation that we called EVAL($p, R_1, \ldots, R_n$) in Chapter 3. This operation takes relations $R_1, \ldots, R_n$ for the subgoals of all the rules for predicate $p$ and produces the relation for $p$ that we get by “applying” the rules to the relations. That is, we look for substitutions for the rules’ variables that make each subgoal a tuple of the corresponding relation. The set of successful substitutions for the variables is called the “relation for the rule”; its attributes are the variables appearing in the rule. We then take each tuple in the relation for the rule, evaluate the head with that substitution, and take the resulting ground atom to be a tuple of the relation for $p$.

Example 12.4: Let us return to Example 12.2, where we discussed a logical definition of addition facts. This example is particularly simple, since there are no function symbols in the subgoals. Thus, EVAL operates almost as in Chapter 3, but when we find a successful substitution for the variables, and we make the substitution into the head, we get nontrivial terms as arguments.

Let us first apply naive evaluation to only rules (1) and (2) of Example 12.2, defining a unary relation int. Note there is no EDB for the rules of this example.

We initialize the IDB relation for int to Ø. Then, on pass 1 of Algorithm 3.3, rule (2) yields nothing, because int($X$) has no tuples. However, rule (1) produces the tuple 0 for relation int.

On pass 2, the first rule produces only 0 again, but the second rule now has a value, 0, for $X$, and so produces the tuple suc(0) for int. Note that this tuple has one component, and that component is a term with function symbol suc.

On the third pass, $X$ can also take on the value suc(0); technically, the relation for rule (2) is a relation over attribute $X$, namely $\{0, \text{suc}(0)\}$. This new value for $X$ yields the new tuple suc(0), suc(0) for int. In general, the $i$th pass produces one new tuple, suc($i-1$), for int. As a result, the value of the relation int in the least fixed point is the set of nonnegative integers $\{0, 1, 2, \ldots\}$, with integer $i$ represented by term suc($i$). Thus, the rules of Example 12.2 define int to be an infinite relation, and an infinite number of iterations of the loop of naive or semi-naive evaluation is necessary to “attain” this solution.

A similar process occurs when we follow rules (3) and (4) to compute the relation sum. On the first pass, the subgoals in the bodies of both rules have empty relations, so no sum tuples are produced. On pass 2, int contains 0, so rule (3) produces tuple (0, 0, 0) for sum.

On pass three, int also contains suc(0), so rule (3) produces

$$\{(0, 0, 0), (0, \text{suc}(0))\}$$

Also, sum contains (0, 0, 0), so rule (4) yields (0, suc(0), suc(0)). In general, the reader can check that the addition fact

$$\text{sum}(\text{suc}(0), \text{suc}(0), \text{suc}^2(0))$$

is added on pass $i + j = 2$, using rule (3) if $j = 0$ and rule (4) if $j > 0$. No other tuples are added to sum. Thus, sum also is an infinite relation in the least fixed point, requiring an infinite number of passes to attain.

In a sense, infinite relations like int and sum of Example 12.4 are not computable at all, since we never reach the least fixed point of their defining rules, no matter how long we run naive evaluation. On the other hand, the noncomputability of the relations does not contradict the fact that they are well defined by the process of taking the least fixed point, and in the next section we shall see how we can use rules defining infinite relations to get some answers in finite time. In particular, we can often answer the question whether a particular tuple is in such a relation, even if we cannot construct the complete relation.

Example 12.5: Now, let us reconsider Example 12.3. To begin, suppose only rules (1) and (2) are available. Recall these rules are

1. \(\text{true}(g(S,T)) \iff \text{go}(S,T)\).
2. \(\text{true}(g(S,T)) \iff \text{true}(g(S,0)) \& \text{true}(g(0,T))\).

On the first pass, relation true is empty, so rule (2) yields nothing. EDB relation GO consists of the ten tuples given in Example 12.3. Thus, on pass 1 the relation for rule (1) is GO($S, T$). From each of the ten tuples in this relation we construct a term g($S, T$), and this term becomes the lone component in a
one-component tuple of the unary relation for true. Thus, after pass 1, the relation for true contains \( g(a, e) \), \( g(e, d) \), and eight other tuples, each with a single component.\(^3\)

Subsequent passes yield no new tuples by rule (1), but now rule (2) begins to yield new tuples for true. To see how, let us first evaluate the relation for true. As shown in Figure 12.3, we recall that the relation for true is the ten tuples of one component each, shown in Figure 12.3. Recall that the relation for this rule will have attributes \( S, T \), and \( U \), corresponding to the three variables appearing in the body of the rule. Each tuple must have values \( s, t, \) and \( u \) for these variables, such that both of the subgoals true \((g(S, U))\) and true \((g(U, T))\) become tuples of the relation for true when we substitute \( s \) for \( S \), \( t \) for \( T \), and \( u \) for \( U \).

\[
\begin{align*}
g(a, e) & \quad g(b, c) & \quad g(b, d) & \quad g(c, b) & \quad g(c, d) \\
g(d, b) & \quad g(d, c) & \quad g(d, e) & \quad g(e, a) & \quad g(e, d)
\end{align*}
\]

**Figure 12.3** Initial relation for predicate true.

To see what values \( s, t, \) and \( u \) might be, we match each subgoal with each tuple of the relation true, which gives us values for those variables appearing in that subgoal, \( S \) and \( U \) for the first subgoal and \( U \) and \( T \) for the second. Then, we take the natural join of the sets of tuples for each subgoal.\(^4\) In our particular example, we must match the subgoal true \((g(S, U))\) against each of the ten tuples in the relation true of Figure 12.3. For example, comparing the first tuple, \( g(a, e) \), we see that a match is possible, and the value of \( S \) becomes \( a \), while \( U \) becomes \( e \). In fact, for each of the ten tuples in true, the match succeeds, and we get the relation 

\[
\{aa, bc, bd, cb, cd, db, dc, de, ca, ed\}
\]

which we shall call \( Q_1(S, U) \), for the first subgoal of rule (2). For the second subgoal, true \((g(U, T))\), we also construct a relation, which we may call \( Q_2(U, T) \). The value of \( Q_2 \) is easily seen to be the same as \( Q_1 \), that is (12.2). However, while \( Q_1 \) is a relation over attributes \( S \) and \( U \), in that order, \( Q_2 \) is over \( U \) and \( T \), in that order.

\[
^3\text{Two not confused between function symbols and predicate symbols. In this context,}
g \text{ is a function symbol, while true is the predicate symbol; } g(a, e) \text{ is a single}
\text{component of a tuple for true, while } a \text{ and } e \text{ are not components of a tuple, but rather }
\text{arguments of the function symbol } g.\]

\[
^4\text{The join corresponds to Step (3) of Algorithm 3.1, where we joined relations called}
\text{the } Q_i's \text{ constructed by relational algebra from each of the subgoals. The difference}
\text{is algebra from the relation for the predicate of the subgoal, so the } Q_i's \text{ were constructed}
\text{called "term matching," to be discussed shortly, on the tuples.}
\]

We now take the join \( R = Q_1 \times Q_2 \) to get the relation for rule (2). \( R \) contains 22 tuples, each with components for attributes \( S, U, \) and \( T \), respectively. Some of these tuples are \( aa \) from \( aa \) and \( ca \), \( ba \) from \( ba \) and \( ca \), \( bd \), and \( edc \). For each tuple, we construct the one-component tuple \( g(S, T) \) that the head of rule (2) says belongs in the relation for true. For example, the four tuples of \( R \) mentioned above yield the following tuples for true: \( g(a, d), g(b, d), g(b, e), \) and \( g(e, d) \). Of these, only the second was already known to be in true.

The iterative computation of true using only rules (1) and (2) will eventually converge, because unlike Example 12.4, we do not build up progressively larger terms; here all terms for the one component of relation true are of the form \( g(x, y) \), where \( x \) and \( y \) are values appearing in the EDB relation \( GO \) (these values are obtained in our example data). If \( GO \) is finite, as it must be, then true cannot be given an infinite number of different tuples.

On the other hand, if we used rules (5) through (7) of Example 12.3, then we could build up an infinity of different tuples in true. For example, suppose we applied rule (7):

\[
\text{true}(\text{and}(X, Y)) \iff \text{true}(X) \land \text{true}(Y).
\]

On pass 2, when true has the ten tuples of Figure 12.3, any one of them could match \( X \) and any could match \( Y \). Thus, the relation for the first subgoal, \( true(X) \), has the ten values for \( X \): \( g(a, e), g(b, c), \ldots \) and the relation for the second subgoal has the same relation, but the attribute is \( Y \). That is, \( X \), \( Y \), and \( Z \), are variables, can each match any term.

The relation for rule (7) thus consists of the 100 tuples over attributes \( X \) and \( Y \), the Cartesian product of the relation of Figure 12.3 with itself. When we substitute for \( X \) and \( Y \) in the head of rule (7), we get 100 tuples for true, of which \( \text{and}(g(a, e), g(b, c)) \) is an example. These tuples are available at the next pass to match \( X \) and \( Y \) in rule (7) to build still bigger terms, like

\[
\text{and}\{\text{true}(X), \text{true}(Y)\}
\]

and so on.

The rules (5) and (6) present a bigger problem since they are not safe rules. We need to solve the problem by one of the ways suggested in Example 12.3: rewriting the rules to be safe, or not trying to use them in a naive-evaluation algorithm, where we compute all the head values that come from substitutions of ground terms for the variables of the body. \( \square \)

**Term Matching**

As we saw from the above examples, the operation we called "term matching" plays a fundamental role in the generalization of naive or semi-naive evaluation. That is, we are given a subgoal, which is an atomic formula normally containing
variables, and we try to match it with a ground atom, that is, with a tuple of the relation for the predicate that appears in the subgoal. When we match an arbitrary atomic formula with a ground atom, we substitute for each of the variables of the first formula to make it identical to the second. The following algorithm tests whether such a substitution exists, and finds it if so.

Algorithm 12.1: Term Matching.

INPUT: An atomic formula $F$ with variables $X_1, \ldots, X_n \ (n \geq 0)$ and a ground atom $G$.

OUTPUT: If it exists, the unique substitution $\tau$ on the variables $X_1, \ldots, X_n$ such that when, for all $i$, we substitute $\tau(X_i)$ for each occurrence of $X_i$ in $F$, the result is $G$. If there is no such $\tau$, we indicate failure.

METHOD: Initially, $\tau(X_i)$ is undefined for all $i$. We then apply the recursive procedure match shown in Figure 12.4 to $F$ and $G$. If the call match($F$,$G$) succeeds (returns true), then $\tau(X_i)$ will be defined for all $i$, and this substitution is the term matching that is produced as output. If the call fails (returns false), then no term matching exists.

The idea behind the procedure match($A$, $B$) is that if $A$ is a single variable, we must have $\tau(A) = B$, no matter what term $B$ is. If $\tau(A)$ has not yet been defined, we define it to be $B$, and should we later try to define $\tau(A)$ to be something else, then we fail. If $A$ is a term or the entire atomic formula, then we consider each of the arguments of $A$, and compare them with the corresponding arguments of $B$, by a recursive call to match. If the numbers of arguments in $A$ and $B$ are different, or if their outermost operators (often called the principal function) of $A$ and $B$ are different, then we fail. Otherwise, we succeed only if all the recursive calls succeed, and we fail if one or more fail.

Example 12.6: Let us match atomic formula $p(f(h(a), b), g(h(a)))$ against ground atom $p(f(h(a), b), g(h(a)))$.

The principal functions of both are $p$, and they each have two arguments, so we must match argument $f(X, Y)$ against $f(h(a), b)$ and $g(X)$ against $g(h(a))$.

For the first pair, we again find that the principal functions, $f$, agree, and that the number of arguments is two in each case. Thus, we recursively match $X$ against $h(a)$ and $Y$ against $b$. These matches succeed, giving us $\tau(X) = h(a)$ and $\tau(Y) = b$. Then we consider the second pair, where the principal functions are both $g$, and each has one argument, $X$ and $h(a)$, respectively. Now, since $\tau(X)$ is defined, we must verify that $\tau(X)$ is identical to the expression against which $X$ is being matched. It is, in this case, so we have checked all subexpressions, and the match succeeds.

A simple example where the match fails is $p(X, X)$ against $p(a, b)$. The first call to match spawns calls match($X, a$) and match($X, b$). The first succeeds, but sets $\tau(X) = a$. Then, the second call checks that $\tau(X) = b$, finds it does not, and fails, causing the entire algorithm to fail.

Term matching is the heart of two operations needed in the generalized naive and semi-naive evaluation algorithms that work for rules with function symbols. The operations, called ATOV (arguments-to-variables) and VTOA (variables-to-arguments), can be thought of as generalized relational algebra operations. We describe these operations, and then give the generalized evaluation algorithms.

Converting from Arguments to Variables

The relation for a rule body is constructed from relations for the subgoals of the body. Each relation for a subgoal has attributes corresponding to the arguments of the predicate for that subgoal. Our first task is to convert the relations for the subgoals to relations over the variables mentioned in that subgoal, using the operation ATOV. The relations for the subgoals, converted to the viewpoint of variables, can then be joined to find the relation for the body of the rule. This relation, in turn, is converted to a relation for the head, by translating from
the viewpoint of variables to the viewpoint of arguments; the VTOA operation, to be described shortly, is used.

The argument-to-variable conversion algorithm for a single subgoal is outlined in Figure 12.5. It converts a relation \( P \), whose attributes correspond to the arguments of a predicate \( p \), into a relation \( Q = \text{ATOV}(G, P) \), whose attributes correspond to the variables appearing in the subgoal \( G \); \( G \) must be a subgoal with predicate \( p \). To do so, we term-match each tuple of \( P \) with the atomic formula \( G \), and each time we match successfully, we create a tuple of \( Q \) from the term matching \( \tau \). Notice how \( \text{ATOV} \) can be thought of as a generalized selection or projection, since, like these operations of relational algebra, they process tuples one at a time, to convert one relation to another.

Figure 12.5 Converting from arguments to variables.

(a) Relation for \( Q \).

<table>
<thead>
<tr>
<th>( Q_1 )</th>
<th>( Q_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h(c) )</td>
<td>( g(f(a), a) )</td>
</tr>
<tr>
<td>( h(h(d)) )</td>
<td>( g(f(a)), f(a) )</td>
</tr>
</tbody>
</table>

(b) Relation for \( R \).

<table>
<thead>
<tr>
<th>( R_1 )</th>
<th>( R_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c )</td>
<td>( b )</td>
</tr>
<tr>
<td>( h(d) )</td>
<td>( g(b, c) )</td>
</tr>
</tbody>
</table>

Example 12.7: Consider the following rule:

\[
p(f(X), g(X, Y)) \vdash q(h(Z), g(f(X), X)) \land r(Z, Y).
\]  

Let the relation for the first subgoal consist of the two tuples shown in Figure 12.6(a), and let the relation for the second subgoal, \( r(Z, Y) \), be that of Figure 12.6(b). If we term-match the first subgoal of (12.3) with the first tuple of Figure 12.6(a), we get the term matching \( \tau_1(X) = a \) and \( \tau_1(Z) = c \). When we do the same for the second tuple of Figure 12.6(a), we get the matching \( \tau_2(X) = f(a) \) and \( \tau_2(Z) = h(d) \). Thus, the output of the algorithm of Figure 12.5, when applied to Figure 12.6(a) and the first subgoal of (12.3), is

\[
\begin{array}{ccc}
X & Z \\
a & c \\
f(a) & h(d) \\
\end{array}
\]

We now term-match the second subgoal of (12.3), \( r(Z, Y) \), with the two tuples of Figure 12.6(b). The result of the algorithm of Figure 12.5 on these two tuples is the relation

\[
\begin{array}{ccc}
Y & Z \\
b & c \\
g(b, c) & h(d) \\
\end{array}
\]

We can take the natural join of the two relations above, to get the relation for the body of (12.3), which is given in Figure 12.7. Notice that in this simple example of a join, each tuple from one relation joins with exactly one tuple of the other relation. Also, each tuple of the relations for the subgoals matches the subgoals, and thus yield tuples over the variables of those subgoals. In typical cases, tuples in the relations for the subgoals may not term-match with the subgoals themselves; it is also typical that the join has dangling tuples and has tuples that join with several tuples of another relation.

Figure 12.7 Example relation for the body of (12.3).

---

\( ^a \) We shall adopt the convention of using \( P_i \) for the attribute name of the \( i \)th argument of predicate \( p_i \), and similarly for other predicate names.
Converting from Variables to Arguments

Given a relation $R(X_1, \ldots, X_n)$ for the body of a safe rule, we can construct the relation $S$ for the head by the algorithm of Figure 12.8. In brief, we look at each tuple $\mu$ of the relation $R$ for the body, and substitute for each variable $X$ that appears in the head the value of the component of $\mu$ that corresponds to attribute $X$. The tuple that results when we substitute for all variables of the head $H$, according to the tuple $\mu$, is a tuple of the relation $S = \text{VTOA}(H, R)$.

for each tuple $\mu$ of relation $R$ do begin
for each variable $X$ appearing in head $H$ do
replace all occurrences of variable $X$ in $H$ by $\mu[X]$;
let the resulting head predicate be $p(t_1, \ldots, t_m)$;
add the tuple $(t_1, \ldots, t_m)$ to $S$
end

Figure 12.8 Constructing a relation for the head from that of the body.

Example 12.8: Suppose that the relation for the body of (12.3) is the one in Figure 12.7. The scheme of that relation is $XYZ$, since the attributes correspond to the variables in (12.3). There are two tuples in the relation; in the first, all three components are simple constants, while in the second, all three happen to be nontrivial terms, consisting of function symbols and constants.

When we substitute the first tuple, $(a, b, c)$, for $(X, Y, Z)$ in the head of (12.3), the head becomes $p(f(a), g(a, b))$. The arguments of this atom become the first and second components of a tuple in the relation for the head. When we make a similar substitution for the second tuple, we get a second tuple for the head relation, which thus is the relation shown in Figure 12.9.

\[
\begin{array}{cc}
P_1 & P_2 \\
f(a) & g(b, c) \\
f(f(a)) & g(f(a), g(b, c))
\end{array}
\]

Figure 12.9 Resulting relation for the head of (12.3).

We shall summarize the above operations in the following algorithm, which computes the relation for a rule head from the relations for the subgoals. In addition to combining the ATOV and VTOA operations, we must include a join of the converted relations for the ordinary subgoals (those not defined by an arithmetic comparison such as $>$), and we must perform a selection for all the built-in (arithmetic comparison) subgoals.

Algorithm 12.2: Construction of the Relation for a Rule Head from Relations for its Subgoals.

INPUT: Relations $R_1, \ldots, R_n$ for subgoals $G_1, \ldots, G_n$ of a given safe rule $r$.

OUTPUT: The relation for the head of rule $r$.

METHOD: Let $X_1, \ldots, X_n$ be the entire list of variables appearing among $G_1, \ldots, G_n$.

1. For each ordinary subgoal $G_i$, use the algorithm of Figure 12.5 to convert relation $R_i$, which is a relation over the arguments of the predicate of $G_i$, into a relation $Q_i = \text{ATOV}(G_i, R_i)$ over the variables that appear in $G_i$.

2. Compute the join $Q = Q_1 \bowtie \cdots \bowtie Q_n$ (omit $Q_i$ if $G_i$ is built-in).

3. Apply to $Q$ a selection for each of the built-in subgoals of $r$, if any. Note that, because rules are assumed safe, it must be that built-in subgoals can be expressed in terms of the attributes of $Q$. The result is the relation for the body of $r$.

4. Use the algorithm of Figure 12.8 to compute the relation for the head of $r$ from the relation for the body of $r$.

In practice, steps (2) to (4) will be combined, so that we never compute the relation over all the attributes of the rule. Rather, as we compute the partial results $Q_1 \bowtie \cdots \bowtie Q_n$, for $i = 2, 3, \ldots, k$, we project out unnecessary attributes and perform selections as soon as possible. An attribute (which is a variable of $r$) is only necessary if it appears either in the head of $r$, or in a later subgoal, $G_j$, for some $j > i$. The matter is discussed in more detail in Section 12.6.

Naive Evaluation

Algorithm 12.2 forms the heart of a naive-evaluation algorithm that generalizes Algorithm 3.3. We state that algorithm formally here.

Algorithm 12.3: Naive Evaluation.

INPUT: A collection of safe rules, possibly involving function symbols, and relation $R_1, \ldots, R_n$ for the EDB predicates mentioned in the bodies of these rules.

OUTPUT: If it is finite, the least fixed point of the rules, with respect to the given EDB relations. If that least fixed point is infinite, we produce an infinite sequence of approximations that approaches the least fixed point as a limit; that is, each tuple that is in an IDB relation of the least fixed point appears in some finite approximation, and in all approximations thereafter.
METHOD: We begin assuming the relation \( P \) for each IDB predicate \( p \) is empty. Suppose that at some time, we have approximations \( P_1, \ldots, P_n \) for the IDB predicates \( p_1, \ldots, p_n \). We obtain the next approximation for \( p_i \) by computing

\[ P'_i = \text{EVAL}(p_i, R_1, \ldots, R_k, P_1, \ldots, P_n) \]

as follows.

1. For each rule for \( p_i \), apply Algorithm 12.2, using the appropriate relation among \( R_1, \ldots, R_k \) and \( P_1, \ldots, P_n \) for each ordinary subgoal of that rule.
2. Take the union, over all rules for \( p_i \), of the relations constructed in (1).

The result is \( P'_i \).

We then compare the \( P'_i \)'s with the \( P_i \)'s. Since \( \text{EVAL} \) is a monotone operation (see Section 3.4), even with its extended meaning used in this section, we know that \( P_i \subseteq P'_i \) for all \( i \). If \( P'_i = P_i \) for all \( i \), then we have converged to the least fixed point, and we may halt. If some \( P_i \) is a proper subset of \( P'_i \), then we must replace the \( P_i \)'s by their corresponding \( P'_i \)'s, and repeat the process, just described, obtaining new approximations for the IDB relations. Note that, when there are function symbols in the rules, it is possible that we never reach a round where the \( P_i \)'s and the \( P'_i \)'s are equal; in that case, this "algorithm" runs forever, producing progressively larger approximations to the infinite least fixed point.

### Semi-Naive Evaluation

Corresponding to Algorithm 12.3 there is a more efficient variant, just as Algorithm 3.4 corresponds to Algorithm 3.3 in Chapter 3. We shall review the essence here, which is to "differentiate" the rules. That is, for each IDB predicate \( p \) we have another IDB predicate \( \Delta P \), representing the change to \( p \) in one round. That is, the relation for \( \Delta P \) is intended to be the difference \( P'_i - P_i \) in Algorithm 12.3.

We can differentiate a rule in a manner reminiscent of the product rule of differential calculus. That is, we replace a rule

\[ H ::= G_1 \land \cdots \land G_n, \]

having one or more IDB subgoals (subgoals with an IDB predicate), by one rule for each IDB subgoal. If \( G_i \) is an IDB subgoal, then we have rule

\[ \Delta H ::= G_1 \land \cdots \land G_{i-1} \land \Delta G_i \land G_{i+1} \land \cdots \land G_n. \]

**Example 12.9:** Consider rule (7) of Example 12.3:

\[ \text{true}(\text{and}(X,Y)) ::= \text{true}(X) \land \text{true}(Y). \]

Both subgoals are IDB subgoals, so we have two rules

\[ \Delta \text{true}(\text{and}(X,Y)) ::= \Delta \text{true}(X) \land \text{true}(Y), \]

\[ \Delta \text{true}(\text{and}(X,Y)) ::= \text{true}(X) \land \Delta \text{true}(Y). \]

Together, they define the predicate \( \Delta \text{true} \) in terms of itself and the predicate \( \text{true} \).

The intuitive idea behind semi-naive evaluation is that on each round, we want \( \Delta P \) to have exactly the tuples that were discovered for \( p \) on the previous round of naive evaluation. A "new" tuple for \( p \), that is, one not discovered on a previous round, must use at least one tuple that was not discovered before the previous round. Thus, each new tuple for \( p \) on one round of naive evaluation will be produced by one of the differentiated rules for \( p \); it might be produced by more than one, if it uses more than one new tuple. Note also that a "new" tuple might not really be new, since it may have been produced by another rule for \( p \) previously. The technique is summarized in the next algorithm.

**Algorithm 12.4:** Semi-Naive Evaluation.

**INPUT:** A collection of safe rules, possibly involving function symbols, and relation \( R_1, \ldots, R_k \) for the IDB predicates mentioned in the bodies of these rules.

**OUTPUT:** If it is finite, the least fixed point of the rules, with respect to the given IDB relations. Otherwise, we produce an infinite sequence of approximations, as in Algorithm 12.3.

**METHOD:** We begin by constructing the differentiated rules for all the IDB predicates, as described above. The algorithm consists of a loop, possibly infinite, in which we produce successive approximations to the IDB relations. We initialize the relation \( P \) for each IDB predicate \( p \) to be empty, and we initialize the relation \( \Delta P \) for \( \Delta p \) by applying the EVAL procedure of Algorithm 12.3, but only to those rules that have no IDB subgoals (others would produce no contribution anyway). We then execute the following steps in a loop.

1. If all the \( \Delta P \)'s are empty, then break the loop. Here, we have converged to the least fixed point, which will be the relation \( P \) for each IDB predicate \( p \).
2. Otherwise, replace each IDB relation \( P \) by \( P \cup \Delta P \).
3. For each IDB predicate \( p \), compute a new differential relation \( \Delta P' \) by applying Algorithm 12.2 to each of the differentiated rules for \( p \), using the EDB relations \( R_1, \ldots, R_k \), the current relations \( P_1, \ldots, P_n \) for the IDB predicates, and the differential relations \( \Delta P_1, \ldots, \Delta P_n \) as needed, and taking the union over all the differentiated rules for \( p \).
4. For each IDB relation \( p \), compute \( \Delta P = \Delta P' - P \), and go to step (1).
by starting with only the facts of the EDB and repeatedly applying rules from right to left, that is, from body to head. This type of computation is often called bottom-up, because we implicitly construct trees that describe the proof of an IDB tuple from the leaves up, working toward the root. That true has a node for each tuple used, and the children of the node for tuple \( \mu \) are the tuples from the relations for each of the subgoals that were used to construct \( \mu \) by applying some rule.\(^6\)

**Example 12.10.** Let us consider the rules of Example 12.3. Figure 12.10 shows a proof tree for the fact \( \text{true}(g(a,d)) \). This fact was constructed by rule (2) from the facts \( \text{true}(g(a,e)) \) and \( \text{true}(g(c,d)) \); these were taken from the relations for the first and second subgoals of rule (2), respectively. In turn, they were each placed in that relation by rule (1), since \( g(a,e) \) and \( g(c,d) \) are given EDB facts. Notice that the order in which these tuples are placed in their respective relations by Algorithm 12.3 is bottom-up according to the tree of Figure 12.10. That is, the leaves \( g(a,e) \) and \( g(c,d) \) are in their EDB relations initially. Then, the atoms on the second level of the tree, \( \text{true}(g(a,e)) \) and \( \text{true}(g(c,d)) \), are inferred on pass 1, while the root is inferred on the second pass.\( \square \)

![Figure 12.10 Proof tree.](image)

Bottom-up processing of logic is sometimes called forward-chaining. The explanation for this location is that rules are sometimes regarded as having a "natural" direction in which they should be applied, that is, from body to head, or from antecedent to consequent: "if such-and-such is true, then this is also true." Reasoning in this way is thought of as "forward" reasoning.

---

\(^6\) Technically, the proof tree might be a DAG (directed, acyclic graph), since a tuple could be used in several places. Alternatively, we can allow a tuple \( \mu \) to appear at several different nodes, and repeat parts of a proof if necessary. If we do so, then the proof tree may be assumed to be a true tree.

---

**12.3 TOP-DOWN PROCESSING OF LOGIC**

**Reasoning from Goals**

There is another mode of reasoning, called top-down or backward-chaining, in which we start with the goal we would like to prove, and we consider ways in which we might succeed in proving the goal. These familiar with Prolog are aware that language uses top-down reasoning with an order of processing goals and their subgoals determined by the order in which rules are listed in the Prolog program. The following is an example that happens to mimic Prolog's algorithm of search, but an understanding of Prolog is not needed to follow the example.

**Example 12.11: Suppose we are given the rules of Example 12.3 and the goal \( \text{true}(g(a,d)) \) to prove. One way to do so is to apply rule (1) of Example 12.3,**

\[ (1) \quad \text{true}(g(S,T)) \rightarrow g(S,T), \]

\[ \text{in such a way that the head becomes the desired goal. To deduce what tuple would have to be in the relation for rule (1), we term-match the head } \]

\[ \text{true}(g(S,T)) \quad \text{against the goal, true}(g(a,d)), \]

\[ \text{finding } S = a \text{ and } T = d. \]

\[ \text{Thus, to apply rule (1) in such a way that tuple true}(g(a,d)) \]

\[ \text{is generated, we need a tuple (a,d) in the relation for rule (1); recall that relation has two components, corresponding to variables S and T, in that order. The only way we could get } \]

\[ S = a \text{ and } T = d \text{ is if the subgoal g(a,d) were a tuple of the relation GO. But } \]

\[ \text{GO is an EDB relation, and we suppose it consists of the ten tuples listed in Example 12.3. These tuples do not include (a,d), so rule (1) does not help us achieve the goal } \]

\[ \text{true}(g(a,d)). \]

\[ \text{Now, let us consider using rule (2) of Example 12.3,} \]

\[ (2) \quad \text{true}(g(S,T)) \rightarrow \text{true}(g(S,U)) \land \text{true}(g(U,T)). \]

\[ \text{to prove our initial goal, true}(g(a,d)). \]

\[ \text{The head of rule (2) is the same as the head of rule (1), so we again match } S = a \text{ and } T = d. \]

\[ \text{If we make these substitutions in the body of rule (2) we get subgoals true}(g(a,U)) \text{ and true}(g(U,d)). \]

\[ \text{We work on each of these in turn, first trying rule (1), then (2). When we match the head of rule (1) against true}(g(a,U)), \]

\[ \text{we find something we have not seen before: both formulas have variables. The goal is really asking for some value of } U \]

\[ \text{for which true}(g(a,U)) \text{ can be proved. In this case, the way} \]

\[ \text{to match is obvious; } S = a \text{ in the head of rule (1) matches } a, \text{ and } T \text{ matches } U. \]

\[ \text{If we substitute } a \text{ for } S \text{ and } U \text{ for } T, \text{ the body of rule (1) becomes } g(a,U). \]

\[ \text{Since } g \text{ is an EDB predicate, we can only satisfy the goal by finding a value of } U \text{ such that } (a,U) \text{ is a tuple of GO. In this case, there is exactly one value that } U \text{ can take, namely, } e, \]
such values.

The binding of \( U \) to value \( e \), which occurred when we considered the first subgoal of rule (2), affects the second subgoal, which was true\( (g(U, d)) \) but now becomes true\( (g(e, d)) \). We attempt to satisfy it too, first by rule (1). That task is easy, since the body of rule (1) becomes \( g(e, d) \), and \( (e, d) \) is a tuple of the EDB relation \( GO \). We have now satisfied all subgoals of rule (2), and the original goal has been proved. Its proof is the tree of Figure 12.10. 

**Backtracking**

In Example 12.11, we proceeded fairly directly from goal to proof. The only detour we took was that our initial attempt to prove the goal true\( (g(a, d)) \) by rule (1) failed. However, there are many other places we could have failed. For example, if \( g(e, d) \) were not an EDB fact, then the subgoal true\( (g(e, d)) \) would fail. We would have to backtrack to the first subgoal of rule (2) and try to find another value of \( U \) besides \( e \).

Since there are no more tuples of the form \( (a, U) \) in \( GO \), we would try to achieve goal true\( (g(a, U)) \) by applying rule (2). Before doing so, we should understand that the use of \( U \) in the goal and in rule (2) are not the same. Rather, \( U \) in rule (2) is a "local variable," with the same name, perhaps, as a variable of the goal, but not thereby identical to it. Thus, to avoid confusion, we should rewrite rule (2) for its application to subgoal true\( (g(a, U)) \) with a new variable in place of \( U \), for example,

\[
\text{true}(g(S, T)) := \text{true}(g(S, Y)) \land \text{true}(g(Y, T)).
\]

When we match (technically we "unify") the head of this rule with the goal \( \text{true}(g(a, U)) \), we find \( S = a \) and \( T = U \). We are thus sent to achieve goal \( \text{true}(g(a, V)) \), which is really no different from the goal \( \text{true}(g(a, U)) \) that we were trying to achieve.

This exploration illustrates the point that top-down search, with backtracking to try alternative rules when one fails, can easily lead to a loop. In fact, loops can occur when performing a top-down search, even when the rules are data-dependent. Then a bottom-up solution will always converge. For example, the rules for transitive closure of a graph

\[
\text{path}(X, Y) := \text{edge}(X, Y),
\]

\[
\text{path}(X, Y) := \text{path}(X, Z) \land \text{path}(Z, Y).
\]

are really no different from rules (1) and (2) of Example 12.3. When we start with a goal like \( \text{path}(a, b) \), where for no value of \( W \) is \( (a, W) \) a tuple of the EDB relation for \( \text{edge} \), we go into the same sort of loop as described above, generating goals \( \text{path}(a, Z), \text{path}(a, Z_1), \text{path}(a, Z_2), \ldots \). In fact, the same thing would happen, by a more complex process, even if \( \text{edge}(a, W) \) were true for some value of \( W \), as long as there is no path from \( a \) to \( b \) in the graph.

### 12.3 Top-down Processing of Logic

We shall not discuss in detail the backtracking algorithm used by Prolog to search for ways to achieve a goal. Rather, we shall give one more example where backtracking is used so top-down processing of rules can obtain complete relations, rather than single tuples that match a given goal. Beginning in Section 12.5, we shall discuss more efficient ways to process complete relations in a top-down manner.

**Example 12.12:** Suppose we wished to take the natural join

\[
R(A, B) \bowtie S(B, C)
\]

of two EDB relations \( R \) and \( S \). We could use a rule like

\[
j(A, B, C) :- r(A, B) \land s(B, C).
\]

and apply it to a goal \( j(X, Y, Z) \). However, what happens during a Prolog-like, top-down search is that each tuple \( (a, b) \) in \( R \) is tried in turn, until one with a value of \( b \) that matched some tuple \( (b, c) \) in \( S \) is found. At that point, the goal is satisfied, with \( X = a, Y = b, \) and \( Z = c \).

That is not exactly what the join means; it asks for all pairs of matching tuples from \( R \) and \( S \). If we processed the above rule bottom-up, say by Algorithm 3.2, we would indeed get all pairs of matching tuples. We can fix the problem by introducing a subgoal \( \text{fail} \), as is used in Prolog, to force backtracking to occur repeatedly. That is, we write

\[
j(A, B, C) :- r(A, B) \land s(B, C) \land \text{fail}.
\]

The effect of this rule with goal \( j(X, Y, Z) \), processed top-down, is that we begin by taking the first tuple of \( R \), say \( (a_1, b_1) \), and looking for tuples of \( S \) that match \( b_1 \). When we find one, say \( (b_1, c_1) \), we proceed to the third subgoal, \( \text{fail} \), which causes us to backtrack to the second subgoal, \( s(B, C) \), again. On backtracking, we look for another tuple matching \( b_1 \), say \( (b_1, c_2) \), in \( S \). We fail again when we reach the \( \text{fail} \) subgoal, and eventually on backtracking to the subgoal \( s(B, C) \) we shall find in \( S \) no more tuples with first component \( b_1 \).

At that point, we fail back to the first subgoal, which finds another tuple in \( R \), say \( (a_2, b_2) \). Now the same process repeats, eventually finding all the tuples of \( S \) with first component \( b_2 \). We then fail back to the first subgoal, obtain the third tuple of \( R \), and so on.

Eventually, we find no more tuples of \( R \). At that point, the entire rule has failed, but before doing so, it has generated all of the tuples in the join of \( R \) and \( S \). If we had inserted an action such as print before the \( \text{fail} \), then the entire

---

8 When \( \text{fail} \) appears as a goal in a Prolog program, it is treated as a subgoal that had failed to match anything.

9 In a real Prolog program, there would presumably be some sort of action that used the tuple, appearing to the left of the \( \text{fail} \). For example, we might print the tuple that was found.
Comparison of Top-Down and Bottom-Up Methods

We presume that the object of any algorithm for "processing" logic is to produce the relation, or relevant portion of a relation, that is needed to answer some query. Examples of queries that ask for parts of relations are

\[ \text{sum}(\text{succ}^5(0), \text{succ}^{25}(0), W) \]  

(12.4)

that is, what is the sum of 47 + 92, using the representation of addition given in Example 12.2, and

\[ \text{true}(\text{or}(g(a, b), g(n, c))) \]

referring to Example 12.3; this query asks whether it is true that either state a can become state b or that state a can become state c, or both.

For query (12.4), the bottom-up approach would start to compute the entire int and sum relations. Recall from Example 12.4, that after 141 passes, the fact that 47 + 92 = 139 would be established, so the value \( W = \text{succ}^{139}(0) \) would be found. In this case, the value of \( W \) is unique, but in general, there would be no way to tell that we did not need to proceed with additional passes to find other values of \( W \). Further, if we are given a query with no solution, then the bottom-up method would surely run forever looking for the first solution.

In contrast, the top-down method would consider rule (4) in Example 12.2, which is

\[ \text{sum}(X, \text{succ}(Y), \text{succ}(Z)) := \text{sum}(X, Y, Z) \]

and try to match (really "unify") its head with the goal

\[ \text{sum}(\text{succ}^8(0), \text{succ}^{20}(0), W) \]

This unification succeeds, with \( X = \text{succ}^8(0), Y = \text{succ}^{20}(0), \) and \( W = \text{succ}(Z) \). Thus, the subgoal in the body of rule (4) becomes

\[ \text{sum}(\text{succ}^8(0), \text{succ}^{20}(1), Z) \]

We then unify the head of rule (4) with this subgoal, after replacing \( Z \) in the rule by \( Z_1 \), to avoid a conflict of variables. This unification yields \( X = \text{succ}^8(0), Y = \text{succ}^{20}(0), \) and \( Z = \text{succ}(Z_1) \), leading to subgoal \( \text{sum}(\text{succ}^8(0), \text{succ}^{20}(0), Z_1) \).

After 90 more iterations, we reach subgoal \( \text{sum}(\text{succ}^8(0), 0, Z_0) \), and we have established the relationships among local variables introduced along the way, as follows:

\[ W = \text{succ}(Z), Z = \text{succ}(Z_1), Z_1 = \text{succ}(Z_2), \ldots, Z_{90} = \text{succ}(Z_{91}) \]

Now, with the middle argument of \( \text{sum} \) equal to 0, rule(3) of Example 12.2,

\[ \text{sum}(X, 0, Z) := \text{int}(X) \]

which has heretofore been unable to match any subgoal (because \( \text{succ}^0 \) and 0 are different nonvariables), suddenly unifies, setting up subgoal \( \text{int}(\text{succ}^{20}(0)) \). That subgoal leads to 47 applications of rule (2) and 1 application of rule (1) of Example 12.2, to establish the truth of \( \text{int}(\text{succ}^{20}(0)) \). We shall omit the details.

The unification of \( \text{sum}(\text{succ}^8(0), 0, Z_0) \) with the head of rule (3) also has the effect of establishing the substitution \( Z_{90} = \text{succ}^{80}(0) \). That value propagates, telling us \( Z_{90} = \text{succ}^{80}(0) \), and so on, until finally we find \( W = \text{succ}^{139}(0) \). Thus, we have our answer to the addition problem: 47 + 92 = 139. In finding this answer, the bottom-up method has focused on exactly the subgoals we needed to establish the answer. The amount of time spent deriving the answer is thus proportional to the numerical value of the answer.

In contrast, the bottom-up method, at the least, would establish all of the facts it could establish in 141 passes, which includes all of the facts

\[ \text{succ}^8(0), \text{succ}^{20}(0), \text{succ}^{21}(0) \]

where \( i + j \leq 139 \) This work is proportional to the square of the numerical value of the answer we obtain, and therefore proportional to the square of the time spent by the top-down method. Further, the bottom-up method uses space proportional to the number of facts it derives, while top-down uses space only proportional to the longest stack of subgoals it creates, which could be much less than the number of facts.

On the other hand, we should not conclude that the top-down method is generally superior to the bottom-up method. We noted earlier that on rules like rules (1) and (2) of Example 12.3, top-down processing can get stuck in a loop, while the bottom-up method will plod along and eventually get the answer.

There is another way that bottom-up can beat top-down, even when both get the answer eventually. Recall from Example 12.12 our discussion of how a join could be taken top-down. If we extend that idea to join three relations, \( R \bowtie S \bowtie T \), each having \( n \) tuples, then the top-down procedure will take \( O(n^3) \) time to explore all \( n^3 \) lists of three tuples, one from each relation. That is, suppose we write the rules for the join as

\[ j_2(A, B, C, D) := j_1(A, B) \bowtie j_3(B, C, D) \]

\[ j_3(B, C, D) := r(B, C) \bowtie t(C, D) \]

Then the top-down method will, for each tuple in \( R \), compute the entire join \( j_2 \) anew.\(^{10}\)

However, if we compute \( R \bowtie S \bowtie T \) bottom-up, we take much less time,

\(^{10}\) That would be true even if we substituted for \( j_1 \) in the first rule to make a single rule with three subgoals, \( r, s, \) and \( t \).
unless the join is really a Cartesian product and the output is as large as \( O(n^2) \).

If we recall Section 11.5, we see that the join of two relations of size \( n \) can be computed in time \( O(n \log n) \) plus the time to write the output. For example, if the image sizes for attributes \( B \) and \( C \) are each \( n \), then the join \( S \bowtie T \) has size \( O(n) \), and so does the join of all three relations. Thus, the entire join can be computed in \( O(n \log n) \) time bottom-up.

A third disadvantage of top-down methods is that they require unification. In contrast, we can evaluate logical rules bottom-up using the simpler operation of term-matching, as we saw in Section 12.2.

We shall see in Chapter 13 that top-down is never better than bottom-up, when the best techniques are used in each case. In particular, the "magic-sets" rule-re-writing technique converts any set of rules into ones that can be evaluated as fast by semi-naive evaluation as the original rules were evaluated top-down. Moreover, there are many cases where the transformed rules are much more efficiently evaluable than were the old rules.

### 12.4 UNIFICATION

The essential step when processing logical rules top-down is unification, the operation whereby we take two atomic formulas, each with variables, and find a substitution for each variable, so that the two formulas become identical. Such a substitution is called a unifier. Unlike term matching, in unification, the expressions we substitute for variables will often themselves involve variables. Thus, any expressions could be substituted for these variables and the resulting formulas would still be identical. As a consequence, there may be more than one unifier for a pair of formulas.

We shall give an algorithm that produces the most general unifier (MGU) of two formulas. That substitution has the property that it is a unifier, and any other unifier can be constructed from the MGU by substitution for its variables. Thus, the MGU is the "simplest" substitution that makes the two formulas identical.

In the algorithm that follows, we tacitly assume that the first atomic formula is a goal, and the second the head of a rule that we hope unifies with the goal, and therefore might help to deduce the goal. This viewpoint has two consequences.

1. Since the variables of the rule are "local," we shall assume that there are no variables in common between the two formulas.
2. If we have a choice of substituting a variable of the first formula for a variable of the second or vice versa, we substitute for the variable of the second. This choice makes it more transparent when variables of a goal carry forward to their subgoals, but it is not essential for top-down processing.

#### Algorithm 12.5: Computing the Most General Unifier

**INPUT:** Atomic formulas \( F \) and \( G \) with disjoint sets of variables.

**OUTPUT:** The most general unifier of \( F \) and \( G \) or an indication that none exists.

**METHOD:** There are two phases. In the first, we determine which subexpressions of the formulas \( F \) and \( G \) must become identical in the MGU. In the second phase, we determine whether an MGU exists. We check for an immediate contradiction, where two identical subexpressions have different nonvariable symbols that must be equated, and if we find such a pair, we conclude that there is no MGU. We then attempt to construct the MGU, which must exist in principle, but may be infinite. If this attempt to construct the MGU fails, then we conclude that it is infinite, and therefore not usable. In that case, we say that "no MGU exists."

**Phase 1:** Finding equivalent subexpressions. The first step is to build a tree for each of \( F \) and \( G \), representing their structure as expressions in the obvious way. That is, we define the tree for term or atomic formula \( T \) recursively, as follows.

1. If \( T \) is a variable or a constant, then the tree for \( T \) is a single node labeled \( T \).
2. If \( T \) is of the form \( \theta(T_1, \ldots, T_k) \), where \( \theta \) is a predicate or function symbol, and \( T_1, \ldots, T_k \) are terms, then the tree for \( T \) has a root labeled \( \theta \) and has children that are the roots of the trees for \( T_1, \ldots, T_k \), in that order, from the left.

Now, having built the trees for \( F \) and \( G \), we attempt to group their nodes into equivalence classes, represented by the equivalence relation \( \equiv \). If we are unsuccessful in completing the construction of \( \equiv \) by the rules below, then no unifier exists. Intuitively, \( n \equiv m \) if and only if, in any unifier the expressions represented by the subtrees with roots \( n \) and \( m \) must be made identical. The rules defining \( \equiv \) are

1. If \( r_F \) and \( r_G \) are the roots of the two trees, then \( r_F \equiv r_G \). This rule follows from the definition of unification; we require that the two atomic formulas become the same.
2. If \( n \equiv m \), then nodes \( n \) and \( m \) must have the same number of children; if not there is no unifier for \( F \) and \( G \). If they have children \( c_1, \ldots, c_n \) and \( d_1, \ldots, d_m \), respectively, then \( c_i \equiv d_i \) for all \( i \). The motivation for this rule is that in order for the expressions of \( n \) and \( m \) to be made identical, they must have the same number of subexpressions, and corresponding

---

11. Technically, the MGU is only unique up to renaming of variables, but we shall speak of "the" MGU, since the particular variables used are not significant.

12. We must consider operators that are both function symbols and predicates, even though it is only at the outermost level that predicates appear.
subexpressions must be made identical.

c) If \( n \) and \( m \) are labeled by the same variable, then \( n \equiv m \). The reason is that we may choose only one substitution for a variable \( X \), so every node labeled \( X \) must become the same expression under any unifier.

d) \( n \equiv n \) for any node \( n \); if \( n \equiv m \), then \( m \equiv n \), and if \( n \equiv m \) and \( m \equiv p \), then \( n \equiv p \). These rules reflect the basic properties of the notion "becomes the same thing." For example, if \( n \) "becomes the same thing" as \( m \), then \( m \) "becomes the same thing" as \( n \). Thus, \( \equiv \) is an equivalence relation; that is, \( \equiv \) partitions the nodes into disjoint sets such that nodes in the same set "become the same thing" while nodes not in the same set need not "become the same thing."

Phase II: Testing for, and constructing the MGU. Having grouped the nodes into equivalence classes by repeatedly applying (a) through (d) until no more nodes can be grouped together, we consider each equivalence class in turn. First, if any class has two nodes that are labeled by two different symbols, neither of which is a variable, then there is no unifier. The reason is that no substitution for variables can make two subexpressions be identical, if their roots are different symbols and neither is a variable.

Assuming a unifier is still possible, we try to find an order to treat the equivalence classes and find for each class a representative expression. We use \( \tau \) for the MGU, and initially, we let \( \tau \) be undefined for each variable. The following rules define \( \tau \) for some of the variables; if there is an MGU, then they will eventually define all of the variables. The first rule serves as a basis and the second can be applied repeatedly.

1. If a class contains only nodes labeled by variables, pick one, preferring a variable from the first expression if there is one. Let the selected variable be \( X \). Then for each variable \( Y \) labeling a node in the class, set \( \tau(Y) = X \). Note \( \tau(X) = X \) as a consequence.

2. Suppose a class contains one or more nodes labeled by variables and also at least one node labeled by a constant, function symbol, or predicate. Suppose also, that one of the latter nodes represents an expression \( E \) in which all variables, if any, have had \( \tau \) defined for them.\(^{13}\) Let \( \tau(E) \) be the result of substituting \( \tau(X) \) for all occurrences of \( X \) in \( E \), for each variable \( X \). Then for every variable \( Y \) labeling a node in this class, let \( \tau(Y) = \tau(E) \).

If the above steps succeed in defining \( \tau(X) \) for all variables \( X \), then \( \tau \) is the MGU; otherwise, there is no MGU; that is, the given atomic formulas are not unifiable. \( \square \)

\(^{13}\) An important part of the proof of correctness for this algorithm is showing that it does not matter which of two expressions in the same equivalence class we choose.

12.4 Unification

Several examples where unifiers do or do not exist should make the steps of Algorithm 12.5 clearer.

Example 12.13: First, we consider a fairly complicated case where the MGU does exist. The two atomic formulas are

\[ p\left(f(X, g(X)), g(g(Y))\right) \]

and

\[ p\left(f(g(U), V), g(V)\right) \]

The trees representing these expressions are shown in Figure 12.11, and the nodes are numbered 1 through 15 for reference.

![Figure 12.11 Expression trees for atomic formulas of Example 12.13.](image)

We now group the nodes into equivalence classes. Rule (a) says \( 1 \equiv 9 \), because these are the roots of the two formulas. Rule (c) says \( 4 \equiv 7 \) because both are labeled with variable \( X \), and \( 13 \equiv 14 \) because both are labeled \( V \). Rule (b) tells us that the corresponding children of 1 and 9 are equivalent; that is, \( 2 \equiv 10 \) and \( 3 \equiv 11 \). Likewise, corresponding children of the equivalent pair 2 and 10 are equivalent, so \( 4 \equiv 12 \) and \( 5 \equiv 13 \).

Now, we can use rule (d) to infer that 4, 7, and 12 all belong in one equivalence class, that is, each is equivalent to the other two. The reason is that \( 4 \equiv 7 \) and \( 4 \equiv 12 \) have both been found. Similarly, \( 13 \equiv 14 \) and \( 5 \equiv 13 \) tell us that 5, 13, and 14 are all equivalent.

Since \( 3 \equiv 11 \), rule (b) says that their children are also equivalent; that is, \( 6 \equiv 14 \). Thus, by (d), 6 is also equivalent to 5 and 14.

Now, since \( 5 \equiv 6 \), rule (b) says that their children are equivalent, so \( 7 \equiv 8 \). Thus, 8 joins the class of 4, 7, and 12, and the equivalence classes are

\[
\begin{align*}
4 & \equiv 7 \equiv 8 \equiv 12 \\
1 & \equiv 9 \\
2 & \equiv 10 \\
5 & \equiv 6 \equiv 13 \equiv 14 \\
3 & \equiv 11 \\
15 & 
\end{align*}
\]

Now, no more equivalent pairs can be found by rules (a) to (d), so we are done with this phase. A straightforward check shows that no equivalence class
contains two distinct nonvariable symbols. For example, the nodes of the class \{5,6,13,14\} are all labeled with \( g \) or \( V \), and of these, only \( g \) is a nonvariable.

Now, we try to define the substitution \( \tau \). The class containing 18 alone has only variables, so we pick one, the lone choice being \( U \), and we define \( \tau(U) = U \). Next, we can consider the class \( 4 \equiv 7 \equiv 8 \equiv 12 \). All of the nodes but 12 are labeled by variables \( X \) or \( Y \), and node 12 represents expression \( g(U) \), whose only variable has had \( \tau \) defined. We therefore set \( \tau(X) \) and \( \tau(Y) \) to

\[
\tau(g(U)) = g(\tau(U)) = g(U)
\]

Next consider the class \( 5 \equiv 6 \equiv 13 \equiv 14 \). Two of the nodes, 13 and 14, are labeled \( V \), while the other two represent expressions \( g(X) \) and \( g(Y) \). Both of these expressions have \( \tau \) defined for their variables, and whichever we choose we get \( \tau(V) = g(\tau(U)) \), since \( \tau(X) = \tau(Y) = g(U) \). We have now defined the MGU \( \tau \) completely by

\[
\begin{align*}
\tau(U) &= U \\
\tau(V) &= g(U) \\
\tau(Y) &= g(g(U))
\end{align*}
\]

We get the atomic formula

\[
p(\langle f(g(U), g(g(U))) \rangle, g(g(g(U))))
\]

when we make this substitution in either of the given atomic formulas. \( \square \)

**Example 12.14:** Now let us consider a pair of formulas that do not have a unifier. Consider \( p(X,X) \) and \( p(f(Y), g(Z)) \). Their trees are shown in Figure 12.12, and the equivalence classes we obtain by applying rules (a) to (d) of Algorithm 12.5 are

\[
\begin{align*}
7 &= 8 \\
1 &\equiv 4 \\
2 &\equiv 3 \equiv 5 \equiv 6
\end{align*}
\]

Since \( 5 \equiv 6 \), but 5 and 6 have different function symbols as labels, there is no way to construct a unifier. Intuitively, these expressions cannot be unified because there is no way \( X \) can become "\( f \) of something" and also "\( g \) of something" at the same time. It should be clearly understood that only variables are subject to substitution, so we cannot substitute for \( f \) or \( g \). \( \square \)

**Figure 12.12** Expression trees for nonunifiable formulas.

**Example 12.15:** Finally, let us consider a very subtle case, where the equivalence classes do not have two nodes labeled by different nonvariables, yet we cannot find a unifier because there is no variable with which we can start to construct the substitution. The reason is that for each variable, the equivalence class containing the nodes labeled by that variable also contains a node corresponding to an expression that involves another variable. Thus, we never define \( \tau \) for any variable by rule (1) in Phase II of Algorithm 12.5.

Our two formulas are \( p(X, f(X)) \) and \( p(f(Y), Y) \). Their trees are shown in Figure 12.13, and their equivalence classes are

\[
\begin{align*}
1 &\equiv 5 \\
2 &\equiv 4 \equiv 6 \\
3 &\equiv 7 \equiv 8
\end{align*}
\]

The class for \( X \), which contains nodes 2, 4, and 6, cannot be used to define \( \tau(X) \) until the expression of node 6, that is \( f(Y) \), is defined, which means \( \tau(Y) \) must be defined before \( \tau(X) \). Similarly, consideration of the class 3 \( \equiv 7 \equiv 8 \) tells us that \( \tau(X) \) must be defined before \( \tau(Y) \), so that the expression of node 3 may be defined. Since rules (1) and (2) do not succeed in defining \( \tau \) for all variables, there is no unifier for these two expressions.

**Figure 12.13** Expression trees for formulas with infinite MGU.

In a sense, these formulas do have an MGU, but it is not finite. If we think of \( \tau(X) \) and \( \tau(Y) \) both being the infinite "term" \( f(f(\cdots(f(X))\cdots)) \), then both formulas become

\[
p(\langle f(f(\cdots(f(X))\cdots)) \rangle, f(f(\cdots(f(X))\cdots)))
\]

However, such a unifier is not very useful when we are trying to do top-down exploration of goals, so we do not consider infinite terms to be acceptable in unifiers. \( \square \)

We shall now sketch the reasons Algorithm 12.5 works. Many of the steps are left for the reader to work out or to consult one of several references on the subject.

**Theorem 12.1:** Algorithm 12.5 correctly computes the most general unifier of its given atomic formulas if it exists.
contains two distinct nonvariable symbols. For example, the nodes of the class \{5, 6, 13, 14\} are all labeled with \( g \) or \( V \), and of these, only \( g \) is a nonvariable.

Now, we try to define the substitution \( \tau \). The class containing 15 alone has only variables, so we pick one, the lone choice being \( U \), and we define \( \tau(U) = U \). Next, we can consider the class \( 4 \equiv 7 \equiv 8 \equiv 12 \). All of the nodes but 12 are labeled by variables \( X \) or \( Y \), and node 12 represents expression \( g(U) \), whose only variable has had \( \tau \) defined. We therefore set \( \tau(X) \) and \( \tau(Y) \) to

\[
\tau(g(U)) = g(\tau(U)) = g(U)
\]

Next consider the class \( 5 \equiv 6 \equiv 13 \equiv 14 \). Two of the nodes, 13 and 14, are labeled \( V \), while the other two represent expressions \( g(X) \) and \( g(Y) \). Both of these expressions have \( \tau \) defined for their variables, and whichever we choose we get \( \tau(V) = g(\tau(U)) \), since \( \tau(X) = \tau(Y) = g(U) \). We have now defined the MGU \( \tau \) completely by

\[
\begin{align*}
\tau(U) &= U \\
\tau(X) &= g(U) \\
\tau(Y) &= g(U) \\
\tau(V) &= g(g(U))
\end{align*}
\]

We get the atomic formula

\[
p(f(g(U), g(g(U))), g(g(g(U))))
\]

when we make this substitution in either of the given atomic formulas.

Example 12.14: Now let us consider a pair of formulas that do not have a unifier. Consider \( p(X, X) \) and \( p(f(Y), g(Z)) \). Their trees are shown in Figure 12.12, and the equivalence classes we obtain by applying rules (a) to (d) of Algorithm 12.5 are

\[
\begin{align*}
7 &
8 & 1 
4 & 2 
3 & 5 
6 & 6
\end{align*}
\]

Since 5 \( \equiv \) 6, but 5 and 6 have different function symbols as labels, there is no way to construct a unifier. Intuitively, these expressions cannot be unified because there is no way \( X \) can become "f of something" and also "g of something" at the same time. It should be clearly understood that only variables are subject to substitution, so we cannot substitute for \( f \) or \( g \).

Figure 12.12 Expression trees for nonunifiable formulas.

Example 12.15: Finally, let us consider a very subtle case, where the equivalence classes do not have two nodes labeled by different nonvariables, yet we cannot find a unifier because there is no variable with which we can start to construct the substitution. The reason is that for each variable, the equivalence class containing the nodes labeled by that variable also contains a node corresponding to an expression that involves another variable. Thus, we never define \( \tau \) for any variable by rule (1) in Phase II of Algorithm 12.5.

Our two formulas are \( p(X, f(X)) \) and \( p(f(Y), Y) \). Their trees are shown in Figure 12.13, and their equivalence classes are

\[
\begin{align*}
1 & \equiv 5 \\
2 & \equiv 4 \equiv 6 \\
3 & \equiv 7 \\
8 & \equiv 8
\end{align*}
\]

The class for \( X \), which contains nodes 2, 4, and 6, cannot be used to define \( \tau(X) \) unless the expression of node 6, that is \( f(Y) \), is defined, which means \( \tau(Y) \) must be defined before \( \tau(X) \). Similarly, consideration of the class \( 3 \equiv 7 \equiv 8 \) tells us that \( \tau(X) \) must be defined before \( \tau(Y) \), so that the expression of node 3 may be defined. Since rules (1) and (2) do not succeed in defining \( \tau \) for all variables, there is no unifier for these two expressions.

Figure 12.13 Expression trees for formulas with infinite MGU.

In a sense, these formulas do have an MGU, but it is not finite. If we think of \( \tau(X) \) and \( \tau(Y) \) both being the infinite "term" \( f(f(...(f(X))...)) \), then both formulas become

\[
p(f(f(...(f(X))...)), f(f(...(f(X))...)))
\]

However, such a unifier is not very useful when we are trying to do top-down exploration of goals, so we do not consider infinite terms to be acceptable in unifiers.

We shall now sketch the reasons Algorithm 12.5 works. Many of the steps are left for the reader to work out or to consult one of several references on the subject.

Theorem 12.1: Algorithm 12.5 correctly computes the most general unifier of its given atomic formulas if it exists.
Proof: First, an easy induction on the number of times rules (a) to (d) are applied shows that when two nodes are made equivalent, their expressions are necessarily the same under any unifying substitution. This part is left for the reader. It has two consequences.

1. The unifier produced by the algorithm is the most general unifier.
2. If the algorithm says there is no unifier, because two equivalent nodes have different numbers of children or different, nonvariable labels, then there really is no unifier.

We next turn our attention to the proof that the substitution \( \tau \), if it is produced, does make the two expressions identical. The crux of the proof is showing that when there are two or more nodes in an equivalence class that are not labeled by variables, then \( \tau \) makes both of their expressions the same, and therefore, it did not matter which one we chose to be \( \tau(Y) \) for each variable \( Y \) labeling a node in the class. Let \( n \equiv m \) be two such nodes. Then the labels of \( n \) and \( m \) are the same and the number of children each has is the same, or else Algorithm 12.5 would have declared that no unifier exists. Further, the pairs of corresponding children must be equivalent, by rule (b). Thus, we may prove by induction on the sum of the heights of the two nodes \( n \) and \( m \) that the substitution \( \tau \) makes their expressions the same. \( \square \)

12.5 THE RELATIONAL APPROACH TO TOP-DOWN LOGIC EVALUATION

In Section 12.3 we considered Prolog’s tuple-at-a-time approach to top-down query evaluation. We shall now consider a variation on this approach where backtracking is replaced by the “simultaneous” exploration of alternative rules for each goal. This section is an informal introduction to the technique. In the two following sections, the algorithms are stated formally and justified. The reader should not expect to grasp all the details immediately, but should follow the extended example of this section to get the flavor of the concepts involved.

We shall continue to treat subgoals of a single rule in order, from the left; in Section 12.10, we shall discuss how one might pick an appropriate order for the subgoals. When we assume “simultaneous” consideration of different rules, we are not supposing parallel implementation, but rather that computation time be shared among the various rules for a goal. In contrast, the Prolog style of evaluation does not work on the second rule until all exploration from the first rule has been completed.

Rule/Goal Trees

When exploring goals, their rules, and their subgoals, think of each consideration of a goal or rule as an “event.” We may build a tree of events, called a rule/goal tree, in which each event is represented by a node. Associated with each node is a description of the goal or rule, with variables named according to a convention given below. The rule/goal tree for a given goal, \( G_0 \), is constructed as follows.

1. The root is a goal node for \( G_0 \).
2. Suppose node \( n \) is a goal node for goal \( G \), and let \( G \) have predicate \( p \). Then for each rule \( r \) for \( p \), unify the head of \( r \) with \( G \). If the unification is successful, create a node for a modified version of rule \( r \) that is a child of node \( n \). Specifically, we perform the following steps to obtain the modified rule \( r \).
   a) First, rename variables of \( r \), if necessary, so \( r \) has no variables in common with \( G \).
   b) When we perform the unification of the head of \( r \) with \( G \), prefer to substitute for variables appearing in the head of \( r \), whenever possible.
   c) When we substitute for a variable \( X \) appearing in the head of \( r \), we must make that substitution for all occurrences of \( X \), not only in the head of \( r \), but throughout \( r \).
   d) All variables \( Y \) of \( r \) that do not appear in the head are “local” to \( r \). We identify these variables for this particular activation record for \( r \) by subscripting them with the number of ancestors of this activation record that are rule nodes, including the present node.\(^{14}\)
3. The children of a rule node are goal nodes corresponding to each of the subgoals in the body of the rule. At these goal nodes, as in the rule, the substitution of \( r(c) \) has been made.

Example 12.16: Consider the following rules for defining ancestors in terms of an EDB relation \( par \), where \( par(C, P) \) means that \( P \) is a parent of \( C \).

\[
\begin{align*}
r_1: & \quad \text{anc}(X, Y) \leftarrow \text{par}(X, Y). \\
r_2: & \quad \text{anc}(X, Y) \leftarrow \text{par}(X, Z) \land \text{anc}(Z, Y).
\end{align*}
\]

In words, \( r_1 \) says that \( Y \) is an ancestor of \( X \) if \( Y \) is a parent of \( X \) and \( Y \) is an ancestor of \( Z \).

Suppose the relation \( P \) for predicate \( par \) is as shown in Figure 12.14; edges represent pairs \((x, y)\) in \( P \), with the child below, and the parent above. Let us begin with the goal \( \text{anc}(j, W) \), that is, find all of \( j \)'s ancestors; this goal becomes the root of the rule/goal tree, the upper levels of which are shown in Figure 12.15.

We begin by creating children of the root for each of the rules \( r_1 \) and \( r_2 \). For \( r_1 \), we unify the head, \( \text{anc}(X, Y) \), with the goal \( \text{anc}(j, W) \), and we find that

\(^{14}\) As we shall see, goal nodes appear at odd-numbered levels, starting at the root, which is level 1, and rule nodes appear at even levels. Thus, the proper subscript is one-half the level number.
$X \equiv j$ and $Y \equiv W$.\footnote{We shall say that two terms are equivalent in a unification when we really mean that the nodes labeled with these terms are equivalent.} In the MGU we must use the constant $j$ in place of the variable $X$, but we can use either variable $W$ or $Y$. Since we wish to eliminate the variables of the head, we use $W$, so the MGU turns the head of $r_1$ into $\text{anc}(j, W)$, which happens to be the same as the goal. The same substitution of $j$ for $X$ and $W$ for $Y$ is made throughout rule $r_1$, so the instance of that rule at the first child of the root is

$$\text{anc}(j, W) := \text{par}(j, W).$$

When we consider $r_2$, we have the same head, $\text{anc}(X, Y)$ as for $r_1$, so we perform the same unification of the goal with the head, resulting in the same substitution of $j$ for $X$ and $W$ for $Y$. We apply this substitution to the entire second rule, and we create local variable $Z_1$ in place of $Z_1$ the subscript 1 is selected because we are at the first level of rule nodes. Thus, for the second child of the root we have instance

$$\text{anc}(j, W) := \text{par}(j, Z_1) \& \text{anc}(Z_1, W).$$

of rule $r_2$.

Now we consider the first child of the root, the child corresponding to rule $r_1$. This node has a child for each subgoal in the body of $r_1$. As there is only one subgoal, there is only one child, a goal node for $\text{par}(j, W)$. As $\text{par}$ is an EDB predicate, and therefore has no rules, this goal node is a leaf.

Now, consider the second child of the root, the one corresponding to $r_2$. It has two goal node children, the first for subgoal $\text{par}(j, Z_1)$ and the second for $\text{anc}(Z_1, W)$. The first of these, having an EDB predicate, is a leaf. The second, $\text{anc}(Z_1, W)$, has rule-node children for rules $r_1$ and $r_2$. We unify the heads of the two rules, which, recall, are both $\text{anc}(X, Y)$, with the subgoal $\text{anc}(Z_1, W)$.

When we unify the heads of $r_1$ and $r_2$ with this subgoal, the instances of the two rules at the rule-node children of goal node $\text{anc}(Z_1, W)$ become

\begin{align*}
\text{r}_1: & \quad \text{anc}(j, W) \Rightarrow \\
& \quad \text{par}(j, W). \\
\text{r}_2: & \quad \text{anc}(j, W) \\
& \quad \Rightarrow \text{par}(j, Z_1) \& \text{anc}(Z_1, W). \\
\text{par}(j, W) & \quad \Rightarrow \text{par}(j, Z_1) \& \text{anc}(Z_1, W). \\
\text{anc}(Z_1, W) & \quad \Rightarrow \text{par}(j, Z_1) \& \text{anc}(Z_1, W).
\end{align*}
Note that if we think of $j$ as $Z_j$, then (1), (3), and (4) become special cases of (2), (5), and (6), respectively.

As we mentioned, it is our intent to work on all rules for the same goal "in parallel," while we work on only one subgoal of each rule at a time, in left-to-right order. The notion of "in parallel," or "simultaneous" computation can be interpreted as breadth-first construction of the rule/goal tree. That is, we work on all nodes at one level and create their children, before working on any node at the next lower level. In practice, we shall not be concerned with the exact order in which we visit nodes in the rule/goal tree. Rather, we shall simply make sure that any node whose entire subtree can complete, will in fact complete eventually.

**Propagation of Bindings and Answers**

As we build the rule/goal tree, we construct certain relations at each node.

1. At each goal node $n$, we have a binding relation $M$. The goal at $n$ will have zero or more arguments that are "bound," in the sense that all variables appearing in that argument are given bound values, either from a constant, like $j$ in Figure 12.15 being passed from above, or by a finite set of tuples being passed "sideways," because they appear in a previous subgoal, as in Example 12.12 (Section 12.3). The tuples of $M$ represent bindings for these bound arguments.

2. At each rule node there are supplementary relations $S_0$, $S_1$, ..., with $S_i$ corresponding to the point in the rule body with $i$ subgoals to the left. The supplementary relations represent the bindings of those variables that have either been bound by the binding relation for the goal node's parent (applied to the rule's head) or that have appeared in a previous subgoal of the rule. Again, Example 12.12 illustrates the process of binding variables that we mimic with the supplementary relations.

3. Both rule and goal nodes have answer or result relations. In the case of a goal node, this relation is the set of tuples that match the goal and are provable from the rules, while for a rule node, this relation has those tuples that match the head of the rule and are provable by a sequence of rule applications, ending with the rule at that node. The result relation for the root goal node is the answer to the query.

**Example 12.17:** Let us consider the computation on the tree of Figure 12.15. Suppose we work for a while on the left child of the root. To compute the relation for that rule node, we must find the relation for the goal node $\text{par}(j, W)$. Since $\text{par}$ is an EDB predicate, we need only term-match the goal $\text{par}(j, W)$ to the tuples in the relation $P_i$; that is, we find those tuples with first (child)

---

The $M$ stands for "magic set," as discussed in Chapter 13.
scheme \( Z_1 \).

We now perform "sideways information passing" to establish that for the subgoal \( \text{anc}(Z_1, W) \), \( Z_1 \) can only have the values \( f \) and \( h \). In a sense, the first argument of \( \text{anc}(Z_1, W) \) has become bound to the unary relation \( \{f, h\} \), much as the first argument of \( \text{par}(j, W) \) was bound to \( j \). The difference is that in the latter case, the binding was established by unification, as we created the rule/goal tree; in the former, the binding is to a set of constants (or more generally, a set of tuples), and the binding is established by sideways information passing through the supplementary relations, after we have explored part of the tree.

With this binding for \( Z_1 \), we can explore the tree from \( \text{anc}(Z_1, W) \), and a fortuitous sequence of events leads us to terminate our exploration of the theoretically infinite tree. We find, by a process similar to that followed so far, that when we get to the goal node \( \text{anc}(Z_4, W) \), we have established the binding \( \{c, d, e\} \) for \( Z_2 \); note those are exactly the grandparents of \( j \) in Figure 12.12, just as the binding for \( Z_1 \) is the parents of \( j \). Then, when we explore to the goal node \( \text{anc}(Z_5, W) \) (not shown explicitly in Figure 12.15), we find that \( Z_5 \) is bound to the great-grandparents of \( j \), that is, \( \{a, b\} \). Lastly, when we reach goal node \( \text{anc}(Z_4, W) \), \( Z_4 \) is bound to \( \emptyset \).

At this point, we know there is no reason to explore further, since the relation for \( \text{anc}(Z_4, W) \) must be empty, no matter how far we explore. Notice how the sideways information passing has established that there are no tuples for the relation of subgoal \( \text{anc}(Z_4, W) \) that could yield answers to the original query. In effect, we have used a sequence of semijoins to discover that one subgoal in the rule/goal tree is empty, and, therefore, that the tree below this subgoal need not be explored. This occurrence was not altogether an accident. We only need to know that the \( \text{par} \) relation is finite and has no cycles, both of which are the case for Figure 12.14, and we are guaranteed that after a finite number of levels, some \( Z_i \) will get the binding \( \emptyset \).

Thus, we can calculate after a finite amount of time that the relation for goal node \( \text{anc}(Z_4, W) \) is

\[
\{fn, fe, fd, fa, fb, ha, hh\}
\]

(12.7)

This relation is translated into an identical relation with scheme \( Z,W \) for the subgoal \( \text{anc}(Z_4, W) \) in the instance (12.6) of \( r_2 \), and that relation is joined with relation \( \{f, h\} \), which has scheme \( Z_1 \), for subgoal \( \text{par}(j, Z_1) \) in (12.6). The result is a relation with scheme \( Z_2,W \) for the body of \( r_2 \) at the right child of the root; it is identical to relation (12.7), and its scheme is \( Z_2,W \).

Finally, we construct the relation for the head, \( \text{anc}(j, W) \), of instance (12.6) of \( r_2 \), by pairing the constant \( j \) with the second \( (W) \) components of the tuples in (12.7). These tuples, \( \{jc, je, ja,jh, jh\} \), together with the tuples \( jf \) and \( jh \) found earlier, form the complete relation for the root of Figure 12.15. □

**Why the Method Works**

It is not hard to show that, when we follow the above algorithm, every tuple implied to be in the answer relation for a subgoal is a fact that follows logically from the database and the rules. The converse is much harder to prove; it says that every fact \( F \) of the form \( p(t_1, \ldots, t_k) \), for predicate \( p \) and ground terms \( t_1, \ldots, t_k \), such that \( p(t_1, \ldots, t_k) \) unifies with a given subgoal, will actually be generated in the result relation for that subgoal after consideration of some number of levels, depending on \( F \), of the rule/goal tree. Intuitively, construction of the most general unifier places the least restriction on rule heads needed to guarantee that the rule will produce all of the facts that both

1. Follow from the rule, and
2. Imply the goal with which the head is unified.

We also need the observation that sideways information passing only restricts the possible values of variables by eliminating tuples that will not join with relations for the other subgoals anyway.

We shall not attempt to prove the completeness of the approach described in this section, that is, if a fact matches a goal, then it will be derived after a finite number of steps. Papers showing the completeness of top-down, or "resolution" methods of proving theorems are mentioned in the bibliographic notes.

If we accept as proven that the rule/goal tree expansion method uncovers all proofs of facts that are instances of the given query, then this algorithm is equivalent to bottom-up evaluation of rules, as long as rules are of a form where the least fixed point and the set of provable facts are the same. That class of rules includes Horn clauses, even those with function symbols, although when there are function symbols, neither the top-down nor bottom-up approaches is guaranteed to converge after a finite time, since there could be an infinite number of facts matching the query. When there are negated subgoals, even stratified ones, we cannot expect the set of provable facts to be equivalent to the "perfect" model, which was the one given in Section 3.6 (Volume I) as the meaning of stratified logic programs.

---

17 In practice, we would not instantiate the full relation for the body, but would try to perform projections as early as possible. For example, since the hypergraph for the subgoals of the body of \( r_2 \) is acyclic, we could use Algorithm 11.15 to construct the relation for the head directly.
There is a problem with the rule/goal tree expansion method of query evaluation, a problem with which we shall deal in Chapter 13. The method may be forced to explore an infinite tree, even though the answer to the query function symbols, we know that there are only finitely many possible answers, since every answer must be constructed from constants appearing in the rules and constants in the finite database.

Example 12.16: The ancestor rules of Examples 12.16 and 12.17 are dataflow rules. However, if the \( \text{par} \) relation is cyclic, then we must consider the entire, infinite rule/goal tree that is suggested by Figure 12.15. Of course, after some point, we shall never generate any more new facts to match the goal at the root, that we have the complete answer; that capability will be added in Chapter 13, when we combine top-down and bottom-up methods in the "magic-sets".

Comparing Rule/Goal Tree Expansion with Backtracking

It is worth noting that, unlike rule/goal tree expansion, the tuple-at-a-time backtracking scheme of Section 12.3 is not guaranteed to find a proof, even if one exists. Sometimes, backtracking gets into an infinite loop when following the first rule it tries for a goal, and thus never gets to consider other rules for the same goal. If the unified rules lead to proofs of facts matching that goal, make the comparison fair, we must assume that the backtracking algorithm is looking for all facts that match the query, not just for the first one. Thus, we implicitly fall after finding each successful match, as discussed in Example 12.12, for instance.

Example 12.19: If we reverse the order of the ancestor rules of Example 12.16, so they appear as

\[
\text{anc}(X,Y) ::= \text{par}(X,Z) \& \text{anc}(Z,Y).
\]

\[
\text{anc}(X,Y) ::= \text{par}(X,Y).
\]

then the first rule will send Prolog into a loop if the \( \text{par} \) relation has cycles. For example, if \( \text{par}(1,2) \) and \( \text{par}(2,1) \) are both true, then a query like \( \text{anc}(1, W) \) and \( \text{anc}(2, W) \), alternately, as the reader may check.

Moreover, if we reverse the order of the subgoals in the recursive rule, or

\[18\] One might question whether it makes sense to consider a cyclic parent-child relationship to paths in a graph, as was discussed in an example from Section 12.3. Surely, cyclic directed graphs are very common.

use two \( \text{anc} \) subgoals (analogous to the use of two \( \text{path} \) subgoals in Section 12.3), we do not change the least fixed point of the rules, but we create even more severe problems. The rules

\[
\text{anc}(X,Y) ::= \text{anc}(X,Z) \& \text{par}(Z,Y).
\]

\[
\text{anc}(X,Y) ::= \text{par}(X,Y).
\]

send Prolog into an infinite loop on any ancestor query, regardless of the EDB relation \( \text{par} \).

One might suppose that situations in which the top-down backtracking approach fails to explore the whole rule/goal tree could be avoided if we ordered the rules properly. For example, a Prolog programmer quickly learns to list the basis rules before the recursive rules, thereby avoiding problems such as those above. However, we can easily find ourselves in a situation where two rules for a predicate each lead both to some finitely provable facts and to an infinite loop from which the backtracking scheme never recovers. Rule/goal tree expansion will follow both infinite paths, but will also, after a finite time, prove all of the provable facts that match the query.

\[
p(X) ::= q(X).
\]

\[
p(X) ::= r(X).
\]

\[
q(X) ::= a(X).
\]

\[
q(X) ::= c(X,Y) \& q(Y).
\]

\[
r(X) ::= b(X).
\]

\[
r(X) ::= c(X,Y) \& r(Y).
\]

Figure 12.16 Example where backtracking cannot reach all provable facts.

Figure 12.16 gives an abstract example of this situation. Suppose \( a, b, \) and \( c \) are EDB relations, consisting of the facts \( a(3), b(4), c(1,2) \) and \( c(2,1) \), and our query is \( p(W) \), that is, find all of the \( p \) facts. If we start with the first rule for \( p \) in Figure 12.16, we shall immediately discover that \( p(3) \) is true, thanks to the rule \( q(X) ::= a(X) \). However, before we finish exploring from goal \( q(X) \), we must try the second rule for \( q \), which leads to an infinite loop, where we repeatedly explore the goal \( q(X) \), with \( X \) bound to \( 1,2 \). Thus, we never uncover the fact \( p(4) \).

Similarly, suppose we start with the rule \( p(X) ::= r(X) \) for \( p \). Then we shall uncover only \( p(4) \), before the recursive rule for \( r \) sends us into an infinite loop.

In contrast, if we construct and explore the rule/goal tree, we shall quickly discover both \( p(3) \) and \( p(4) \), after expanding the root goal for four levels, that is, after reaching the third level of goals. We also follow both infinite paths, one where goal \( q(X) \) repeats, and the other where \( r(X) \) repeats.

\[\square\]
12.6 COMPUTING RELATIONS DURING RULE/GOAL TREE EXPANSION

Now we shall take a closer look at the general algorithms for computing the relations associated with the nodes of a rule/goal tree. First, let us review and formalize the kinds of relations we compute.

1. At each goal node there is an answer (or result) relation, whose components correspond to the arguments of the goal.

2. At each rule node there is a result (or rule) relation whose components correspond to arguments of the rule head.

3. At each goal node there is a binding relation, whose components correspond to those arguments of the goal that are bound. The same arguments are also bound in the head of every rule-node child of this goal node. We say that variables appearing in one of these bound arguments of a rule head are "bound by the head" of that rule.

4. At each rule node for a rule with \( k \) subgoals, there are supplementary relations \( S_0, \ldots, S_k \). The arguments of a supplementary relation correspond to (some of the) variables of its rule instance. In particular, \( S_i \) has arguments corresponding to those variables that are both bound and relevant after considering the first \( i \) subgoals. Recall that variables are bound either by appearing in a bound argument of the head or by appearing in one of the first \( i \) subgoals; variables are relevant if they appear either in the head or in the \( (i+1) \)st or a subsequent subgoal.

The reader should understand clearly that bindings are tuples, with one or more components. Roughly, a tuple \( \mu \) binding certain arguments or variables means that during the backtracking algorithm of Section 12.3, these arguments or variables are simultaneously given the values found in the components of \( \mu \). In the simple example of the previous section, we saw only one argument or variable bound at a time.

Each of the relations in (1) through (4) is computed from the values of one or more other relations. As we shall see in the next section, it requires a delicately chosen sequence of operations to make sure that each tuple belonging in a relation at some node of the rule/goal tree is placed there eventually. However, in this section, we shall simply study the matter of how tuples should, in principle, be placed in these relations. These calculations are summarized in the following paragraphs.

Sideways Information Passing

Let us consider a rule node with rule instance \( H \Rightarrow G_1 \& \cdots \& G_k \). For \( i = 1, \ldots, k \), we compute \( S_i \) from \( S_{i-1} \) and the relation for subgoal \( G_i \). That is, let the result relation for the child of this rule node corresponding to subgoal \( G_i \)

be \( R_i \), and let \( Q_i = \text{ATOV}(G_i, R_i) \). Then \( S_i \) is computed by joining \( S_{i-1} \bowtie Q_i \) and then projecting out any irrelevant variables.

Example 12.20: Consider the rule

\[
p(f(x), y) \leftarrow q_1(x, u, v) \land q_2(g(u, w), f(u)) \land q_3(w, x, y).
\]

Suppose we are given a relation \( S_0(x) \) that provides a set of values for variable \( X \). The remaining three supplementary relations are computed by

\[
S_1(x, u) = \pi_{xu}(S_0(x) \bowtie Q_1(x, u, v)).
\]

\[
S_2(x, w) = \pi_{xw}(S_1(x, u) \bowtie Q_2(u, w)).
\]

\[
S_3(x, y) = \pi_{xy}(S_2(x, w) \bowtie Q_3(w, x, y)).
\]

Where \( Q_i \) is the relation for the \( i \)th subgoal, translated from the point of view of arguments of \( q_i \) to the point of view of variables, as described above.

For example, \( X \) and \( U \) are attributes of \( S_1 \), because they are both bound and relevant. In particular, \( X \) is bound because it is bound by the head (it is an attribute of \( S_0 \)) as well as \( Q_1 \), while it is relevant because it is used later on, both in subgoal \( q_1 \) and in the head. \( U \) is bound in \( Q_1 \), and it appears subsequently in \( Q_2 \). \( V \) is not an attribute of \( S_1 \) because, although it is bound by \( Q_1 \), it appears neither in \( Q_2 \), \( Q_3 \), nor the head. □

An equivalent way to express the operation of sideways information passing is with a rule that combines the argument-to-variable conversion with the natural join. That is, let \( \text{sup}_{\phi}(X_1, \ldots, X_n) \) and \( \text{sup}_{\phi-1}(Y_1, \ldots, Y_m) \) be predicates corresponding to the two supplementary relations \( S_i \) and \( S_{i-1} \), and let \( \phi(t_1, \ldots, t_i) \) be the \( i \)th subgoal. Then we can write

\[
\text{sup}_{\phi}(X_1, \ldots, X_n) \bowtie \text{sup}_{\phi-1}(Y_1, \ldots, Y_m) \bowtie \phi(t_1, \ldots, t_i).
\]

If we let the relation \( R_i \) be the relation for the \( p \) subgoal in (12.9), then tuples for the body of (12.9) are formed by finding assignments \( \tau \) for the variables of the body that

1. Form tuples in \( S_{i-1} \), when restricted to \( Y_1, \ldots, Y_m \), and
2. Make \( \tau(\phi(t_1, \ldots, t_i)) \) a tuple of \( R_i \).

But condition (2) is the same as saying that \( \tau \), restricted to the variables appearing in \( \phi(t_1, \ldots, t_i) \), is a tuple of \( Q_i = \text{ATOV}(\phi(t_1, \ldots, t_i), R_i) \). Thus, the join of \( S_i \) and \( Q_i \) is accomplished by the logical AND of the subgoals in (12.9), and the projection onto the relevant variables is accomplished by the fact that irrelevant variables will not appear among the \( X \)'s.

For instance, the rules corresponding to the sideways information passing of Example 12.20 can be written
sup₂(₁, U) :- sup₂(₁ X, Y).

sup₂(₂, W) :- sup₂(₁ X, U) & q₂(g(U, W), f(U))

sup₂(₁, Y) :- sup₂(₂, W) & q₂(₂, X).

Passing Bindings from Rules to Goals

Suppose we have a rule node corresponding to rule instance \( H \vdash G₁ \& \cdots \& Gₖ \).

We can pass bindings from supplementary relation \( S_{i-1} \) to the goal-node child corresponding to \( G_i \) as follows.

1. First, we must determine which arguments of \( G_i \) are bound. An argument is bound if and only if all variables appearing therein are bound, that is, all variables of that argument are attributes of \( S_{i-1} \).

2. Let \( G'_i \) be the predicate of \( G_i \) with all and only the arguments of \( G_i \) that are bound according to \( S_{i-1} \).

3. Compute \( M = \text{vtoa}(G'_i, S_{i-1}) \).

Example 12.21: Let us consider rule (12.8) from Example 12.20. \( S_0 \) binds only variable \( X \), and that is enough to bind the first argument of the first subgoal, \( q₁(X, U, V) \). Thus, the binding relation for the first goal-node child of the rule node for (12.8) has one argument and is obtained by computing \( \text{vtoa}(q₁(X), S₁(X)) \). Note that \( q₁(X) \) is the first subgoal of (12.8) restricted to the bound argument.

Now consider the second subgoal, \( q₂(g(U, W), f(U)) \). \( S₁ \) has arguments \( X \) and \( U \), which are enough to bind the second argument, but not the first. Note that the first argument, \( g(U, W) \) is not bound, even though it is "partially bound," because one of its two variables is bound. Thus, the binding for the second goal-node child is

\[
\text{vtoa}(q₂(f(U)), S₁(X, U))
\]

That is, we take each tuple \( µ \) in \( S₁ \), let \( t = µ(U) \), and make \( f(t) \) a one-component tuple in the binding relation for the second goal-node child.

Finally, the third subgoal, \( q₃(W, X, Y) \), has its first two arguments bound by \( S₂(X, W) \). The binding relation for this subgoal is

\[
\text{vtoa}(q₃(W, X), S₂(X, W))
\]

is a copy of \( S₂ \) with the order of columns reversed.

The operation of passing bindings from supplementary relations to binding relations can also be expressed as a rule. That is, suppose \( S_{i-1}(X₁, \ldots, Xₖ) \) is the \((i-1)\)st supplementary relation, and the \(i\)th subgoal is \( p(t₁, \ldots, tₖ) \). Let arguments \( i₂, \ldots, iₖ \) of this subgoal be bound by \( S_{i-1} \). Then \( m \), the predicate corresponding to the binding relation for the \(i\)th subgoal, is

\[
m(t₁, \ldots, tₖ) :- \text{sup}_{i-1}(X₁, \ldots, Xₖ).
\]

We leave the correctness of this rule as an exercise.

Passing Bindings from Goals to Rules

Now, suppose we have a goal node for goal \( G \), with binding relation \( M \). Further, suppose this goal node has a rule-node child for rule instance \( H \vdash G₁ \& \cdots \& Gₖ \), where head \( H \) unifies with \( G \). The bindings for arguments of \( G \) also bind the same arguments of \( H \). In turn, these arguments bind whatever variables appear in them, and the bindings for the variables become tuples of the zeroth supplementary relation for this rule node. That is, \( S₀ = \text{ato}(H', M) \), where \( H' \) is \( H \) restricted to the arguments that are bound by \( M \).

Example 12.22: Suppose there is a goal node for predicate \( p \), whose binding relation \( M \) binds the first argument only. One child of this goal node corresponds to the rule (12.8), with head \( p(f(X), Y) \). The binding for the first argument of \( p \) binds variable \( X \), but not \( Y \), so \( S₀ \) at this node has scheme \( X \); its value is \( \text{ato}(f(X), M) \). That is, we look at each tuple of \( M \) and take its lone component. If that component is of the form \( f(t) \) for any term \( t \), then \( t \) becomes the lone component of a tuple of \( S₀ \). A tuple of \( M \) that is not of the form \( f(t) \) yields nothing for \( S₀ \).

This operation too can be expressed as a rule, the correctness of which is left as an exercise. Let \( p(s₁, \ldots, sₖ) \) be our goal node, and \( p(s₁, \ldots, sₖ) \) the head of the rule node. If the binding relation \( M \) binds arguments \( i₁, \ldots, iₖ \) and \( X₁, \ldots, Xₖ \) are the variables found in these arguments of the head, that is, among \( s₁, \ldots, sₖ \), then the rule for computing \( S₀ \) at this rule node is

\[
\text{sup}(X₁, \ldots, Xₖ) :- m(s₁, \ldots, sₖ).
\]

Passing Answers from Rules to Goals

When a rule node produces its answer relation, it contributes those answers to its goal-node parent. As we discussed above, a rule node for a rule with \( k \) subgoals computes supplementary relation \( Sₖ \), which binds all of the variables that are relevant after the \( k \)th subgoal, that is, exactly those variables that appear in the head. We need only to convert \( Sₖ \) from the point of view of variables to the arguments point of view, applied to the head \( H \). That is, the answer relation for this rule node is \( \text{vtoa}(H, Sₖ) \).

\[19\] Note that answer relations from goal nodes contribute to their rule-node parents by combination with the previous supplementary relation, to form the next supplementary relation, as described when we covered "sideways information passing." Thus, we need not say anything more about this, the second operation needed as we pass answers up to the root.
For example, rule (12.8) has a final supplementary relation \( S_1(X, Y) \), and a head \( p(f(X), Y) \). The result relation is thus

\[
\text{VTOA}\left(p(f(X), Y), S_1(X, Y)\right).
\]

That is, we take each tuple of \( S_1 \) and put function symbol \( f \) around the first component, leaving the second component intact.

We can express this operation as well in rule form

\[
H \vdash \text{sup}_1(X_1, \ldots, X_n).
\]

if \( X_1, \ldots, X_n \) are the variables of the \( k \)th supplementary relation, and \( H \) is the head of the rule. We can even dispense with the final supplementary set altogether, and go directly from \( S_{k-1} \) to \( H \) with

\[
H \vdash \text{sup}_{k-1}(Y_1, \ldots, Y_m) \land G_k.
\]

Here, \( Y_1, \ldots, Y_m \) are the variables of the \( (k-1) \)st supplementary relation, and \( G_k \) is the \( k \)th subgoal of the rule.

Finally, let us note that the result relation for a rule node is one of several relations that make up the result relation for its goal-node parent. In order to produce the result for the goal node, we must take the union of the results for each rule-node child of that goal node.

The reader may observe from the example of the previous section that it is not always clear when a complete result has been produced, and therefore, we may have difficulty producing result relations at either goal or rule nodes. In fact, without result relations at goal nodes, we cannot do sideways information passing, and therefore we cannot even develop binding sets for second and subsequent subgoals. In the next section, we shall see that all of the calculations that we implied were done on complete relations are, in fact, done incrementally in almost all cases. Thus, the calculations of this section really are performed, but not necessarily on complete relations at once.

**Improvements in the Passing of Bindings**

We saw in Example 12.21 that it is possible, when there are function symbols, for an argument to be "partially bound." Since we regard an argument as bound only when all variables appearing in that argument are bound, we may lose some binding information, as the following example shows.

**Example 12.23:** Suppose \( G_i \) is subgoal \( q_2(p(U, W), f(U)) \), and \( H \) is rule head \( q_2(p(A, B), C) \) for some rule \( r \), whose body is not important. Also assume that \( S_1 \), the prior supplementary relation at some rule node \( n \), binds \( W \), but not \( U \) (unlike Example 12.20, where we assumed \( U \) was bound). Then neither argument of \( G_i \) is bound by \( S_1 \), since \( U \) appears in both. Thus, \( G_i \) is \( q_2(A, B) \), that is, a zero-argument predicate. When we set \( M = \text{VTOA}(G_i, S_1) \),

we find none of the variables are bound in \( M \), so that relation consists of the empty tuple alone. Thus, in the rule-node child corresponding to \( r \), the zeroth supplementary relation will also have no attributes; that is, no variables are bound by the head of rule \( r \).

However, it seems intuitively that variable \( B \) is bound in the head of this instance of rule \( r \). That is, if we unify \( G_i \) with \( H, B \) and \( W \) are identified, and the values of \( W \), which we can obtain from \( S_1 \) at the rule node \( n \), become values of \( B \) at the rule node \( r \) that is a grandchild of \( n \). That is, we skip over goal nodes, jumping directly from a rule node to its descendants two levels below. By doing so, we can sometimes avoid losing bindings due to partially bound arguments that look completely unbound at the intermediate goal nodes.

We can formalize the process used in Example 12.23. Suppose we have a subgoal \( G_i \) some rule, and \( H \) is a rule head that unifies with \( G_i \). We can pass bindings from the supplementary relation \( S_{i-1} \) in the rule node \( n \) where \( G_i \) is located, to \( S_0 \) for the grandchild of \( n \) corresponding to the rule \( r \) with head \( H \), as follows.

1. Assume that \( r \) is rewritten so \( H \) and \( G \) share no variables. Unify \( G \) with \( H \), and let the MGUB be \( r \).
2. Identify bound variables of both \( H \) and \( G \) as follows.
   a) A variable of \( G \) bound by \( S_{i-1} \) is bound.
   b) For any variable \( X \), if \( X \) appears in \( \tau(Y) \) for some bound variable \( Y \), then \( X \) is also bound.
   c) For any variable \( X \), if all variables of \( \tau(X) \) are bound, then \( X \) is bound.
   Rules (a) to (c) are applied repeatedly, until no more bound variables can be found.
3. Construct \( S \), the zeroth supplementary relation for rule instance \( \tau(r) \). Its attributes are all the variables of \( H \) that are found in step (2) to be bound. For each tuple \( \nu \) in \( S_{i-1} \) at rule node \( n \), we construct a tuple \( \nu \) of \( S \), by mimicking the inferences of (2b) and (2c) above. That is,
   a) If \( X \) appears in \( \tau(Y) \) for some bound variable \( Y \) (whose value \( t \) we may assume has already been found), then strip away symbols, if necessary, from \( t \) to obtain the value of \( X \); the operation is essentially \( \text{ATOM} \). If no value of \( X \) is thus defined, and \( X \) is a variable of \( H \), then \( \nu \) does not exist. If \( X \) is a variable of \( G_i \), it is possible that \( \nu \) exists, but it may not derive any of its components from \( X \).
   c) If \( \tau(X) \) consists only of variables whose values have been determined, construct the value for \( X \) by substitution; the operation is essentially \( \text{VTOA} \).
If we are successful in constructing from μ a value for every component of ν (i.e., for every variable that is an attribute of S), then we add ν to S. If any component of ν is undefined, then μ makes no contribution to S.

Example 12.24: Let us consider a subtle example based on Example 12.13. Let the subgoal C be
\[ p(f(X, g(X)), g(g(Y))) \]
and let the head H be
\[ p(f(g(U), V), g(V)) \]
Recall from Example 12.13 that the MGU is
\[ τ(U) = U \]
\[ τ(X) = g(U) \]
\[ τ(Y) = g(U) \]
Suppose that Y is an attribute of \( S_{i-1} \), but X is not. Thus, by rule (2a), Y is bound. Since U appears in \( τ(Y) \), we conclude by (2b) that U is bound. It then follows by (2c) that V and X are also bound.

Hence, both variables appearing in the head are bound, and the scheme for S is UV. We construct S from \( S_{i-1} \), as follows. Let \( μ \) be any tuple in \( S_{i-1} \), and let \( t = μ[Y] \). If t is of the form \( g(s) \), then s is the value of U, and \( g(g(s)) \), or \( g(t) \), is the value of V. Thus, \( (s, g(t)) \) is a tuple of S. If t is not of the form \( g(s) \), then \( μ \) does not yield a tuple for S.

Despite the fact that skipping from rule nodes to their grandchild nodes, when passing bindings down the rule/goal tree, can sometimes produce more bound variables, we shall use the technique first described, of going through the intermediate goal nodes. One reason is that for datalog, it is easy to see that arguments cannot be partially bound unless they are completely bound, because arguments can only be constants or single variables. Even in nondatalog cases, there is no guarantee that there will be partially-bound arguments, or that anything can be gained by treating them specially. The most compelling reason will only appear when we consider the magic-sets rule-rewriting technique in Chapter 13. There, we shall see that there is a significant savings in the number of rules needed by going through the intermediate goals. However, should there be the need to do so, either the rule/goal tree algorithm to be described formally in the next section, or the magic-sets technique can be modified to do more precise passing of bindings, as we have described here.

Improving Sideways Information Passing

Another modification to the basic algorithm that sometimes can improve performance is to revise the way bindings are passed sideways. That is, we assumed bindings were passed through supplementary relations, one for each position in the rule body, and each supplementary relation had as attributes exactly the bound and relevant variables at that position. However, we can sometimes save work, as the next example shows, by using another pattern of information passing.\(^{20}\)

Example 12.25: Consider the rule
\[ p(X, Y) \leftarrow q_1(X, Z) \land q_2(X, W) \land q_3(Z, Y) \land q_4(W, Y). \]  
(12.10)

Suppose that at some rule node for (12.10), the goal-node parent binds the first argument of p, but not the second. Then \( S_5 \) has attribute X, \( S_1 \) has attributes X and Z, and most importantly, \( S_2 \) has attributes X, Z, and W. For convenience, let \( S_2 \) have n different values, and suppose the relations for \( q_1 \) and \( q_2 \) each have m values of the second argument associated with each value of the first argument. Thus, \( S_2 \) has \( mn^2 \) tuples.

However, suppose we compute the result relation for (12.10) by taking
\[ π_{XY}(S_5(X) \bowtie Q_1(X, Z) \bowtie Q_3(Z, Y)) \]
and intersecting it with
\[ π_{XY}(S_2(X) \bowtie Q_2(X, W) \bowtie Q_4(W, Y)) \]
where \( Q_i \) is the relation for \( q_i \), 1 ≤ i ≤ 4. The first step in each case, joining \( S_5 \bowtie Q_1 \) and \( S_2 \bowtie Q_2 \), produces a relation of size \( mn \), by our assumption. If W and Z have large domain sizes, we may suppose that joining with \( Q_3 \) and \( Q_4 \), respectively, does not increase the size of the relations involved. If that is the case, then our alternative method of evaluating the result relation never produces any relation as large as \( S_2 \). The alternative method is therefore more efficient by about a factor of \( m \). The intuitive reason for this improvement is that in (12.10), the roles played by variables W and Z are independent, yet their values are paired in \( S_2 \), effectively taking a useless Cartesian product.

We should be aware of the opportunity sometimes afforded by varying the sideways information passing strategy. However, we shall, in the future, assume that only the "standard" method of passing information sideways is used; that is, we compute each of the supplementary relations in turn.

12.7 THE RULE/GOAL TREE EVALUATION ALGORITHM

We are now ready to put the informal algorithm of Section 12.5 together with the techniques of Section 12.6, to form the complete rule/goal tree expansion and evaluation algorithm. We shall begin with the basic algorithm, which does

\(^{20}\) We do not take advantage of the option to reorder subgoals. Rather, in this example, we still pass bindings from left to right only, but we do so in a manner other than the construction of supplementary relations.
a preorder traversal, that is, a depth-first search, of the rule/goal tree, assuming all branches will terminate in a finite time; termination might occur because the predicates encountered are not recursive, or because we eventually find empty sets of bindings, as we did in Example 12.17. We then discuss how to modify the algorithm so it is queue-based, that is, it performs breadth-first search of the rule/goal tree, as was implicit in Section 12.5.

**Initial Bindings**

For uniformity, we shall treat constants in the query somewhat differently from the way we dealt with constant \( j \) in Examples 12.16 and 12.17. Here, we shall assume that all arguments of the query are variables, but we may supply a relation to the root of the rule/goal tree, providing bindings for certain arguments of the root goal.

For example, instead of starting the rule/goal tree of Figure 12.15 with the root goal \( \text{anc}(j, W) \), we shall use the goal \( \text{anc}(Z_0, W) \) as the query. We then provide that goal the unary binding relation \( \{j\} \), whose attribute corresponds to the first argument of \( \text{anc} \). Note how the choice of \( Z_0 \) in place of \( j \) fits in with the pattern of variables used in Figure 12.15.

**The Basic Algorithm**

We shall now present the simple form of the rule/goal tree expansion and evaluation algorithm for which we have been preparing. As many of the details were given already, we shall allude to several algorithms and techniques from the previous sections.

**Algorithm 12.6: Rule/Goal Tree Expansion and Evaluation.**

**INPUT:** A collection of safe rules, a database, a goal \( G_0 \) that is the query, and a relation \( M_0 \) that binds zero or more of the arguments of \( G_0 \).

**OUTPUT:** The relation of tuples that satisfy the query \( G_0 \) according to the given database. However, if the rules and database do not guarantee convergence (for example, because the rules are recursive), then this "algorithm" may in fact run forever, and will not produce an answer.

**METHOD:** The heart of the algorithm is a pair of recursive routines

1. expandGoal(\( M, G, R \)) takes goal \( G \), binding relation \( M \) for zero or more of \( G \)'s arguments, and returns result relation \( R \), over all the arguments of \( G \). Result \( R \) is the set of tuples that can be inferred from the given rules and database, match goal \( G \), and agree with some one tuple of \( M \) on all arguments in the scheme of \( M \).

2. expandRule(\( S_0, r, R \)) takes rule instance \( r \) and zeroth supplementary set \( S_0 \), which binds zero or more variables of \( r \), and produces result relation \( R \), whose attributes correspond to the arguments of the head of \( r \). The value of \( R \) is the set of tuples that match the head of \( r \), are inferred from the database and rules, and agree with some one tuple of \( S_0 \) in all variables for which \( S_0 \) has an attribute.

These routines are exhibited in Figure 12.17. Some additional details are as follows.

1. For procedure expandGoal of Figure 12.17(a):
   a) Line (1) lets us terminate searches that are guaranteed not to return any tuples. This step is essential if the search of some recursive rule/goal trees, as in the example of Section 12.5, are to converge.
   b) Line (6) requires a unification, as described in Section 12.4.
   c) Line (7) passes bindings from goals to rules, as in Section 12.6.

2. For the procedure expandRule of Figure 12.17(b):
   a) Line (3) passes bindings from rules to goals, also as described in Section 12.6.
   b) Line (5) translates a relation that represents an answer from the argument viewpoint to the viewpoint of variables.
   c) Line (6) performs sideways information passing, as described in Section 12.6.
   d) In line (7), the set \( S_0 \) will be a relation over exactly those variables that appear in the head \( H \), as discussed in Section 12.6.

The algorithm itself consists of a call to expandGoal(\( M_0, G_0, R_0 \)), followed by the printing of the answer, the relation returned as \( R_0 \). □

**Example 12.26:** Let us redo the example of Section 12.5 according to Algorithm 12.6. We make the modification that the query goal is now \( \text{anc}(Z_0, W) \), with the binding \( \{j\} \) for \( Z_0 \). Figure 12.15, with this modification, is repeated as Figure 12.18.

To begin, we call expandGoal(\( M_0, \text{anc}(Z_0, W), R_0 \)), where \( M_0 = \{j\}; R_0 \) is the name of the relation that we shall compute. This call results in two more calls, expandRule(\( S_0^1, r_1, R^1 \)) and expandRule(\( S_0^2, r_2, R^2 \)), where \( S_0^1 \) and \( S_0^2 \) are each relations with scheme \( Z_0 \) and value \( \{j\} \), constructed according to line (7) of expandGoal.21 Rule instances \( r_1 \) and \( r_2 \) correspond to the children of the root in Figure 12.15 (with \( Z_0 \) in place of \( j \)), and are

\[
\begin{align*}
r_1^1 & : \text{anc}(Z_0, W) := \text{par}(Z_0, W), \\
r_2^1 & : \text{anc}(Z_0, Z_0) := \text{par}(Z_0, Z_0) \land \text{anc}(Z_0, Z_0)
\end{align*}
\]

Now let us follow the first of these calls to expandRule. At line (3) we get the binding relation \( M^0 = \{j\} \) with scheme PAR, the first argument of \( \text{par} \), for the subgoal \( \text{par}(Z_0, W) \). That results in a call to

21 We shall assign parameter names for each call by using a new number for each call and superscripting all of its variables by this number.
procedure expandGoal(M, G, R);
begin
  (1) if M = ∅ then begin R := ∅; return end;
  (2) if G is a goal with an EDB predicate then begin
      R := M ⊕ P; return end /* P is the relation for the predicate of G. Note this operation is a selection on the database relation P */
      /* here, G has an IDB predicate */
      R := ∅ /* we accumulate relations for each of G's rule heads in R */
  (3) for each rule r whose head H unifies with G do begin
    (4) let τ be the NGU of H and G;
    (5) compute S₀ = ATOV(H', M); /* H' is τ(H) restricted to arguments that are attributes of M */
    (6) expandRule(S₀, τ(r), R);
    (7) R := R ∪ R;
  end
end

(a) Procedure to expand goal nodes.

procedure expandRule(S₀, r, R);
begin
  (1) let r = H := G₁ & … & Gₖ;
  (2) for i := 1 to k do begin /* compute bindings for ith subgoal */
    Mᵢ := VTOA(Gᵢ, S₀₋₁); /* Gᵢ is Gᵢ restricted to arguments whose variables are attributes of S₀₋₁ */
    expandGoal(Mᵢ, Gᵢ, R);
  end /* convert Rᵢ to the viewpoint of variables and use it to compute the next supplementary relation */
  (3) Qᵢ := ATOV(Gᵢ, R);
  (4) Sᵢ := πT(S₀₋₁ β Qᵢ); /* T is the set of variables that appear in the scheme of S₀₋₁ or Qᵢ, and also appear in one of H, G₁,…,Gₖ */
end;

R := VTOA(H, S₀);

(b) Procedure to expand rule nodes.

Figure 12.17 Recursive procedures of Algorithm 12.6.

12.7 The Rule/Goal Tree Evaluation Algorithm

expandGoal(M², par(Z₀, W), R²)

which at line (3) of expandGoal returns with R² = {ff, jh}, according to the relation for par given in Figure 12.14 of Section 12.5.

Figure 12.18 Rule/goal tree for goal par(Z₀, W).

The first call to expandRule is complete, so we may consider the second call, the one for rule instance r₂. Here, for the first subgoal, par(Z₀, Z₁), we again compute at line (3) binding set M₄ = {j}, and proceed as for r₁ with a call

expandGoal(M₄, par(Z₀, Z₁), R₄)

that returns R₄ = {ff, jh}. At line (5) of expandRule, we convert R₄ to Q₄, a relation with scheme Z₀Z₁ and value {ff, jh}. Then at line (6) we compute the next supplementary relation

S²(Z₀, Z₁) = πZ₀,Z₁(S²(Z₀) β Q₄(Z₀, Z₁)) = {ff, jh}

Recall that S² has scheme Z₀ and value {j} in the above computation. S² has both variables Z₀ and Z₁ in its scheme, because Z₀ is found in the head, and Z₁ is found in a subsequent subgoal of r₃. Thus, the projection has no effect in this case.
We now return to the loop of line (2) in the execution of expandRule for rule instance \( r_z^2 \), and we consider the second (and last) subgoal, \( \text{anc}(Z_1, W) \). At line (3) we compute binding relation \( M^2 = (f, h) \), whose lone attribute corresponds to the first argument of \( \text{anc} \). We are lead at step (4) to a call 

\[ \text{expandGoal}(M^2, \text{anc}(Z_1, W), R^2) \]

If we formalize the discussion of Example 12.17 (Section 12.3), in terms of Algorithm 12.6, we discover that eventually, this call terminates and returns 

\[ R^2 = \{ fc, fe, hd, fn, jh, ha, bb \} \]

The key observation is that after several levels of recursive calls, we reach the "top" of the genealogy in Figure 12.14, resulting in a call 

\[ \text{expandGoal}(\emptyset, \text{anc}(Z_4, W), R) \]

Line (1) of \( \text{expandGoal} \) intercepts this situation and terminates the recursion, correctly returning \( R = \emptyset \).

Now, let us revisit the call of expandRule for \( r_z^2 \). At line (6), we compute \( Q^5 \); it is the same relation as \( R^3 \), but its scheme is \( Z_1 W \). At line (6) we compute 

\[ S^5_1(Z_0, W) = \text{anc}(W, S^5_1(Z_2, Z_4)) \]

Recall \( S^5_1 = \{ ff, jh \} \), and its scheme is \( Z_0 Z_1 \), while \( Q^5 \) is the relation given above as \( R^3 \), but with scheme \( Z_1 W \).

Finally, the loop of lines (2) through (6) terminates, and we compute the returned relation \( R^2 \); it is the same as \( S^5_1 \) above, but its components correspond to the two arguments of predicate \( \text{anc} \).

Now, we have completed both calls made by the original call, 

\[ \text{expandGoal}(\{ j, jh \}, \text{anc}(Z_0, W), R_0) \]

and at line (9) we have been accumulating the relations returned by the two calls to expandRule. Thus, we may set 

\[ R_0 = \{ jf, jh, jc, je, jn, jh, jd \} \]

which is the answer to the query. □

A Queue-Based Version of Rule/Goal Tree Expansion

Algorithm 12.6 terminates only in those situations where the backtracking, top-down scheme of Section 12.3 terminates. In a sense, we cannot do better without reordering the subgoals of some rules, because Algorithm 12.6 terminates whenever the rule/goal tree can be made finite by excising nodes to which an empty set of bindings is passed. In another sense, we can improve on Algorithm 12.6 if we adopt the breadth-first, or queue-based order of node exploration, suggested in Section 12.5. If we do, and if the answer is finite, we shall converge eventually to the correct answer, even though it is difficult to tell we have converged.\(^{22}\)

We shall sketch the necessary modifications to Algorithm 12.6, so that the top-down tree expansion algorithm will converge to the query's answer, just as the bottom-up method does. The reader should remember from Section 12.2 that, when function symbols are involved, it is possible that the answer is infinite, and therefore, convergence does not occur at any finite time. Nevertheless, we can show that the limits of what the bottom-up and the queue-based top-down algorithms produce are the same. That is, a fact inferred by one method is eventually inferred by the other.

Algorithm 12.6, with its mutually recursive procedures expandGoal and expandRule, is easily seen to visit nodes of the rule/goal tree in a depth-first order; that is, it makes a preorder traversal of the fully expanded tree. Instead of stacking activations of the two procedures, as one normally does, we could put the calls on a queue. So doing has the effect of executing the calls corresponding to the nodes of the rule/goal tree level-by-level, that is, in a breadth-first search of the tree.

If we visit nodes in this order, we are not prepared to provide bindings in some cases. For example, suppose we have a rule instance \( r \) with subgoals \( G_1 \) and \( G_2 \). When we visit the rule node for \( r \), we put the goal nodes for \( G_1 \) and \( G_2 \) at the end of the queue. The binding relation for \( G_1 \) will be available now, and provided \( G_0 \), the zeroth supplementary relation for \( r \), is available. However, the binding relation for \( G_2 \) is not available, and will not be available until we have completely explored the tree rooted at \( G_1 \), which may never happen, since that tree may be infinite. Thus, we shall expand \( G_2 \) when it reaches the head of the queue, with the binding relation that is obtained from whatever tuples we know are in \( S_1 \), perhaps none. At intervals, new bindings for \( G_2 \) may be discovered and passed down the tree as far as the tree has grown.

Propagation of New Information

As we expand the tree level-by-level, we sometimes discover new facts. All such facts have an origin that is a goal node with an EDB predicate, for which we look up the database and produce the answer relation for that goal. The discovery of tuples in the answer relation for one node can have a rippling effect, as follows.

1. Answer tuples for a goal node \( G_i \) can join with a supplementary set \( S_{i-1} \) at the parent rule node to produce new tuples in the next supplementary relation, \( S_i \), at that rule node.

\(^{22}\) In comparison, bottom-up logic evaluation, as discussed in Section 12.2, makes it easy for us to tell when convergence has occurred, we simply check that at one round, nothing is added to any predicate.
2. Additional tuples in a supplementary relation $S_i$ can
   a) Produce new binding tuples for the goal node of subgoal $G_{i+1}$.
   b) Produce new answer tuples for the rule node, if $S_i$ is the last supplementary relation; that is, there are only $i$ subgoals.
   c) Produce new tuples for the next supplementary relation $S_{i+1}$, by combining with tuples already found for subgoal $G_{i+1}$.

3. New binding tuples for a goal node can produce
   a) New tuples for the zeroth supplementary relations of its rule-node children.
   b) New answer tuples, if the goal has an EDB predicate.

4. New answers for a rule node can produce new answers for its goal-node parent.

Each addition of tuples to the answer relation belonging to some goal node in the portion of the tree already constructed can thus spawn events in which tuples are added to one or more other relations, usually at different nodes. It turns out that this spawning of new tuples cannot go on indefinitely if the rule/goal tree remains fixed; the next lemma proves this contention.

**Lemma 12.1:** If we have a finite portion of the rule/goal tree constructed, and some new tuples for the answer relation of some goal node are discovered, the propagation rules described above result in only a finite number of operations in which new tuples are inserted into some answer relation, supplementary relation, or binding relation.

**Proof:** New answer tuples can propagate upwards, resulting in new answers for its rule and goal ancestors. At each rule ancestor, the supplementary relations $S_1, S_2, \ldots$ can receive new tuples if $G_i$ is the subgoal of the rule receiving new answers. In fact, it is essential that supplementary relations be augmented all of the way to the right end of the rule, if the rule node is to obtain new answers.

Figure 12.19 suggests this process.

However, the augmented supplementary relations can spawn new bindings down the tree. Bindings are passed downward further, from binding relations for goals, to supplementary relations for rules. We must observe that any such bindings are at nodes to the right of the original goal node ($G$ in Figure 12.19). It is possible for these additional tuples in binding relations to "turn around" and produce new answer tuples in two situations.

1. Bindings are passed to an EDB goal, and answers are produced by database lookup.
2. Bindings that are passed to the zeroth supplementary relation for a rule propagate all of the way to the end of the rule using tuples already found as answers for the various subgoals of the rule. Thus, we produce new answers for that rule node and for the goal node above it.

Figure 12.19 Passing answers up and to the right.

It may not be obvious that (2) can ever occur; we give an example after this proof is completed.

As mentioned, any goal node receiving new answer tuples must lie to the right of the goal node whose additional answer tuples started this propagation. Let us regard such goal nodes as new sources of answer tuples, unrelated to the propagation of the original answer tuples. Thus, each source's propagation must die out after a finite number of steps, perhaps to be replaced by new "sources" to the right of the old source. Since the rule/goal tree is fixed, new sources of answer tuples cannot be created forever. Thus, there is a limit on the number of changes induced by an initial discovery of new answers, including all propagation from secondary sources.

**Example 12.27:** Figure 12.20 shows a hypothetical rule node and its current supplementary relations. We also show the goal nodes for the subgoals of $(X, Z)$ and $t(Y)$, along with their current binding relations (indicated by the downward arrow) and their current answer relations, going upward. Suppose that we propagate the additional binding tuple $ab$ to $S_0$. The existence of $ae$ in the answer relation for $q$ tells us that $abc$ must be added to $S_1$. Then, the existence of tuple $b$ in the answer relation for $t$ gives us $abc$ in $S_2$, and this tuple is also an answer tuple for the rule node.

**Example 12.28:** Now let us have an example of the entire process of generating the rule/goal tree and propagating new tuples. We shall reconsider the rule/goal tree discussed in Example 12.26 and shown in Figure 12.18. We begin as in Example 12.26, with the call to `expandGoal(M_0, ans(Z_0, W), R_0)`, and $M_0 = \{ 1 \}$. That call places on the queue two more expansion actions.
\[
\begin{array}{c|c|c}
X & Y \\
--- & --- \\
\hline
a & d \\
\hline
c & b \\
\hline
\end{array}
\quad
\begin{array}{c|c|c}
X & Y & Z \\
--- & --- & --- \\
\hline
a & d & e \\
\hline
c & b & f \\
\hline
\end{array}
\quad
\begin{array}{c|c|c}
X & Y \\
--- & --- \\
\hline
a & e \\
\hline
c & b \\
\hline
\end{array}
\]

\[S_0 \quad S_1 \quad S_2\]

\[p(X, Y, Z) \quad q(X, Z) \quad t(Y)\]

\[\begin{array}{c|c|c|c|c|c}
1 & q & \uparrow & 1 & t & \uparrow \\
\hline
Q_1 & a & X & Z & T_1 & Y \\
\hline
c & a & e & b & c & f \quad d \\
\end{array}\]

Figure 12.20 Example where bindings yield answers immediately.

\text{expandRule}(S_1, r_1^1, R^1) \text{ and expandRule}(S_2, r_2^1, R^2), \text{ where } S_1 = S_2 = \{f\}.

These are executed in turn, and they spawn three more actions for the queue, corresponding to the third level of Figure 12.15,

\text{expandGoal}(M^3, \text{par}(Z_0, W), R^3) \quad \text{expandGoal}(M^4, \text{par}(Z_0, Z_1), R^4) \quad \text{expandGoal}(M^5, \text{anc}(Z_1, W), R^5)

Here, \(M^3 = M^4 = \{j\}\), but unlike Example 12.26, the initial value of \(M^5\) is \(\emptyset\). The reason is that we have not yet had time to process the goal \(\text{par}(Z_0, Z_1)\), and thereby develop values for \(Z_1\) to pass sideways.

When we process the goal node \(\text{par}(Z_0, W)\), we generate the answer tuples \((jf, jh)\) for \(R^3\), and these tuples must be allowed to join with the zeroth supplementary set for the rule node above, that is \(S_0\), so tuples may be added to the next supplementary relation, \(S_1\). We add \(jf\) and \(jh\) to \(S_1\). Since this supplementary relation is the last for its rule node, we use it to generate answers for the rule node, which puts \(jf\) and \(jh\) in \(R^1\). That, in turn, puts these tuples in the answer relation \(M_0\).

Now, consider the next goal node on the queue, \(\text{par}(Z_0, Z_1)\). We similarly generate answer tuples \((jf, jh)\), and these join with the zeroth supplementary relation, \(S_0\), at the rule node above, to yield the same two tuples for \(S_1\). This, supplementary relation, not being the last for its node, does not propagate answers upward, but it does propagate bindings downward to goal node \(\text{anc}(Z_1, W)\). Thus, \(M^5\) is set equal to \(\{f, h\}\). The balance of the simulation is omitted; the ideas are similar. □

In summary, we have described a variant of Algorithm 12.6, which we shall call queue-based rule/goal tree expansion (QRGT). In QRGT, expansion of nodes occurs level-by-level, and at each stage, propagation of new tuples for binding relations, supplementary relations, and answer relations is possible. These relations thus converge to their correct values; in some cases, for example, if the rules are datalog rules, then convergence will occur after a finite time. Otherwise, the values of the relations approach, but never reach, their correct values, in the sense that every tuple belonging to a relation will eventually be added to the relation, although there is no finite time at which all necessary tuples are in all of the relations.

Equivalence of Bottom-Up and Top-Down Computations

When we compare the top-down QRGT algorithm with bottom-up computation, that is, with naive or semi-naive evaluation, we find an important similarity: tuples are eventually generated by QRGT if and only if they match the root goal and they are generated bottom up. Since the bottom-up computation is likely to generate as well many tuples that neither match the root goal, nor match any other goal needed to infer tuples for the root goal, it appears that QRGT provides considerably more "focus" than bottom-up. That is, top-down is better at avoiding the inference of irrelevant tuples. There is a compensating factor that convergence is easier to detect when computing bottom-up. However, in the next chapter we shall see that the advantages of both methods can be combined into one general technique. For the moment, let us sketch the proof that top-down and bottom-up query evaluations yield the same answer.

Theorem 12.2: The QRGT variant of Algorithm 12.6 eventually produces a tuple (ground atom) \(\mu = \mu(t_1, \ldots, t_k)\) in response to query goal \(G\) with binding relation \(M_0\), if and only if \(\mu\)

1. Is inferred bottom-up from the same logical rules.
2. Agrees with some tuple of \(M_0\) on the attributes of \(M_0\), and
3. Term-matches \(G\).

Proof: The "only if" portion is easy. An induction on the number of steps of QRGT tells us that only tuples matching their goals and binding relations are ever generated. A similar induction shows that all tuples produced for answers are correctly inferred from the rules and would therefore be generated bottom-up.

Now consider the "if" part, that is, if \(\mu\) is generated bottom up from the database and rules, matches some goal \(G\), and agrees with some tuple of the binding relation for \(G\), then \(\mu\) is eventually added to the answer relation for \(G\), during QRGT. The proof is an induction on the number of steps required to infer \(\mu\) bottom-up. The basis, zero steps, only occurs if \(G\)’s predicate is an
EDB predicate. Then $\mu$ is placed in the answer relation by lines (2) and (3) of expandGoal, or by a propagation action of type (3b) if the goal node for $G$ was “expanded” before the binding tuple that matches $\mu$ was propagated to $G$.

For the induction, suppose $\mu$ is inferred bottom-up by a use of rule

$$r: H \geq G_1 \& \cdots \& G_m$$

Then $G$, $H$, and $\mu$ all have the same predicate symbol. Moreover, the head $H$ becomes $\mu$ when some substitution $\rho$ is applied to this rule; that is, $\rho(H) = \mu$. Since we are given that $G$ term-matches $\mu$, it must also be the case that $\mu = \phi(G)$ for some substitution $\phi$. Now, let $\tau$ be the MGU of $G$ and $H$. By Theorem 12.1, since $\phi$ and $\rho$, applied to $G$ and $H$, respectively, make $G$ and $H$ identical, these two substitutions must be obtainable by substitutions on $\tau(G)$ [which equals $\tau(H)$]. That is, $\rho = \rho'\tau$ and $\phi = \phi'\tau$.

$$G$$

$$\tau(G_1) \cdots \tau(G_m)$$

Figure 12.21 Part of a rule/goal tree.

Let $\nu_i, 1 \leq i \leq m$, be the tuples that are used to infer $\mu$ bottom-up using rule $r$; that is, $\nu_i = \rho(G_i)$. In the rule/goal tree, the goal node for $G$ has a child for the rule instance $\tau(r)$, which has children for each of the goals $\tau(G_1), \ldots, \tau(G_m)$, as suggested in Figure 12.21. Since $\rho = \rho'\tau$, each $\tau_i$ term-matches $\tau(G_i)$; that is, $\nu_i = \rho'\tau(G_i)$.

We know that each $\nu_i$ is inferred bottom-up using fewer steps than are used for $\mu$; thus the inductive hypothesis applies to the $\nu_i$'s. Now it is simple to show, for $i = 1, 2, \ldots, m$ in turn, that

1. A tuple matching $\nu_i$ is eventually placed by the top-down algorithm in the binding relation for goal node $\tau(G_i)$.
2. Hence, by the inductive hypothesis, each $\nu_i$ is eventually placed in the answer relation for $\tau(G_i)$.

Thus, we can infer that $\mu$ is placed in the answer relation for $\tau(r)$, and therefore in the answer relation for $G$. When we let $G$ be the root goal, we have the theorem. $\square$

---

24 Recall that we may write tuples as ground atoms with the appropriate predicate symbol.
An adornment for a rule indicates which variables are bound at a point and which are free. The notation we use for a rule adornment is a superscript of the form \([X_1, \ldots, X_m | Y_1, \ldots, Y_n]\), where the \(X\)'s are bound and the \(Y\)'s are free. The adornment for rule \(r_i\), prior to consideration of any subgoals, is attached as a superscript to \(r_0\), while the adornment that applies after consideration of the \(i\)th subgoal is a superscript of \(r_i\).25

**Example 12.29:** Consider the recursive ancestor rule, which we have called \(r_2\):

\[r_2: \text{Anc}(X,Y) \iff \text{par}(X,Z) \& \text{Anc}(Z,Y)\]

Suppose that \(\text{Anc}\) is called with a binding for the first argument. Then in the rule before consideration of any subgoals, only \(X\) is bound, and we represent this fact by the adorned rule \(r_{2,0}^{[X_1]}\). The first subgoal, \(\text{par}(X,Z)\), provides a binding for \(Z\), so the situation after consideration of the first subgoal is represented by \(r_{2,1}^{[X_1]}\). There is no need to represent the situation after the last subgoal is considered; all the variables will be bound by then, assuming the rule is safe. □

**Constructing a Rule/Goal Graph**

We can represent the patterns of binding that occur in a rule/goal tree by a finite structure called a rule/goal graph. Suppose we are given a set of Horn-clause rules and a query goal. If \(p\) is the predicate of the query, and \(\alpha\) is the adornment that has \(b\) whenever the query specifies a value for the corresponding argument and has \(f\) whenever no value is specified, then we begin construction of the rule/goal graph for this query with the node \(p^0\). We then consider each node in the rule/goal graph and expand it according to the following rules. As we expand, we add goal nodes, which are adorned predicates, and rule nodes which are nodes representing a rule and some number of the subgoals for that rule. Using the notation described above, we have \(r_0\), with an adornment, to represent rule \(r\) before considering any subgoals, and \(r_i\), with an adornment, to represent rule \(r\) after considering its first \(i\) subgoals.

1. A goal node with an EDB predicate has no successors.
2. A goal node that is an IDB predicate \(p\) with an adornment \(\alpha\) has successors corresponding to all of the rules with head predicate \(p\). If \(r\) is such a rule, then \(p^\alpha\) has successor \(r_0^{[X_1, \ldots, X_m | Y_1, \ldots, Y_n]}\).

---

25 Often, we call the rules \(r_2, \ldots, r_1\); in that case, we attach the adornments for rule \(i\) to \(r_{i,0}, j = 0, 1, \ldots, i - 1\).
we have $sg^{bf}$ again. Thus, we need add no more nodes, and the construction of the rule/goal graph is complete.

Note, however, that we could have merged the two occurrences of $\text{person}^{b}$ and the three occurrences of $\text{par}^{bf}$. We could also merge the two nodes for $r_{2,3}$. We shall not make such mergers here or elsewhere. Rather, we prefer to show each occurrence of an EDB subgoal separately, and we prefer to expand separately each rule node.26

It turns out that the order of subgoals in $r_{2}$ is not likely to be the best we could have chosen. The reason is that, when starting with $sg^{bf}$, we are forced to evaluate $\text{par}^{bf}$, that is, to include the entire par relation in a join to evaluate $r_{2}$. Intuitively, we would like to avoid evaluation of subgoals that do not have at least one bound argument whenever possible, since typically, binding even one argument will cut down significantly on the number of tuples. It happens that for the same-generation example, we can avoid examining the full par relation if we move the $sg$ subgoal into the middle of the body; that is, we rewrite the rules as

$$
\begin{align*}
 r_{1} : & \quad sg(X, X) \Leftarrow \text{person}(X). \\
 r_{2} : & \quad sg(X, X) \Leftarrow \text{par}(X, Xp) \land sg(Xp, Yp) \land \text{par}(Y, Yp).
\end{align*}
$$

The rule/goal graph for these rules and query $sg^{bf}$ is shown in Figure 12.23.

---

12.9 MAKING BINDING PATTERNS UNIQUE

Consideration of any subgoals, has variable $X$ bound and no free variables."

The other child of the root,

$$
\text{par}^{bf}
$$

has child $\text{par}^{bf}$, representing the first subgoal of $r_{2}$. The adornment is $bf$, because the binding on $X$ provides a binding for the first argument of $\text{par}(X, Xp)$, but not for the second. It also has child

$$
\text{par}^{bf}
$$

indicating that, after the first subgoal, $Xp$ as well as $X$, is bound.

The latter node has child $\text{par}^{bf}$; the adornment indicates that bindings for $X$ and $Xp$ do not provide bindings for either argument of the second subgoal, $\text{par}(Y, Yp)$. It also has child

$$
\text{par}^{bf}
$$

which indicates that after processing the first two subgoals of $r_{2}$, we have bindings for all of the variables. Thus, this node has child $sg^{bf}$, which represents the third and last subgoal of $r_{2}$; by the time we reach this subgoal, both its arguments are bound.

We now have a second node for predicate $sg$, this time with adornment $bb$. It is expanded in much the same way that the first node, $sg^{bf}$ was. The adornments of the rule node are different, but by the time we get to the third subgoal,

---

26 The motivation for doing so is additional clarity. Repeating EDB goals tends to avoid long arcs, and expanding each rule occurrence independently helps us visualize the sideways information passing without following long arcs.
each IDB predicate has a unique binding pattern associated. The technique is simple.

1. For each adornment \( \alpha \) of an IDB predicate \( p \), such that \( p^\alpha \) appears in the rule/goal graph, create a new predicate \( p.\alpha \).

2. For each of \( p.\alpha \) rules, say \( r \), make a copy \( r.\alpha \) of the rule with \( p.\alpha \) as the head predicate.

3. Examine the rule-node child \( r_0 \) of the goal node \( p^\alpha \). Extending to the right of a node \( r_0 \) will be a chain of nodes called \( r_1, \ldots, r_k \), where \( k \) is the number of subgoals of \( r \). These will have a collection of goal-node children, corresponding to the subgoals of \( r \).
   a) If \( q^{\beta} \) is such a goal node, and \( q \) is an EDB or built-in predicate, leave the corresponding subgoal in \( r.\alpha \) the same as it is in \( r \).
   b) If \( q \) is an IDB predicate, then change the predicate in the corresponding subgoal of \( r.\alpha \) to \( q.\beta \).

**Example 12.31:** Consider the “same-generation” rules of Example 12.30 and the rule/goal graph of Figure 12.22. The IDB predicate \( sg \) has two goal nodes, with adornments \( bf \) and \( bb \). We thus create two predicates, \( sg.bf \) and \( sg.bb \). The first rule, \( r_1 \), has only the EDB subgoal \( person \), so the two copies of \( r_1 \) are the same, except for the head predicate.

The rule \( r_2 \) has an EDB subgoal, \( sg(X,Y,Y) \), and whether the head predicate in \( sg.bf \) or \( sg.bb \), the adornment for this subgoal is found, in Figure 12.22, to be \( bb \). Thus, the bodies of the two versions of \( r_2 \), which we call \( r_{2.bf} \) and \( r_{2.bb} \), both have subgoal \( sg.bb(X,Y,Y) \). The new set of predicates and rules is shown in Figure 12.24.

\[
\begin{align*}
  r_{1.bf}: & \quad sg.bf(X,X) := person(X). \\
  r_{1.bb}: & \quad sg.bf(X,Y) := person(X,Ap) & \text{par}(Y,Yp) & sg.bb(Xp,Yp) \\
  r_{1.bf}: & \quad sg.bb(X,X) := person(X). \\
  r_{2.bf}: & \quad sg.bb(X,Y) := \text{par}(X,Ap) & \text{par}(Y,Yp) & sg.bb(Xp,Yp)
\end{align*}
\]

**Figure 12.24** Split rules for same-generation.

It happens that, should we build the rule/goal tree with root goal \( sg.bf \) and a binding for the first argument, then the adornments tell us the binding patterns of the goal nodes. That is, all occurrences of a goal node with predicate \( sg.bf \) (there is only one — the root) have binding pattern \( bf \), and all occurrences of \( sg.bb \) have both arguments bound. In general, all nodes with predicate \( p.\alpha \) in a rule/goal tree will have at least the bindings indicated by \( \alpha \), but in some situations, there can be other bindings. The next example illustrates the point.

**Example 12.32:** Consider the rules

\[
\begin{align*}
  r_1: & \quad p(X,Y) := a(X,Y). \\
   r_2: & \quad p(X,Y) := q(X,Y) & p(Y,Y). \\
\end{align*}
\]

and let the query goal be \( p^{f_1} \), that is, a query asking for the entire \( p \) relation. The rule/goal graph starting with goal node \( p^{f_1} \) is shown in Figure 12.25.

**Figure 12.25** Rule/goal graph for rules of Example 12.32.

One would expect from Figure 12.25 that all calls to \( p \) are with both arguments free. However, if we expand the rule/goal tree with root goal \( p(X,Y) \), and neither argument bound, we get the first five levels shown in Figure 12.26. The first occurrence of \( r_2 \) has subgoal \( p(Y,Y) \), and neither argument of \( p \) has been bound, either by an occurrence of \( Y \) in a bound argument of the head or by its presence in a previous subgoal. That is what the rule/goal graph of Figure 12.25 predicts. However, by the time we get to the instance of \( r_2 \) in the lower right corner of Figure 12.26, we notice that when subgoal \( p(Y,Y) \) is reached, the value of \( Y \) is bound because of its presence in the previous subgoal \( q(Y,W) \). Thus, \( p^g \) appears to be the proper binding for this subgoal, a fact that is at variance with the rule/goal graph in Figure 12.25.

**Rectifying Subgoals**

It is not hard to see that the source of the problem in Example 12.32 is the fact that the two arguments of subgoal \( p(Y,Y) \) are the same variable. That leads to "aliasing" between \( X \) and \( Y \) in the head of \( r_2 \), and a binding for \( X \) implicitly is a binding for \( Y \) as well. The solution, at least for datalog rules, is to eliminate duplicate occurrences of variables in IDB subgoals and to eliminate constants appearing in subgoals. That is, we must "rectify" subgoals, just as we rectified heads in Section 3.3 (Volume I).\(^{27}\) Note that rectifying heads is not sufficient, as the heads in Example 12.32 are rectified.

We shall say that a subgoal is **rectified** if each of its arguments is a distinct variable. The property of rectified subgoals that we shall exploit is that the

\(^{27}\) A rule was considered rectified in Section 3.3 if the arguments of its head were distinct variables.
unification of a rule head with a rectified subgoal is trivial and results in a rule instance identical to the original rule. This property, in turn, will guarantee that the binding patterns in the rule/goal tree match those predicted by the rule/goal graph. The next algorithm shows how to rectify subgoals. It is guaranteed only to work for datalog, although it can succeed on some nondatalog examples.

**Algorithm 12.7:** Making Binding Patterns Unique.

**INPUT:** A set of datalog rules and a query goal.

**OUTPUT:** A revised set of rules such that in the rule/goal tree expansion of the query goal, each predicate occurs in goal nodes with only one binding pattern. If the given rules are safe, so are the revised rules.

**METHOD:** We begin by rectifying the subgoals. Consider some subgoal that has one or more occurrences of duplicate arguments and/or constant arguments, say \( p(X_1, \ldots, X_k); \) some of the \( X \)'s could be constants. Let \( Y_1, \ldots, Y_m \) be all of the variables appearing among the \( X \)'s; presumably, \( m < k \).

1. Create a new predicate \( p'(Y_1, \ldots, Y_m) \), and substitute it for the subgoal \( p(X_1, \ldots, X_k) \). For example, we would substitute \( p'(U, V) \) for the subgoal \( p(U, U, V, a) \).

2. Consider each rule \( r \) with head \( p(Z_1, \ldots, Z_k) \); assume that variables of \( r \) have been renamed, if necessary, so none are identical to any of the variables among the \( X \)'s. Unify \( p(Z_1, \ldots, Z_k) \) with \( p(X_1, \ldots, X_k) \), substituting \( X \)'s for \( Z \)'s wherever possible, and let the resulting MGU be \( r' \). Create a corresponding rule for predicate \( p' \) by substituting \( r(Z_i) \) for each occurrence of a variable \( Z_i \) in the body of \( r, i = 1, 2, \ldots, k \). Let the head of the new rule be \( p'(\tau(Y_1), \ldots, \tau(Y_m)) \). We claim that the resulting rule is safe if \( r \) is safe. It is left as an exercise to show that \( \tau(Y_i) \), which is \( \tau(X_i) \) for some \( i \) (because \( Y_i = X_i \)), must be equal to \( \tau(Z_i) \). If \( Z_i \) is a constant, then so is \( \tau(Y_i) \), so there could be no violation of safety. If \( Z_i \) is a variable, then it appears in the body of \( r \) because \( r \) is safe. If there are other occurrences of predicate \( p \) that have the same pattern of equalities among its arguments, and the same constants, if any, then we can substitute \( p' \) for \( p \) in these subgoals, provided we make an appropriate selection of variables for these subgoals, analogous to \( Y_1, \ldots, Y_m \) in (1) above.

This process of replacing predicates of subgoals and creating rules for the new predicates can be repeated many times. In fact, applications of the transformation can create new opportunities for its application in the new rules. However, all new predicates have fewer arguments than the predicate from which they were created, so the process cannot go on forever. Thus, at some point, we have a set of rules, none of whose subgoals have repeated variables.

The final step is to construct the rule/goal graph for these rules and the original query goal. Then split predicates to make the bindings for each predicate be unique, as described at the beginning of this section. Let the query goal have predicate \( p \) and binding pattern \( \alpha \); that is, \( \alpha \) has \( \delta \) in exactly those positions that correspond to arguments in which the query provides a constant value. Then revise the query to have predicate \( p, \alpha \) and the same arguments. The result is an equivalent collection of rules and query, in whose rule/goal tree no predicate will appear with two different binding patterns.}

\[ \tau_1: \quad p(X, Y) \quad \leftrightarrow \quad s(X, Y). \]
\[ \tau_2: \quad p(X, Y) \quad \leftrightarrow \quad q(X, W) & p(Y, Y). \]
\[ \tau_3: \quad p_i(Y) \quad \leftrightarrow \quad s(Y, Y). \]
\[ \tau_4: \quad p_i(Y) \quad \leftrightarrow \quad q(Y, W) & p(Y, Y). \]

**Figure 12.27** Rules with rectified subgoals.

\[ ^{28} \text{Note that this transformation still makes sense if the subgoal } p(X_1, \ldots, X_k) \text{ in question is a subgoal of } r, \text{ although renaming of variables would then take place.} \]
Example 12.33: Consider the rules of Example 12.32. There is only one occurrence of duplicate variables, subgoal \( p(Y, Y) \) in \( r_2 \). We thus create new predicate \( p_1 \), and modify \( r_2 \) to be

\[ r_2 : p(X, Y) = q(X, W) \land p_1(Y). \]

We also create new versions of \( r_1 \) and \( r_3 \) for predicate \( p_1 \). That is, we must unify the heads of \( r_1 \) and \( r_3 \) with the offending subgoal, \( p(Y, Y) \). Before doing so, we rename the variable \( Y \) in \( r_1 \) and \( r_3 \). Let us think of it as \( U \). Then both rules have head \( p(X, U) \), and the unifications each equate \( X \), \( Y \), and \( U \). We pick \( Y \) as the representative for all; that is, \( \tau(X) = \tau(Y) = \tau(U) = Y \); the resulting rules are shown in Figure 12.27. The rule/goal graph for these rules is in Figure 12.28.

![Rule/goal graph for rules of Figure 12.27.](image)

Theorem 12.3: Suppose we build the rule/goal tree for the given query and for the rules constructed by Algorithm 12.7. Then each predicate \( p, \alpha \) of these rules appears in goal nodes only with the binding pattern (set of bound arguments) \( \alpha \).

Proof: We shall show by induction on the depth of an occurrence of a goal node with predicate \( p, \alpha \), that \( \alpha \) is the binding pattern for this goal node; that is, the bound arguments correspond exactly to the \( b \)'s in \( \alpha \). The basis is the root node of the rule/goal tree, and Algorithm 12.7 selects the predicate for the revised query so that the binding pattern is correct.

For the induction, we first note that since subgoals are rectified, all unifications are trivial, so every rule node in the tree corresponds to a rule instance that is identical to the original rule, except perhaps for renaming of variables, which is of no significance. Thus, for each rule node in the rule/goal tree, the sideways information passing binds exactly the same variables after each subgoal that are bound in the expansion of that rule in the rule/goal graph. It follows that if, say, a goal node \( p, \alpha \) is given the binding pattern \( \alpha \), then each goal node two levels below, say \( g, \beta \), will be given the binding pattern \( \beta \). Further details are left as an exercise.

12.10 REORDERING SUBGOALS

When we construct the rule/goal graph, we have the option to reorder the subgoals of a rule as we expand the corresponding rule nodes. If we follow Algorithm 12.7 and create a new predicate \( p, \alpha \) for each adornment \( \alpha \) of each original predicate \( p \), then we can order the subgoals of the rules for \( p, \alpha \) as we wish, independently of the order of the corresponding subgoals in any other variant of \( p \). In fact, even if we do not split \( p \), we can compute the rule/goal graph in such a way that different subgoal orders are used for different instances of the same rule. Our justification is that the least fixed-point semantics of rules, as defined in Sections 3.4 and 12.2, does not depend on the order in which we list the subgoals.

The problem of ordering subgoals in a rule is similar to the problem of ordering the join of many relations, which we discussed in Chapter 11. However, when we evaluate rules, we ordinarily do not have available relations for all the subgoals initially. Rather, through the process of sideways information passing, we develop bindings for some of the subgoals, and only then do we attempt to instantiate the relevant portion of the relation for that subgoal. To do otherwise might force us to compute the entire relation for some IDB predicates, when all we wanted was a small fraction of that relation.

Further, it is possible that some subgoals can only be evaluated with certain binding patterns. Some examples of this phenomenon are the following.

1. Built-in predicates usually require some bindings. For example, the subgoal \( X = Y \) cannot be "evaluated" unless at least one of \( X \) and \( Y \) is bound. If, say, \( X \) is bound to the set of values \( \{a_1, \ldots, a_n\} \), then we can use the finite relation \( \{(a_1, a_1), \ldots, (a_n, a_n)\} \) for subgoal \( X = Y \). If neither \( X \) nor \( Y \) is bound, then there is an infinite set of pairs that makes the subgoal true.

2. Similarly, subgoal \( X < Y \) cannot be "evaluated" unless both \( X \) and \( Y \) are bound; otherwise an infinite set of pairs satisfies this subgoal.
Example 12.33: Consider the rules of Example 12.32. There is only one occurrence of duplicate variables, subgoal \( p(Y, Y) \) in \( r_2 \). We thus create new predicate \( p_1 \), and modify \( r_2 \) to be

\[
r_2: \quad p(X, Y) \leftarrow q(X, W) \land p_1(Y).
\]

We also create new versions of \( r_1 \) and \( r_2 \) for predicate \( p_1 \). That is, we must unify the heads of \( r_1 \) and \( r_2 \) with the offending subgoal, \( p(Y, Y) \). Before doing so, we rename the variable \( Y \) in \( r_1 \) and \( r_2 \); let us think of it as \( U \). Then both rules have head \( p(X, U) \), and the unifications each equate \( X \), \( Y \), and \( U \). We pick \( Y \) as the representative for all; that is, \( \tau(X) = \tau(Y) = \tau(U) = Y \); the resulting rules are shown in Figure 12.27. The rule/goal graph for these rules is in Figure 12.28. □

![Rule/goal graph](image)

Figure 12.28 Rule/goal graph for rules of Figure 12.27.

Theorem 12.3: Suppose we build the rule/goal tree for the given query and for the rules constructed by Algorithm 12.7. Then each predicate \( p, \alpha \) of those rules appears in goal nodes only with the binding pattern (set of bound arguments) \( \alpha \).

Proof: We shall show by induction on the depth of an occurrence of a goal node with predicate \( p, \alpha \), that \( \alpha \) is the binding pattern for this goal node; that is, the bound arguments correspond exactly to the \( \beta \)'s in \( \alpha \). The basis is the root node of the rule/goal tree, and Algorithm 12.7 selects the predicate for the revised query so that the binding pattern is correct.

For the induction, we first note that since subgoals are rectified, all unifications are trivial, so every rule node in the tree corresponds to a rule instance that is identical to the original rule, except perhaps for renaming of variables, which is of no significance. Thus, for each rule node in the rule/goal tree, the sideways information passing binds exactly the same variables after each subgoal that are bound in the expansion of that rule in the rule/goal graph. It follows that if, say, a goal node \( p, \alpha \) is given the binding pattern \( \alpha \), then each goal node two levels below, say \( q, \beta \), will be given the binding pattern \( \beta \). Further details are left as an exercise. □

12.10 REORDERING SUBGOALS

When we construct the rule/goal graph, we have the option to reorder the subgoals of a rule as we expand the corresponding rule nodes. If we follow Algorithm 12.7 and create a new predicate \( p, \alpha \) for each adornment \( \alpha \) of each original predicate \( p \), then we can order the subgoals of the rules for \( p, \alpha \) as we wish, independently of the order of the corresponding subgoals in any other variant of \( p \). In fact, even if we do not split \( p \), we can compute the rule/goal graph in such a way that different subgoal orders are used for different instances of the same rule. Our justification is that the least fixed-point semantics of rules, as defined in Sections 3.4 and 12.2, does not depend on the order in which we list the subgoals.

The problem of ordering subgoals in a rule is similar to the problem of ordering the join of many relations, which we discussed in Chapter 11. However, when we evaluate rules, we ordinarily do not have available relations for all the subgoals initially. Rather, through the process of sideways information passing, we develop bindings for some of the subgoals, and only then do we attempt to instantiate the relevant portion of the relation for that subgoal. To do otherwise might force us to compute the entire relation for some IDB predicates, when all we wanted was a small fraction of that relation.

Further, it is possible that some subgoals can only be evaluated with certain binding patterns. Some examples of this phenomenon are the following.

1. Built-in predicates usually require some bindings. For example, the subgoal \( X = Y \) cannot be "evaluated" unless at least one of \( X \) and \( Y \) is bound. If, say, \( X \) is bound to the set of values \( \{a_1, \ldots, a_n\} \), then we can use the finite relation \( \{(a_1, a_1), \ldots, (a_n, a_n)\} \) for subgoal \( X = Y \). If neither \( X \) nor \( Y \) is bound, then there is an infinite set of pairs that makes the subgoal true.

2. Similarly, subgoal \( X < Y \) cannot be "evaluated" unless both \( X \) and \( Y \) are bound; otherwise an infinite set of pairs satisfies this subgoal.
3. We may wish to declare that EDB relations cannot be evaluated unless there is at least one bound attribute on which that EDB relation has an index. For if there is no such attribute, then we must examine the entire relation, even if there are some bound arguments whose corresponding attributes do not have indices.

4. An IDB predicate whose rules involve function symbols may be evaluable if certain arguments are bound, but not if they are free; the next example illustrates the point.

**Example 12.34:** Suppose we have a database of "good" elements, that is, a unary relation good. Let us construct lists by using the binary function symbol cons. That is, cons(H,T) represents the list with head element H and tail (remaining elements) T. We shall also use the constant nil to represent the empty list. Then we can define "good lists" by the rules

\[
\begin{align*}
r_1 & : \text{goodlist(nil).} \\
r_2 & : \text{goodlist(cons(H,T)) :- good(H) & goodlist(T).}
\end{align*}
\]

That is, the empty list is good, and a list is good if the first element H is good, and the balance of the list, represented by the tail T \(r_2\) is a good list.

For example, the list \(a, b, c\) would be represented by the term

\[\text{cons}(\text{cons}(\text{cons}(a, b), c), \text{nil})\]  \hspace{1cm} (12.11)

Assume \(a, b, \) and \(c\) are each in the relation for good. Then rule \(r_2\) tells us that \(a, b, c\) is a good list if \(b, c\) is a good list. Further, \(b, c\) is good if list \(c\) is a good list. Finally, \(r_3\) says that \(c\) is a good list if the empty list is good, and \(r_1\) affirms that the empty list is good.

If we ask a query of the form \(\text{goodlist}^b\), say a query with argument (12.11), then we can answer the query. For instance, we can deduce, by rule/goal tree expansion, that (12.11) is a good list, provided \(a, b, \) and \(c\) are good. However, if we pose the query \(\text{goodlist}(X)\), that is, find all the good lists, we have an instance of the adored goal \(\text{goodlist}^g\). The result of this query is infinite whenever EDB relation \(\text{good}\) is nonempty. Thus, we cannot expect rule/goal tree expansion, or any other query evaluation algorithm, to return an answer. The conclusion is that, while we can allow a subgoal with binding pattern \(\text{goodlist}^g\), we cannot allow one with pattern \(\text{goodlist}^b\). □

**The Feasibility Problem for Rule/Goal Graphs**

One can develop a number of heuristics for picking an order for subgoals, and we shall return to this topic shortly. However, to begin, let us pose the question of ordering subgoals as one of feasibility. That is, suppose we are told which adorments for EDB predicates are permissible and which are not. Can we build a rule/goal graph for a given query and given rules, that uses only permissible adorments on the EDB goals?²⁹

**Example 12.35:** Let us consider a slightly more complex version of the "same-generation" rules

\[
\begin{align*}
r_1 & : \text{sg}(X, X) :- \text{person}(X). \\
r_2 & : \text{sg}(X, Y) :- \text{par}_1(X, X, Xp) \& \text{par}_2(Y, Yp) \& \text{sg}(Y, Xp). 
\end{align*}
\]

The difference between these rules and the ones studied previously is that here we have reversed the order of the arguments in the \(\text{sg}\) subgoal. Since \(\text{sg}\) is evidently symmetric in its two arguments, it doesn't matter whether we say \(\text{sg}(Y, X, Xp)\), as we have done here, or \(\text{sg}(X, Y, Yp)\), as we did previously. However, the new order of arguments does affect the binding patterns for the \(\text{sg}\) predicate, and it will illustrate the advantage of ordering subgoals differently for different adorments.

In our new version of the rules, we also show subscripts 1 and 2 for the \(\text{par}\) subgoals. That notation does not imply we are using two different relations for parenthood. Rather, we shall carry along the subscripts to indicate which \(\text{par}\) subgoal corresponds to various \(\text{par}\) goal nodes in rule/goal graphs that we develop.

Suppose first that we use the subgoals in the order written. Then, starting with the \(\text{sg}^g\) query goal, we develop the rule/goal graph shown in Figure 12.29. For example, the instance of \(r_2\) for the \(\text{sg}^g\) goal gets a binding for \(X\) from the head, binds \(Xp\) when it reaches \(r_3\), and calls \(\text{par}_2^g\), since neither argument \(Y\) nor \(Yp\) of \(\text{par}_2\) is bound at that point. Finally, \(\text{sg}^b\) is called, since both arguments \(Yp\) and \(Xp\) are bound when we reach the third subgoal. The expansion of \(\text{sg}^g\) is similar to the expansion of \(\text{sg}^b\), and we omit it.

Now consider the rule/goal graph of Figure 12.30. Here, we again start from query goal \(\text{sg}^g\), but we reorder subgoals judiciously, preferring subgoals with the most bound arguments. Then we avoid ever having to evaluate \(\text{par}_1^g\); that is, we never have to access the entire relation for \(\text{par}\). That could well be a significant saving, since with indices on both attributes, the EDB relation \(\text{par}\) can, normally, be accessed far more efficiently with either argument bound to a finite set, than with both arguments free.

Thus, in the instance of \(r_2\) for the \(\text{sg}^g\) goal, we first work on \(\text{par}_2^g\), which provides a binding for \(Xp\). Since we prefer subgoals with bound arguments, we next choose \(\text{sg}(Y, Xp)\), whose adorment is \(\text{fb}\), as \(Xp\) is now bound and \(Yp\) is not. Last, we work on \(\text{par}_1^g\). The instance of \(r_2\) for the \(\text{sg}^b\) goal is treated symmetrically. We first work on \(\text{par}_1^g\), since we have only a binding for \(Y\).

²⁹ More generally, we could build the rule/goal graph for one or more mutually recursive EDB predicates and imagine that all other predicates appearing in subgoals are EDB subgoals. In reality, these subgoals could have "lower-level" EDB predicates, whose possible adorments have already been determined. We shall examine a system using this generalization in Section 16.2.
3. We may wish to declare that EDB relations cannot be evaluated unless there is at least one bound attribute on which that EDB relation has an index. For if there is no such attribute, then we must examine the entire relation, even if there are some bound arguments whose corresponding attributes do not have indices.

4. An IDB predicate whose rules involve function symbols may be evaluable if certain arguments are bound, but not if they are free; the next example illustrates the point.

Example 12.34: Suppose we have a database of "good" elements, that is, a unary relation good. Let us construct lists by using the binary function symbol cons. That is, cons(H,T) represents the list with head element H and tail (remaining elements) T. We shall also use the constant nil to represent the empty list. Then we can define "good lists" by the rules

\[ r_1: \text{goodlist}(nil). \]
\[ r_2: \text{goodlist}(\text{cons}(H,T)) \leftarrow \text{good}(H) \& \text{goodlist}(T). \]

That is, the empty list is good, and a list is good if the first element H is good, and the balance of the list, represented by the tail T in \( r_2 \), is a good list.

For example, the list \( a, b, c \) would be represented by the term \( \text{cons}(a, \text{cons}(b, \text{cons}(c, nil))) \).

Assume \( a, b, \) and \( c \) are each in the relation for good. Then rule \( r_2 \) tells us that \( a, b, c \) is a good list if \( b, c \) is a good list. Further, \( b, c \) is good if list \( c \) is a good list. Finally, \( r_2 \) says that \( c \) is a good list if the empty list is good, and \( r_1 \) affirms that the empty list is good.

If we ask a query of the form \( \text{goodlist}^k \), say a query with argument \( (12.11) \), then we can answer the query. For instance, we can deduce, by rule/goal tree expansion, that \( (12.11) \) is a good list, provided \( a, b, \) and \( c \) are good. However, if we pose the query \( \text{goodlist}(X) \), that is, find all the good lists, we have an instance of the adored goal \( \text{goodlist}^\infty \). The result of this query is infinite whenever EDB relation \( \text{good} \) is nonempty. Thus, we cannot expect rule/goal tree expansion, or any other query evaluation algorithm, to return an answer. The conclusion is that, while we can allow a subgoal with binding pattern \( \text{goodlist}^k \), we cannot allow one with pattern \( \text{goodlist}^\infty \).

The Feasibility Problem for Rule/Goal Graphs

One can develop a number of heuristics for picking an order for subgoals, and we shall return to this topic shortly. However, to begin, let us pose the question of ordering subgoals as one of feasibility. That is, suppose we are told which adornments for EDB predicates are permissible and which are not. Can we build a rule/goal graph for a given query and given rules, that uses only permissible adornments on the EDB goals?29

Example 12.35: Let us consider a slightly more complex version of the "same-generation" rules

\[ r_1: \text{sg}(X, 1) \leftarrow \text{person}(X). \]
\[ r_2: \text{sg}(X, Y) \leftarrow \text{par}_1(X, Xp) \& \text{par}_2(Y, Yp) \& \text{sg}(Yp, Xp). \]

The difference between these rules and the ones studied previously is that here we have reversed the order of the arguments in the \( \text{sg} \) subgoal. Since \( \text{sg} \) is evidently symmetric in its two arguments, it doesn't matter whether we say \( \text{sg}(Yp, Xp) \), as we have done here, or \( \text{sg}(Xp, Yp) \), as we did previously. However, the new order of arguments does affect the binding patterns for the \( \text{sg} \) predicate, and it will illustrate the advantage of ordering subgoals differently for different adornments.

In our new version of the rules, we also show subscripts 1 and 2 for the \( \text{par} \) subgoals. That notation does not imply we are using two different relations for parenthood. Rather, we shall carry along the subscripts to indicate which \( \text{par} \) subgoal corresponds to various \( \text{par} \) goal nodes in rule/goal graphs that we develop.

Suppose first that we use the subgoals in the order written. Then, starting with the \( \text{sg}^k \) query goal, we develop the rule/goal graph shown in Figure 12.29. For example, the instance of \( r_2 \) for the \( \text{sg}^k \) goal gets a binding for \( X \) from the head, binds \( Xp \) when it reaches \( r_1 \), then calls \( \text{par}_2(X, Xp) \), since neither argument \( Y \) nor \( Yp \) of \( \text{par}_2 \) is bound at that point. Finally, \( \text{sg}^k \) is called, since both arguments \( Yp \) and \( Xp \) are bound when we reach the third subgoal. The expansion of \( \text{sg}^k \) is similar to the expansion of \( \text{sg}^h \), and we omit it.

Now consider the rule/goal graph of Figure 12.30. Here, we again start from query goal \( \text{sg}^h \), but we reorder subgoals judiciously, preferring subgoals with the most bound arguments. Then we avoid ever having to evaluate \( \text{par}_1^h \); that is, we never have to access the entire relation for \( \text{par} \). That could well be a significant saving, since with indices on both attributes, the EDB relation \( \text{par} \) can, normally, be accessed far more efficiently with either argument bound to a finite set, than with both arguments free.

Thus, in the instance of \( r_2 \) for the goal \( \text{sg}^h \), we first work on \( \text{par}_2^h \), which provides a binding for \( Xp \). Since we prefer subgoals with bound arguments, we next choose \( \text{sg}(Yp, Xp) \), whose adornment is \( \text{par}_2 \), as \( Xp \) is now bound and \( Yp \) is not. Last, we work on \( \text{par}_1^h \). The instance of \( r_2 \) for the \( \text{sg}^h \) goal is treated symmetrically. We first work on \( \text{par}_2^h \), since we have only a binding for \( Y \).

29 More generally, we could build the rule/goal graph for one or more mutually recursive EDB predicates and imagine that all other predicates appearing in subgoals are EDB subgoals. In reality, these subgoals could have "lower-level" EDB predicates, whose possible adornments have already been determined. We shall examine a system using this generalization in Section 16.2.
Then we work on $sg^f$, and last on $par^f$. Note that the adornments for the two instances of $r_2$ are similar, but the order of subgoals is different for the two rules, and neither order matches the natural order in which the subgoals were written. □

The Bound-is-Easier Assumption

Intuitively, replacing a free argument of a predicate by a bound argument cannot make the predicate harder to evaluate. There is good reason to believe this principle, and we give its justification in the next paragraph. However, since we cannot prove it for any possible model of computation, we shall refer to it as an assumption, the bound-is-easier assumption. The motivation for making this assumption is that it simplifies subgoal ordering significantly. It allows us, once we find a feasible choice for the next subgoal, to "cast that choice in concrete," never having to backtrack to find another feasible choice.

Formally, define the relation $\leq$ on adornments by saying $\alpha \leq \beta$ if $\beta$ has $\beta$ in every position where $\alpha$ has $\beta$. For example, $ff \leq fbb$, and $fbf \leq fbf$. However, $bfff \leq fbbf$ is false, because of the first positions of the two adornments. Now, suppose $\alpha \leq \beta$. If we have an algorithm to evaluate $p^\alpha$, then surely we can evaluate $p^\beta$: just evaluate the relation for $p^\alpha$ with the given bindings for those arguments that are bound in $\alpha$. Then, if $\beta$ has additional bound arguments, apply a selection to the relation for $p^\alpha$ to get the relation for $p^\beta$. However, the converse is not true; we might be able to evaluate $p^\beta$ but not $p^\alpha$. Example 12.34 covered a simple case where $\alpha = f$ and $\beta = b$.

Thus, we shall adopt the bound-is-easier assumption; that is, we shall assume that whenever we can evaluate $p^\alpha$, we can also evaluate all adored goals $p^\alpha$, where $\alpha \leq \beta$. That assumption has two useful consequences. First, when listing the adornments for which we can evaluate a predicate, we need to list only those adornments that are minimally bound.

**Example 12.36:** Suppose EDB relation $par$ has indices on both arguments, and it is required that some bound argument have an index, if we are to evaluate an EDB subgoal. We need only state that $par^f$ and $par^f$ are permissible. It then follows that $par^f$ is also permissible. However, $par^f$ would not be permissible.

Second, the bound-is-easier assumption allows us to select an ordering for subgoals of a rule in a greedy way. Suppose we have a set of bound variables for the head of rule $r$, and we know that some subgoal $p^\alpha$, where $\alpha$ is the appropriate adornment given the variables bound by the head, is permissible. Then we cannot err if we assume that $p^\alpha$ is the first subgoal for $r$.

More precisely, suppose that after picking $p^\alpha$ as the first subgoal, we discover that no order for the remaining subgoals gives us permissible adornments for all subgoals. However, suppose that had we chosen some other subgoal, $p^\beta$, then...
first, then a successful order $L$ for all subgoals can be found. In $L$, the subgoal $p$ appears somewhere. Delete it and place $p^*$ at the front of $L$ to make a new ordering $M$. We claim that $M$ has no impermissible subgoals. Suppose in $M$ that $s^j$ is impermissible. Then in $L$, the corresponding adornment is $s^j$. We can see that $j \leq \gamma$. The reason is that $p^*$, at the beginning of $M$, binds any variable that the corresponding subgoal did in the middle of $L$. Thus, whether $s$ precedes or follows $p$ in $L$, every variable that was bound when we reach subgoal $s$ in $L$ will be bound when we reach it in $M$. Since $p^*$ is known to be permissible, and since $s^j$ must be permissible because $s^j$ is, we conclude that $M$ has only permissible subgoals. Therefore, our assumption that no ordering could succeed if it began with $p^*$ is false.

The above observations are generalized in the following lemma. However, before proceeding, let us observe that the notion of permissible adornments can apply to IDB, as well as EDB, subgoals. That is, we specify permissible for EDB subgoals, but, as a consequence, some adornments for IDB predicates become permissible, in the sense that they can appear in a rule/goal graph, and others are not permissible, meaning that there is no rule/goal graph in which they can appear. In the following lemma, all subgoals, whether IDB or EDB, are assumed to have their set of permissible adornments known.

Lemma 12.2: Under the bound-is-easier assumption, suppose we have a rule with subgoals $G_1, \ldots, G_n$, and suppose that, based on particular bindings of the head, we have determined that the first $k$ subgoals could be $G_1, \ldots, G_k$: that is, each of these subgoals has a permissible adornment in the order shown. Suppose also that $G_j$, one of the remaining subgoals, is permissible with the bindings for variables implied by the head and the $k$ previously selected subgoals. Then we may assume without loss of generality that $G_j$ is the next subgoal. More precisely, if there is any feasible order for the subgoals at all, then there is one that begins $G_1, \ldots, G_k, G_j$.

Proof: The proof is a simple generalization of the argument given above, and we leave it as an exercise.

Example 12.37: To make the point more concrete, consider the rule

$p(X, Y) :- q(X, Z), s(Z, W), t(W, Y)$.

Suppose that adorned goals $q^H, q^b, s^H, t^H$, and of course, their "more bound" versions, are permissible; note that $q^H$ is not permissible. Let us give the goal $q^H$, so we start with a binding for variable $X$. Our heuristic, that we prefer subgoals with as many bound variables as possible, suggests that we should pick $q(X, Z)$ to be first. Since $q^H$ is permissible, that is an acceptable choice.

We now have a binding for $Z$ as well as $X$, and we may next tackle $s^H$, another permissible subgoal. However, for the third subgoal, we can only choose

$q(W, Y)$. $Y$ is not yet bound, and $q^H$ is not a permissible adornment. Thus, we have "lost"; it is not possible to find a feasible ordering for the subgoals.

However, the fault does not lie with our choice of $q(X, Z)$ as the first subgoal. We could have chosen $s(Z, W)$ first, since $s^H$ is permissible. However, then, only $q(X, Z)$ could go second, and when we are left with $t(W, Y)$, we have the same set of bound variables as before—all but $Y$. Since no adornment of $t(W, Y)$ is permissible unless $Y$ is bound, we still fail, and indeed there is no feasible subgoal order for this rule.

An Algorithm for Feasibility of Rule/Goal Graphs

Suppose we have an algorithm $A$ that takes the following information.

1. A rule $H :- G_1 \land \cdots \land G_n$.
2. An adornment for the head predicate of the rule.
3. Information telling which adornments are permissible for the predicates of all the subgoals of the rule.

Further, suppose Algorithm $A$ finds, if it exists, a feasible order for the subgoals, that is, an order in which the adornment of each subgoal is

a) Permissible, and
b) Correct, given the bindings for the head and the bindings of variables implied by the previous subgoals in the selected order.

Then we can find a rule/goal graph for a given query goal by the following algorithm.

Algorithm 12.8: Feasibility Test for Rule/Goal Graphs.

INPUT: We are given a collection of rules, an adorned query goal $p^*$, a list of the permissible adornments for the EDB goals, and an algorithm $A$, as described above, for finding feasible orders of subgoals. We make the bound-is-easier assumption, so the list of permissible EDB goals implicitly includes all their more bound versions; that is, if $q^H$ is on the list, then $q^b$ is implicitly on the list whenever $b \leq \gamma$, even if we do not say so explicitly.

OUTPUT: A rule/goal graph for $p^*$, if one exists, or an indication that there is none.

METHOD: We maintain a set $F$ of forbidden adorned goals. Initially, $F$ consists of those adorned EDB goals that are not among the given permissible adorned EDB goals; no IDB goal is initially in $F$. Intuitively, we are going to make the optimistic assumption that all IDB predicates are permissible with any adornment, until the algorithm $A$ tells us otherwise, by failing to find a feasible ordering for one of the rules for that adorned IDB predicate.

We also maintain a set $I$ of interesting adorned IDB goals. Initially, only $p^*$ is in $I$. Later, we place an adorned goal in $I$ if it appears in the ordering selected by $A$ for one of the rules of an adorned IDB goal already in $I$. We
remove $q^g$ from $I$ and place it in $F$ if we find that there is no feasible ordering for one or more of the rules for $q$.\footnote{We could attempt to remove adorned goals from $I$ if we find them no longer "interesting," but this nuance will not be considered here.}

for each adorned goal $q^g$ in $I$ do
  for each rule $r$ for predicate $q$ do begin
    use algorithm $A$ to find a feasible ordering for
    the subgoals of $r$, given that $\beta$ is the adornment
    for the head of $r$ and that $F$ is the set of
    impermissible adorned goals;
    if an ordering is found then
      add to $I$ the adorned predicate for each IDB
      subgoal in the selected order
    else begin
      delete $q^g$ from $I$;
      insert $q^g$ into $F$
      \(\exists\) by implication, $q^g$ is also put
      in $F$ for all $\gamma \leq \beta$ \(*\)
    end
  end

Figure 12.31 One pass of Algorithm 12.8.

The heart of the algorithm is shown in Figure 12.31. There, we take our current knowledge about what adorned goals are not permissible, as represented by $F$, and our current knowledge about what adorned IDB goals we have to handle, as represented by $I$, and check that there is a feasible order for every rule for every interesting adorned goal. Those adorned goals found not to have a subgoal order for some rule are discovered to be forbidden, and are therefore moved from $I$ to $F$. Further, we may, while ordering subgoals of one rule, find an adornment for one of its IDB predicates that we have not seen before. If so, we must add this adorned goal to the set $I$.

The complete algorithm consists of the initialization described above, and an iteration, in which we perform the steps of Figure 12.31 until one of two things happens.

1. We remove $p^h$ from $I$ and place it in $F$.
2. On one pass, we do not change $I$.

In case (1), we have demonstrated that there are no orders for the subgoals of the rules for $p$ and the rules for any other predicates appearing in $I$, that make all the subgoals of all these rules have permissible adornments, according to the given information for the EDB predicates. Thus, there is no rule/goal graph containing $p^h$.

In case (2), we know that each adorned goal in $I$ has, for each of its rules, an ordering of its subgoals giving each subgoal an adornment that

a) is in $I$ if it is an IDB subgoal, and
b) is not in $F$ if it is an EDB subgoal.

Thus, we can construct a rule/goal graph containing $p^h$ as follows. For each adorned goal $q^g$ in $I$, we have a goal node, and the children of this node are the rule nodes for $q$, with subscript 0 and their proper superscripts representing bound variables. Descending in a chain to the right from each of these rule nodes $r^0_{(i[M])}$ are nodes $r^\gamma_{(i[M])}$, for $i = 1, 2, \ldots, k - 1$, if rule $r$ has $k$ subgoals. Here, $L_i$ and $M_i$ are intended to represent appropriate lists of variables.

Each of the nodes $r^\gamma_{(i[M])}$, for $i = 0, 1, \ldots, k - 1$, also has a goal-node child. That child has the predicate of the $i$th subgoal of $r$ in the selected order, and has the appropriate adornment. By (a) above, if the subgoal is an IDB subgoal, then this adorned goal is also in the constructed rule/goal graph. By (b), if the subgoal is EDB, then it has a permissible adornment, and we may add a goal node for this adorned EDB predicate. Thus, every node in the constructed rule/goal graph has the requisite children also in the rule/goal graph, and all EDB goal nodes are permissible. We have therefore constructed a feasible rule/goal graph containing the query goal.\footnote{In some cases, where adorned goals become interesting in one pass, and later become uninteresting because there is no longer any way to reach them from the query goal, we shall be able to trim parts of the constructed rule/goal graph and still have a feasible rule/goal graph for the query.} \(\square\)

Example 12.38: Let us reconsider the “twisted” same-generation rules of Example 12.35. Suppose the query corresponds to adorned goal $sp^s$, and the algorithm $A$ orders subgoals heuristically, by choosing at each step that unchosen subgoal that

1. Has a permissible binding pattern,
2. Has as many bound arguments as any unchosen subgoal satisfying (1), and
3. Appears to the left of any other unchosen subgoal satisfying (1) and (2).

Also, assume that $par^f$ and $person^f$ are impermissible adornments, but any other adornment for $par$ and $person$ is permissible.

Initially, $I = \{sp^s\}$ and $F = \{par^f, person^f\}$. For $sp^s$, algorithm $A$ picks the only possible order for rule $r_1$, since $person^h$ is not in $F$. $A$ succeeds on $r_1$. For $r_2$, $A$ picks the order $par^f, sp^s, par^f$. The IDB adorned subgoal
$sg^b$ is added to $I$, so on the next pass, $A$ must find orders for the subgoals of the two rules with both adornments for the heads, $bf$ and $fb$. Since no more adorned goals were added to $F$, the treatment of $sg^b$ is the same. For $sg^b$, rule $r_2$ again has the only possible order, and for $r_1$, $A$ picks the order $par_f$, $sg^b$. Now, no more adorned IDB goals must be added to $I$, and we have succeeded in finding orders for all the rules for all the members of $I$, without using any adorned goal in $F$. The resulting rule/goal graph was exhibited in Figure 12.30.

**Example 12.39:** To see some of the nuances of Algorithm 12.8, consider the nonrecursive rules

$\begin{align*}
\text{r}_1 & : \quad p(x, y) \leftarrow q(x, z) \land r(z, y).
\text{r}_2 & : \quad q(x, z) \leftarrow s(x) \land t(z).
\end{align*}$

Let the only forbidden adorned EDB goal be $t^f$; all adornments for EDB predicates $r$ and $s$ are permissible. Start with query goal $p^b$, and suppose that the ordering algorithm is $A$ from Example 12.38. Initially, $I = \{p^b\}$ and $F = \{t^f\}$. We thus apply $A$ to order $r_1$, obtaining the list of subgoals $q^b$, $r^b$. We then add adorned IDB goal $q^b$ to $I$.

On the next pass, we treat $p^b$ the same. However, $A$ finds no order for the subgoals of $r_2$, with adornment $bf$ for the head, since $t^f$ is forbidden, and there is no way to bind the argument $Z$ of $r_2$, if $Z$ is not bound by the head of $r_2$. Thus, we move $q^b$ to $F$, leaving $I = \{p^b\}$ and $F = \{t^f, q^b\}$.

On the third pass, we again tackle $r_1$ with head binding pattern $bf$. However, now $q^b$ is forbidden, so algorithm $A$ can only select the order $r^f, q^b$, which is feasible, since neither adorned goal is in $F$. Thus, $q^b$ is added to $I$.

Now, on the fourth pass, $p^b$ is handled the same way as on the third pass, but we must also consider $r_2$ with head binding pattern $bf$. Now $A$ produces the order of subgoals $s^b, t^b$, which is feasible. On this pass, we add no adorned IDB goal to $I$, and we found feasible orders for all $s^b$ and $t^b$, which is feasible. Thus, algorithm $A$ terminates successfully. The resulting rule/goal graph is shown in Figure 12.32.

**Theorem 12.4:** Algorithm 12.8 correctly determines if there is a feasible rule/goal graph including the given query goal.

**Proof:** In the description of the algorithm, we showed how to construct a feasible rule/goal graph when the algorithm terminates with success. We must consider the converse: why can there be no rule/goal graph containing the query goal if the algorithm places the query goal in $F$? On the assumption we can show by induction on the number of adorned goals placed in $F$ that no adorned goal $g^q$ in $F$ can appear in no rule/goal graph. Intuitively, the reason is that we can put $g^q$ in $F$ only when there exists a rule for $q$ such that every order of the subgoals for that rule must use a subgoal with an adornment already in $F$. The easy inductive proof is left as an exercise. Thus, when the query goal is placed in $F$, it too can appear in no rule/goal graph, and the theorem is proved.

**Theorem 12.5:** Suppose that algorithm $A$ has running time $T(m)$ on a rule of length $m$. Then on rules of total length $n$, whose IDB predicates have arity no greater than $k$, the running time of Algorithm 12.8 is $O(n^{2k+2}T(n))$.

**Proof:** If the rules are of total length $n$, there can be no more than $n$ IDB predicates. As these predicates have arity at most $k$, there can be no more than $n^k$ adornments for each, or a total of $n^{k+1}$ adorned goals. On each round of Algorithm 12.8, we must either terminate, move an adorned goal from $I$ to $F$, or add to $I$ a new adorned goal, one that has never before been in $I$. Thus, there can be no more than $2n^{k+1}$ rounds. On each round, the principal cost is finding an ordering for each rule for each adorned goal in $I$. There are at most $n^{k+1}$ members of $I$, at most $n$ rules for each, and at most $T(n)$ time is taken ordering any one rule with a given binding pattern for the head. Thus, each round takes time at most $n^{k+2}T(n)$, and the theorem follows when we multiply this quantity by the maximum number of rounds.

**Heuristic Ordering Algorithms**

Using the bound is easier assumption, there are reasonable choices for the algorithm $A$ that are at most quadratic in $n$, the length of the rule; that is, $T(n)$ in Theorem 12.5 can be $O(n^2)$. For example, we could use the heuristic suggested in Example 12.38, of favoring subgoals with the most bound arguments.
provided that the binding pattern makes the subgoal permissible, and breaking ties in favor of the leftmost subgoal. This heuristic A is shown in detail in Figure 12.33.

(1) \( V := all \ variables \ bound \ by \ the \ head; \)
(2) mark all subgoals “unchosen”;  
(3) for \( i := 1 \) to \( m \) do begin  
(4) \( b := -1; \)
(5) for each unchosen subgoal \( G \) do begin 
(6) find the bound arguments of \( G \), given that \( V \) is the set of bound variables;  
(7) if there are \( c > b \) bound arguments of \( G \) and with this binding pattern \( G \) is permissible then begin 
(8) \( b := c; \)
(9) \( H := G \)
end;  
(10) if \( b \neq -1 \) then begin 
(11) mark \( H \) “chosen”;  
(12) add to \( V \) all variables appearing in \( H \)
end  
(13) else fail
end

Figure 12.33 Subgoal ordering heuristic.

We suppose that the algorithm of Figure 12.33 is called with a rule with \( m \) subgoals and a given binding pattern for its head. We are also given a set of forbidden adjoined goals, referred to in line (7). Several local variables are used.

1. \( V \) is a set of variables, those that are bound, either because they are bound in the head \( [V \) is initialized in line (1)], or because they appear in a chosen subgoal [line (12)].

2. In the loop of lines (5) through (9), we select an unchosen subgoal with the largest number of bound arguments. Variable \( b \) is used to record the greatest number of bound arguments seen so far; it is initialized to \(-1\) at line (4) so that some choice will always be made; that is, the test of line (7) will eventually succeed if there are any permissible, unchosen subgoals. Variable \( H \) is used to record the current candidate for the subgoal with the most bound arguments. Variable \( c \) is the number of bound arguments for the subgoal under consideration.

EXERCISES

After considering all subgoals, if there are any feasible, unchosen subgoals, we make \( H \) chosen and adjust \( V \) to include the variables of \( H \) at lines (11) and (12). However, if \( b \) remains at \(-1\), then we could not find any choice of next subgoal, then we fail at line (13) and terminate the algorithm of Figure 12.33. The outer loop of lines (3) through (13) selects a subgoal to be the \( i \)th in the order, for each \( i = 1, 2, \ldots, m \), in turn. If all selections can be made, then the order in which the subgoals are selected, that is, they appear as \( H \) in line (11), is the desired order. If any selection fails, then there is no feasible ordering, according to Lemma 12.2.

If we analyze the running time of Figure 12.33, we get the following corollary to Theorem 12.5.

Corollary 12.1: If we use the heuristic A of Figure 12.33, then the running time of Algorithm 12.8 on rules of total length \( n \), having predicates of maximum arity \( k \), is \( O(n^{2k+3}) \).

Proof: We have only to note that \( T(n) \) is \( O(n^3) \) in Theorem 12.5, if this heuristic is used. That is, the inner loop of lines (5) through (9) scans the subgoals, taking time proportional to the length of the rule, provided we use an appropriate data structure, such as a hash table, for \( V \), so we can look up variables quickly and see if they are bound. Thus, the time taken by the inner loop is \( O(n) \). It follows that the body of the outer loop of lines (3) through (13) also takes \( O(n) \) time. Since the body is executed \( m \leq n \) times, we conclude that \( T(n) \) is \( O(n^3) \).

In particular, if there is a bound on the arity of predicates, then Corollary 12.1 says that Algorithm 12.8 with this heuristic takes polynomial time. If \( k \) is not bounded, then the running time is exponential.

12.1: We could represent persons as terms as

\[ \text{person}(N, \text{mother}(M), \text{father}(F)) \]

where \( N \) is replaced by the name of the person, \( M \) is replaced by a term representing the person who is the mother of the person \( N \) or by the constant \( unk \) (unknown), and \( F \) is likewise a term representing the father of the person, or \( unk \). Note that \( M \) and \( F \), if they are not \( unk \), are terms with principal functor \text{person}, so persons are represented by arbitrarily large terms, representing their ancestry as far back as it is known. Suppose that the database consists of a single EDB relation PERSONS, whose tuples are unary and consist of a \text{person} term as described above. Write rules to express the following relations.

a) The set of pairs \((X, Y)\) such that \( X \) is a child of \( Y \).
b) The set of pairs \((X, Y)\) such that \(Y\) is the grandmother of \(X\).

c) The set of pairs \((X, Y)\) such that \(Y\) is an ancestor of \(X\).

12.2: Write the rules for lines 1, 2, 4 of addresses, as in Example 12.1.

12.3: In Example 12.3 we pointed out that rules (5) and (6), expressing the meaning of logical OR, were not safe. One solution to the problem is to write rules for a predicate possible, such that possible(T) is true whenever \(T\) is a term whose arguments are constants found in the EDB relations \(GO\) and/or \(AC\); presumably the function symbols of \(T\) are those used in any rules we have for the predicate true, for instance, \(g, a, and, or\) in Example 12.3. Write the rules that define the predicate possible.

12.4: Another approach to the nonsafety problem is to consider the bindings provided by the query. Give an algorithm that takes a collection of rules and a binding pattern (adornment) for some predicate and determines whether the set of tuples that are inferred from a finite database and match a query with the given binding pattern is finite.

12.5: We can generalize the “blocks world” problem of Example 12.3 to a world with an arbitrary number of “blocks” as follows. Suppose we have an EDB predicate smaller(B, C) that says block \(B\) is smaller than block \(C\). Assume for convenience that smaller is transitively closed. We have a function symbol \(on(B, P)\) that says block \(B\) is on top of pile of blocks \(P\). \(P\) can be the constant symbol \(table\), so \(on(a, table)\) represents block \(a\) alone, and \(on(b, on(a, table))\) represents \(b\) on top of \(a\). We also have the function symbol \(list(P, L)\), where \(P\) is a pile of blocks on the table, and \(L\) is either a list of piles of blocks on the table (formed with the function symbol list), or the constant symbol \(nil\). Write rules for the following predicates:

a) \(pile(P)\) meaning that \(P\) is a legal pile of zero or more blocks on the table; that is, each block \(b\) is either on the table or on a pile whose top block is larger than \(b\).

b) \(disjoint(P, Q)\) meaning that \(P\) and \(Q\) are disjoint piles of blocks on the table. Hint: Remember that piles must be sorted in size order.

c) \(state(L)\) meaning that all the piles in the list \(L\) are mutually disjoint.

d) \(move(L, M)\) meaning that state (list of disjoint piles) \(L\) can become state \(M\) by one or more legal moves, where a move consists of taking a block from the top of one pile and moving it to the top of another pile, while maintaining legality; that is, the top block of the receiving pile must be larger than the block moved.

12.6: Verify that in Example 12.4, the tuple \(\text{sum}(\text{succ}^0(0), \text{succ}^0(0), \text{succ}^0(0))\)

is added to the relation for \(\text{sum}\) at round \(i + j + 2\) of naive or semi-naive evaluation.

12.7: A possible improvement to naive or semi-naive evaluation is to use the new value of an IDB relation as soon as it is computed, rather than waiting for the next round to use it. Suppose we evaluate the rules of Exercises 12.2 and 12.4 (Figure 12.1) so that the new relation for \(\text{int}\) is computed before the relation for \(\text{sum}\). Then, on what round will the sum fact of the previous exercise be added to the relation for \(\text{sum}\)? What if we compute \(\text{sum}\) before \(\text{int}\) in each round?

12.8: We wish to match the ground atom

\[ f(h(a, b), g(b, h(c))) \]

against each of the following atoms.

a) \(f(X, Y)\).

b) \(f(h(X), g(b, h(X)))\).

c) \(f(h(g(X, Y)), g(Y, Z))\).

d) \(f(g(X), h(Y))\).

For each, tell whether a matching exists and give the substitution, if so.

<table>
<thead>
<tr>
<th>(R_1)</th>
<th>(R_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(b)</td>
</tr>
<tr>
<td>(f(a))</td>
<td>(g(b))</td>
</tr>
<tr>
<td>(f(g(a)))</td>
<td>(g(f(b)))</td>
</tr>
<tr>
<td>(f(f(a)))</td>
<td>(f(a))</td>
</tr>
</tbody>
</table>

Figure 12.34 The relation \(R\).

12.9: Suppose we have the relation \(R\) whose value is the four tuples shown in Figure 12.34. Perform the following translations.

a) \(Q(X, Y) = \text{ATOM}\{p(f(X), g(Y)), R\}\).

b) \(Q(X, Y) = \text{ATOM}\{p(f(X), X), R\}\).

c) \(P = \text{VTOA}\{p(h(X), g(X)), R(X, Y)\}\).
2.10: Suppose we have the parent-ancestor rules of Example 12.16, and the relation for par is a single chain,
\[
\{(i, i+1) | 0 \leq i < n\}
\]
Describe the behavior of the following algorithms on this data, assuming the query anc(0, X), that is, find the ancestors of individual 0.
(a) Prolog's SLD resolution, as described in Section 12.3.
(b) The basic rule/goal tree expansion algorithm (Algorithm 12.6).
(c) The queue-based version of Algorithm 12.6 (QRGT).
2.11: Unify each of the atoms (a) through (d) of Exercise 12.8 with the atom \( f(h(U), U) \). Give the MGU if it exists.
2.12: Repeat Exercise 12.11 with the atom \( f(U, g(V, U)) \).
2.13: In Section 12.6 we gave rules that express certain operations: (i) passing bindings from (supplementary relations for) rules to subgoals; (ii) passing bindings from goals to (the zeroth supplementary relations of) rules; (iii) passing answers from rules to goals. Show that the rule constructions given in Section 12.6 to express these operations are each correct.
2.14: We also gave in Section 12.6 a method for passing bindings from rules to the zeroth supplementary relations of their grandchildren in the rule/goal tree directly, that is, not going through binding relations for the intermediate goal nodes. Prove that for datalog rules, the bindings obtained are the same as for the ordinary method, where we do go through goal nodes.
2.15: Example 12.25 illustrated the point that our technique of passing bindings linearly from left to right need not be as good as some ad-hoc sideways information passing strategy (SIPS). Show that for the rule of Example 12.25, reordering the subgoals and then using the standard SIPS cannot result in performance better than that of the SIPS used in that example, although the standard SIPS can approximately equal the ad-hoc SIPS if the order of subgoals is selected properly.
2.16: Find an example where an ad-hoc SIPS is significantly better than the standard SIPS, regardless of what order is chosen for the subgoals.
2.17: Modify the QRGT algorithm of Section 12.7 to pass bindings directly from rule/nodes to rule nodes, as discussed at the end of Section 12.6.
2.18: Modify the rule/goal graph construction of Section 12.8 to use passing of bindings from rule nodes to rule nodes, directly.
2.19: Construct the rule/goal graph for the goodList rules of Example 12.34, for the adorned goals
   a) \texttt{goodList}.
   b) \texttt{goodList}.

**EXERCISES**

Use the heuristic of Example 12.38 to order subgoals. That is, favor subgoals with the most bound arguments, and favor subgoals to the left in the initial ordering to break ties. This exercise may be done either (i) in an ad-hoc manner, or (ii) by Algorithm 12.8.

12.20: Construct the rule/goal graph for the following rules
   \[
   \begin{align*}
   r_1: & \quad p(x, y) \leftarrow q(x, y) \\
   r_2: & \quad p(x, y) \leftarrow p(x, z) \land p(z, w) \land p(y, w)
   \end{align*}
   \]
   and the query goal \( p^H \). Use the heuristic of Example 12.38 to order subgoals, and use either (i) an ad-hoc construction or (ii) Algorithm 12.8.

12.21: Rewrite the rules of Exercise 12.20 so that binding patterns are unique in any rule/goal tree expansion starting with query goal \( p^H \) and using the subgoal-ordering heuristic of Example 12.38.

12.22: Rectify the following rules.
   \[
   \begin{align*}
   p(x, y) & \leftarrow q(x, x, z, y) \land q(x, z, w, a) \\
   q(a, b, c, d) & \leftarrow p(a, b) \land p(c, d)
   \end{align*}
   \]

• 12.23: Prove that whether you rectify subgoals first and then split predicates to make binding patterns unique or you first split predicates and then rectify subgoals, you get the same result. Note: You must assume that there is no ordering of subgoals when you construct a rule/goal graph in the predicate-splitting process. Also, whenever you rewrite rules, you preserve the order of subgoals.

• 12.24: Prove that Algorithm 12.7 (making binding patterns unique) preserves safety; that is, if the given rules are safe, so are the constructed rules.

• 12.25: Complete the proof of Theorem 12.3 by showing that goal nodes for predicate \( p.a \) in the rule/goal tree always have the binding pattern \( a \).

• 12.26: Complete the proof of Lemma 12.2. That is, show that a greedy heuristic succeeds in finding an order for subgoals whenever one exists, given the bound-is-easier assumption.

• 12.27: Complete the proof of Theorem 12.4. That is, show that any adorned goal placed in \( F \) cannot appear in a feasible rule/goal graph.

• 12.28: Sometimes, Algorithm 12.8 can leave an adorned goal \( q^H \) in \( F \), even though in the rule/goal graph it constructs for the query goal, the node for \( q^H \) is not reachable from the node for the query goal. Give an example of this phenomenon.

• 12.29: Consider the rules of Exercise 12.22, as written. What adornments for the IDB predicates \( p \) and \( q \) are permissible if the permissible adornments for the EDB predicate \( q_0 \) are
a) Any adornment?
b) All but ffff?
c) Only those adornments with at least two 's's?

2.30: Sometimes we can express a query involving function symbols with rules that have rectified subgoals; the goodList rules of Example 12.34 are one instance. Find an example of a query that cannot be expressed with rectified subgoals, and prove your claim.

2.31: We used the set of "interesting" adorned goals, \( f \), in Algorithm 12.8 in order to give us a "top-down" algorithm, where we avoided ever having to examine the rules for an adorned goal unless there was some reason to believe that we needed to do so. Suppose we simplified Algorithm 12.8 by assuming every adorned IDP goal was interesting. What effect would that change have on the worst-case running time of the algorithm?

BIBLIOGRAPHIC NOTES

A number of relevant references appeared in the bibliographic notes of Chapter 3 (Volume 1), and some of these will be mentioned briefly here. Three general references are Lloyd [1984] on logic programming, and the surveys Gallaire, Minker, and Nicola [1984] and Minker [1987] on applications of logic to database systems.

Naive and Semi-Naive Evaluation

As mentioned in Chapter 3, the fundamentals of bottom-up (least-fixed-point, or "naive" evaluation) are found in Van Emde and Kowalski [1976], Apt and Van Emde [1982], and Chandra and Harel [1982].

Semi-naive evaluation is a fundamental idea, which keeps reappearing, as in Bayer [1985], Bancilhon [1986], Balbin and Ramakrishnan [1986], and Gonzalez-Dulio, Rochman, and Drozd [1987]. Sacca and Zaniolo [1988] extend the idea to handle rules with "stratified negation" (see Section 3.6 or the references below).

Top-Down Evaluation

The fundamental paper is Robinson [1965] on resolution theorem proving. Kowalski and Kuehner [1971] and Hill [1974] examined restricted cases of resolution, leading to the SLD-resolution (top-down, backtracking) strategy used in Prolog. A survey of these developments is in Kowalski [1988]. Also see the texts by Tanimoto [1987] and Genesereth and Nilsson [1988] for discussions of resolution theorem proving.

Convergence of top-down evaluation is considered in Afrati et al. [1986] and Ullman and Van Gelder [1986]. The matter of different SIPS (sideways information passing strategies) was examined by Van Gelder [1986b] and Beeri and Ramakrishnan [1987]; the term "SIPS" is from the latter.

The issue of "top-down safety," that is, safety of rules when we can take advantage of certain bindings at the head, as mentioned in Exercise 12.4, was explored by Zaniolo [1986], Ramakrishnan, Bancilhon, and Silberschatz [1987], and Kifer, Ramakrishnan, and Silberschatz [1988]. There are a number of safety-related references in Chapter 3.

Rule/Goal Graphs

The rule/goal graph is from Ullman [1985]. There are a number of other forms of graphs that carry less information, but have found use in logic processing, such as "connection graphs" (Kowalski [1977], Sichler [1978]). Mendelson [1985] and Debray and Warren [1986] offer other approaches to the discovery of binding patterns.

Morris [1988] gives an algorithm for constructing rule/goal graphs when reordering of subgoals is permitted and there are forbidden adornments; it is essentially Algorithm 12.8, although the simplified presentation given here, and the analysis of its running time in Theorem 12.5, are by M. Y. Vardi. Ullman and Vardi [1988] show that the feasibility of rule/goal graphs requires exponential time in general (but note it can be done in polynomial time assuming a bound on the arity of predicates, per Corollary 12.1).

Prolog and Databases

There are a large number of projects designed to apply Prolog's logic processing algorithm to database queries. These include Warren [1981], Bocca [1986], Reetz, O'Hare, and Travis [1986], Moffat and Gray [1986], Scirocco and Warren [1986], Ceri, Gottlob, and Wiederhold [1987], Ioannidis, Chen, Friedman, and Tsangaris [1988], and Naphade and Herbemer [1988]. Also see the references in Chapter 16 to the NU-Prolog system.

Expressiveness of Languages

There has been a considerable body of work on logic languages that are more expressive than the Horn-clause logic discussed in this chapter. Chandra [1988] is a survey of the general subject. Chapter 3 contains references and comments; some of the references are repeated here.

One of the central issues in extended languages is how one deals with negation. Ginsberg [1988] is a volume of papers in the field. The first feasible approach, "stratified negation," was covered in Section 3.6. The key references are Apt, Blair, and Walker [1985] and Van Gelder [1986].

More recently, treatments of negation that are more powerful (able to express more queries) have been considered. One, "inflationary semantics," ap-
plies the rules in a fixed-point calculation, but does not throw out any tuple from an IDB relation, even if newly discovered negative facts remove the justification for inferring the positive fact. This idea is from Kolaitis and Papadimitriou [1988]. Its power was explored in Kolaitis [1987] and Abiteboul and Vianu [1988].

Other recent approaches to the problem include Ross, Van Gelder, and Schlimp [1988], Van Gelder [1988], Shepherdson [1988], and Imielinski and Naqvi [1988].

Another important enhancement of logic languages is the ability to deal with set-valued variables. This direction has been explored by Beeri, Naqvi, Ramakrishnan, Shmueli, and Tsur [1987], Kuper [1987, 1986], and Shmueli and Naqvi [1987]. Also see the references to the LDL system in Chapter 16.

In Hull and Su [1988] and Paredaens and Van Gucht [1988], logics for “complex objects,” an even more general expansion of the types of variables, is explored.

CHAPTER 13
Combining Top-Down and Bottom-Up Logic Evaluation

We now explain the “magic sets” technique for rewriting logical rules so bottom-up evaluation offers all the advantages associated with top-down as well as bottom-up evaluation. In this approach, rules and a query are used to construct new rules that answer the query, although they are generally not equivalent to the original rules. Section 13.1 gives the algorithm for rewriting rules, and the proof of its correctness is found in Section 13.2. Section 13.3 shows that for datalog, the rewritten rules are guaranteed to be as efficient, when semi-naive evaluation is applied, as the original rules are when the top-down algorithm of Sections 12.5 through 12.7 (called QRTG) is used.

In Section 13.4, we see how to simplify the magic-sets rules by eliminating some predicates, and Section 13.5 briefly discusses the modifications that are needed to avoid losing bindings, as was illustrated in Example 12.23; the cost of doing so is an increased number of rules. Then, in Section 13.6 we introduce the generalized magic-sets rule-rewriting technique, which offers the benefits of both top-down and bottom-up processing for arbitrary logic programs (magic sets only makes that guarantee for datalog). However, the generalized method requires that we work with relations whose tuples may not be ground atoms, but could involve variables. The disadvantage of doing so is that joins of tuples become, in effect, unifications.

13.1 THE MAGIC-SETS RULE REWRITING TECHNIQUE

Recall our discussion of the ancestor rules

\[ r_1: \text{anc}(X,Y) \leftarrow \text{pa}(X,Y). \]
\[ r_2: \text{anc}(X,Y) \leftarrow \text{pa}(X,Z) \land \text{anc}(Z,Y). \]
COMBINING TOP-DOWN AND BOTTOM-UP LOGIC EVALUATION

18.1 THE MAGIC-SETS RULE Rewriting Technique

beginning with Example 12.16 (Section 12.5). We observed that answering the query \( \text{anc}(j, W) \), where \( j \) is a particular individual, could be performed rather efficiently top-down. That is, we could build and explore a rule/goal tree, considering only individuals who are ancestors of \( j \). In comparison, the bottom-up algorithms, Algorithms 12.3 and 12.4, would construct the entire ac relation. If most of the individuals mentioned in the \( \text{par} \) relation are not ancestors of \( j \), the bottom-up approach must take much more time than the top-down approach. However, there are a number of reasons why bottom-up calculation is preferable to top-down.

1. If we are not careful how we do top-down calculation, we can get trapped in infinite loops and never find the answer. That would be the case if we followed the simple preorder traversal approach of Algorithm 12.6 (Section 12.7) on a \( \text{par} \) relation that has cycles.

2. If we follow the \( \text{QRGT} \) version of Algorithm 12.6 discussed at the end of Section 12.7, we do not get trapped by cycles, but detecting termination, even for datalog rules, is not easy.\(^1\)

3. Even when convergence of rule/goal tree expansion is assured, we can be led to expand the same subgoal many times in the tree, thus repeating significant amounts of work.

4. The bottom-up algorithms can make use of efficient techniques for taking joins of massive relations. (Recall that join is the central step in the operation \( \text{EVAL} \), which, in turn, is the heart of semi-naive evaluation; see Section 12.2.) In comparison, the top-down approach tends to deal with many small relations, each associated with one of the nodes of the rule/goal tree. If we use \( \text{QRGT} \), where the tree is explored breadth-first, we do not even have these relations as wholes; rather we construct them little by little, in a relatively unpredictable manner.

5. Top-down algorithms require unification, but bottom-up algorithms only need term-matching, a simpler operation.

Fortunately, there is a way to get the advantages of both top-down and bottom-up methods. We rewrite a given set of rules and a query goal in such a way that the new set of rules produces the answer to the query. Moreover, if we apply the semi-naive evaluation method to the new rules, then each step we perform can be mapped into at least one step of \( \text{QRGT} \), applied to the original rules. Thus, the bottom-up evaluation of the new rules is no more expensive than the top-down evaluation of the old rules. Detection of convergence is easy for the bottom-up algorithm, when the rules are datalog. If there are function symbols, convergence after a finite number of rounds is not guaranteed, but if we do converge, we know it immediately, because we reach a round in which no more facts are added to any \( \text{IDB} \) relation.

The Unique Binding Property

The algorithm for modification of the rules is closely related to the way we organized the evaluation of rule/goal trees in Algorithm 12.6. The key idea is that we do not even consider whether a fact \( p(\mu) \) is true, for \( \text{IDB} \) predicate \( p \) and tuple \( \mu \), unless we have established that such a fact would be "interesting," in the sense that \( \mu \) matches a tuple of the binding relation for one of \( p \)'s goal nodes.

In order for the notion of a "binding pattern" for a predicate to make sense, we require our rules to have the unique binding property with respect to a given adorned goal. A logic program (set of rules) has the unique binding property if, when we construct the rule/goal graph starting with that adorned goal and respecting the order of the subgoals of rules as written, no \( \text{IDB} \) predicate appears with two different adornments. We can always obtain rules with the unique binding property; if we construct the rule/goal graph and split each \( \text{IDB} \) predicate into different predicates for the different adornments, as suggested by Algorithm 12.7.

Magic Predicates

Assuming the unique binding property, we create for each \( \text{IDB} \) predicate \( p \) a magic predicate, which we shall call \( m_p \). The arguments of the magic predicate for \( p \) correspond to those arguments that the unique binding pattern for \( p \) makes bound. For example, if \( p(\text{bf}, \text{bf}) \) is a binding pattern, then \( m_p \) is a binary relation whose tuples are bindings for the first and third arguments of \( p \).

The intent of \( m_p \) is that \( m_p(\alpha) \) should be true if and only if, in the rule/goal tree, \( \alpha \) is a binding passed to some goal node with predicate \( p \). That is not exactly right, because in the rule/goal tree there may be binding sets that cover more arguments than the supposedly "unique" binding pattern for \( p \); Example 12.32 illustrated this point. Any such additional bindings are ignored in the magic predicates, which cannot lead to incorrect behavior, but may make the evaluation of the resulting rules less efficient than they could be.

Supplementary Predicates

We also use predicates that represent the supplementary relations, which were used for sideways information passing in Chapter 12. If \( r \) is a rule with \( k \) subgoals, we shall use predicates \( \text{sup}_{r,t} \) for \( t = 0, 1, \ldots, k - 1 \). The arguments of \( \text{sup}_{r,t} \) are variables of rule \( r \), and they correspond to the arguments of the \( t \)th supplementary relation for rule \( r \), as defined in Section 12.6. That is, they

\(^{1}\) Recall that safe datalog rules are guaranteed to produce a finite answer from a finite database, thus, their evaluation surely terminates. However, knowing that an evaluation converges eventually, and detecting when convergence occurs, are quite different matters.
are all those variables of \( r \) that are either
1. bound in the head (according to the unique binding pattern for the head),
   or
2. appear in one of the first \( i \) subgoals of \( r \) and also appear in a later subgoal
   or in the head.

Note that we do not use the last supplementary relations, as we did in Algorithm
12.6, preferring to go directly from the last subgoal to the head. The difference
is not essential; we could have cut out a step in Algorithm 12.6, at the cost of
making it slightly more complicated.

As we usually refer to rules as \( r_1, \ldots, r_n \), we shall adopt the following
convention. If an enumeration of the rules is understood, then \( \text{sup}_{i,j} \) will stand
for the \( i \)th supplementary predicate for rule \( r_j \); that is, \( \text{sup}_{i,j} \) and \( \text{sup}_{j,i} \) will be
treated as synonyms. We trust that the ambiguity, which is technically present
in this convention, will not cause difficulty in practice.

The Basic Magic-Sets Transformation

We shall now give a mechanical way to convert a given set of rules and a query
into a new set of rules, with a predicate whose relation is the answer to the query.
We shall give some examples and show that the new set of rules mimics rule/goal
tree expansion. Thus, the new rules can be shown both correct, because they
produce the same answer as rule/goal tree expansion, and relatively efficient,
because they perform no more work than the latter algorithm. In Section 13.4,
we shall discuss some modifications to the technique that simplify the rules and
preserve the efficiency with which bottom-up evaluation can be performed.


**INPUT:** A collection of rules and a query goal \( g(r_1, \ldots, r_n) \). We assume that
the rules have the unique-binding property with respect to this query goal.

**OUTPUT:** A new set of rules such that the relation for \( g \) is the answer to the query.

**METHOD:** We begin by creating the following new predicates.

i) **Magic predicates for the IDB predicates.** For each IDB predicate \( p \), \( m.p \) is
   the magic predicate for \( p \). The arguments of \( m.p \) are the bound arguments
   of \( p \); recall that the binding pattern for each IDB predicate is assumed
   unique.

ii) **Supplementary predicates.** For each rule \( r_j \) with \( k \) subgoals, create sup-
   plementary predicates \( \text{sup}_{j,i} \), for \( i = 0, 1, \ldots, k - 1 \).

We now create rules for each of these new predicates, and we create new
rules for the original IDB predicates.

1. **Rules for the magic predicates.** Let \( p \) be an IDB predicate, and consider
each of the subgoals of the given rules that has predicate \( p \). Suppose \( p \)
appears in the \( i \)th subgoal of rule \( r_j \) and this subgoal is \( p(t_1, \ldots, t_m) \). Let
the arguments for the \((i-1)\)st supplementary relation be the variables
\( U_1, \ldots, U_l \). Finally, let the list of arguments of \( p \) that are bound in
the binding pattern for \( r \) be \( t_1, \ldots, t_m \). Then we create the rule
\[
\text{m.p}(t_1, \ldots, t_m) \leftarrow \text{sup}_{i-1}(U_1, \ldots, U_l).
\]

For example, suppose the subgoal in question is \( p(f(X), Y) \), and the arguments of the \((i-1)\)st supplementary predicate for rule \( r_j \) are \( X, U, \)
and \( V \). Then the first argument of \( p \) is bound, and the second is free.
Thus, \( m.p \) has only one argument, and one of the rules for \( m.p \) will be
\[
\text{m.p}(f(X)) \leftarrow \text{sup}_{i-1}(X, U, V).
\]

Notice that rules of this type implement the variable-to-argument translation
VTOS discussed in Section 12.2.

2. **Rules for the zeroth supplementary predicates.** Let rule \( r_j \) have head
\( p(t_1, \ldots, t_m) \), and suppose that \( t_1, \ldots, t_m \) are the bound arguments of \( p \).
Further, let \( X_1, \ldots, X_l \) be the variables that are bound by the head of
\( r_j \), that is, the variables appearing among \( t_1, \ldots, t_m \). Then the one rule

deфиниция \( \text{sup}_{j,0} \) is
\[
\text{sup}_{j,0}(X_1, \ldots, X_l) \leftarrow \text{m.p}(t_1, \ldots, t_m).
\]

For example, let \( r_j \) have head \( p(f(X, Y), Z) \), and suppose that only the
first argument of \( p \) is bound. Then the zeroth supplementary relation for
\( r_j \) has arguments corresponding to variables \( X \) and \( Y \), and its rule is
\[
\text{sup}_{j,0}(X, Y) \leftarrow \text{m.p}(f(X, Y)).
\]

Note that this rule implements the argument-to-variable conversion ATOV
mentioned in Section 12.2.

3. **Rules for the other supplementary predicates.** For each rule \( r_j \), having \( k \)
subgoals, and for each \( i = 1, 2, \ldots, k - 1 \), we create a single rule for \( \text{sup}_{j,i} \).
Let the variables of \( r_j \) that are attributes of the \((i-1)\)st supplementary
relation for \( r_j \) be \( U_1, \ldots, U_l \), and let the variables that are attributes of
the \( i \)th supplementary relation for \( r_j \) be \( V_1, \ldots, V_m \). Finally, let \( p(t_1, \ldots, t_m) \)
be the \( i \)th subgoal of \( r_j \). Then we have the rule
\[
\text{sup}_{j,i}(V_1, \ldots, V_m) \leftarrow \text{sup}_{i-1}(U_1, \ldots, U_l) \land p(t_1, \ldots, t_m).
\]

Note that the \( U \)‘s will always appear either among the \( U \)‘s or among the
arguments of \( p \), so the above rule is safe. For example, suppose that the
\( i \)th subgoal is \( p(f(X), Y) \), the variables bound by \( \text{sup}_{i-1} \) are \( W, X, \) and
\( Z \), while those bound by \( \text{sup}_{i,j} \) are \( W, Y, \) and \( Z \). (Apparently, \( X \) is not
needed past the $k$th subgoal, but $Y$ is.) Then the rule for $sup_j$ is

$$sup_j(V, Y, Z) : - sup_{j-1}(W, X, Z) & p(f(X), Y).$$

4. **Rules for the IDB predicates.** Let $p$ be an IDB predicate, and let $r_j$ be one of the rules for $p$. Finally, suppose $r_j$ has $k \geq 1$ subgoals. Then we obtain tuples for the predicate $p$ from the $(k-1)$st supplementary predicate and the last subgoal. That is, if $U_1, \ldots, U_k$ are the variables that are attributes of the $(k-1)$st supplementary relation for $r_j$, $h(t_1, \ldots, t_m)$ is the head of $r_j$, and the $k$th subgoal of $r_j$ is $p(v_1, \ldots, v_m)$, then we have rule

$$h(t_1, \ldots, t_m) : sup_{j-k+1}(U_1, \ldots, U_k) & p(v_1, \ldots, v_m).$$

For example, suppose $r_j$ looks like

$$h(f(X), Y) : \ldots & p(Z, g(W, Y)).$$

where there are $k$ subgoals in all, and the variables appearing in the $(k-1)$st supplementary relation are $W, X$, and $Y$. Then one of the rules for IDB predicate $h$ is

$$h(f(X), Y) : sup_{j-k+1}(W, X, Y) & p(Z, g(W, Y)).$$

Note that these rules combine two steps of the QRGT algorithm, since we do not use a predicate corresponding to the $k$th supplementary relation; rather we go directly to the head from the $(k-1)$st supplementary relation. There is an exception to rule (4) when there are zero subgoals in a rule. In that case, there is no "$(k-1)$st supplementary relation," and we construct the head from $sup_{j,0}$ alone; that is, the rule is

$$h(t_1, \ldots, t_m) : sup_{j,0}(U_1, \ldots, U_k).$$

where $U_1, \ldots, U_k$ are the arguments of the zeroth supplementary predicate for rule $r_j$.

5. **The initialization rule.** Let the query goal be $q(s_1, \ldots, s_m)$, and suppose that $t_1, \ldots, t_m$ are the bound arguments of the query; that is, $s_1, \ldots, s_m$ are ground terms. Then we have the rule

$$m.q(s_1, \ldots, s_m).$$

Intuitively, this rule says we are interested in tuples of $q$ that match the constants of the query. ☐

In examples, we shall refer to the rules of the five types listed above as Groups I through V, respectively. Note that only the Group V rule depends on the query. The other rules depend on the binding pattern of the query, although not on the particular constants involved. The rules generated by Algorithm 13.1 often will be referred to as magic-set rules or magic rules.

---

### Example 13.1

Let us consider the original "same-generation" rules,

- $r_1: \text{sg}(X, Y) : = \text{person}(X)$.  
- $r_2: \text{sg}(X, Y) : = \text{par}(X, Xp) & \text{sg}(Xp, Yp) & \text{par}(Y, Yp)$.

As in Example 12.30 (Section 12.8), with the query $\text{sg}(a, W)$; that is, we are interested in finding the cousins of individual $a$. Note we have reordered subgoals to get the preferred rule/goal graph of Figure 12.23.

The rules of the five groups are shown in Figure 13.1. For example, rule (1) appears because the second subgoal of $r_2$ is an $sg$ subgoal. At that point in the rule, only $X$ and $Xp$ are bound, so only the first argument of $sg$ is bound. In fact, $bf$ is the unique adornment associated with $sg$ in these rules, so we know the magic predicate for $sg$ will have only one argument. In rule (1), the arguments of $m.sg$ is a copy of the first argument of subgoal $sg$ in $r_2$, and the arguments of $sup_{j,1}$ are the bound variables after the first subgoal of $r_2$.

**Group I**

1. $m.sg(Xp) : = sup_{j,1}(X, Xp)$.

**Group II**

2. $sup_{j,0}(X) : = m.sg(X)$.

3. $sup_{j,0}(X) : = m.sg(X)$.

**Group III**

4. $sup_{j,1}(X, Xp) : = sup_{j,0}(X) & \text{par}(X, Xp)$.

5. $sup_{j,1}(X, Yp) : = sup_{j,0}(X, Xp) & \text{sg}(Xp, Yp)$.

**Group IV**

6. $\text{sg}(X, X) : = sup_{j,0}(X) & \text{person}(X)$.

7. $\text{sg}(X, Y) : = sup_{j,0}(X) & \text{par}(Y, Yp)$.

8. $m.sg(a)$.

**Figure 13.1** Magic-sets transformation of same-generation rules.

We can show the following facts for each tuple added to one of the IDB relations in Figure 13.1, by induction on the number of rounds of bottom-up evaluation.

1. The relations $m.sg$, $sup_{j,0}$, and $sup_{j,1}$ each represent the set of (not necessarily proper) ancestors of individual $a$.
2. Supplementary relation $sup_{2,1}$ is that subset of the EDB relation $per$ such that the first argument is an ancestor of $a$.

3. Supplementary relation $sup_{2,2}$ is the set of pairs $(X,Yp)$, such that $X$ is an ancestor of $a$ and $Yp$ is a cousin of $X$, but $Yp$ is one generation older than $X$.

4. The only tuples added to $sg$ itself are those pairs $(X,Y)$ that are cousins at the same generation, and $X$ is an ancestor of $a$.

Thus, the modified rules avoid inferring any $ag$ facts that do not, at least, have the potential to help derive facts of the form $ag(a,b)$ for some $b$. In comparison, the original rules derive facts $ag(c,d)$, where $c$ is not an ancestor of $a$. Such a fact could not be used in a proof of any fact of the form $ag(a,b)$, and thus are provably unnecessary if our goal is to answer the query $ag(a,W)$.

Note that the rules of Figure 13.1 can be simplified. For example, we can substitute $m.ag$ for both $sup_{1,0}$ and $sup_{2,0}$. We shall take up these simplifications in Section 13.4.

Example 13.2: Now, let us consider an example with function symbols, based on the "good list" rules of Example 12.34,

$$
\begin{align*}
& r_1 : \quad gl(nil) \\
& r_2 : \quad gl(cons(H,T)) \leftarrow g(H), gl(T)
\end{align*}
$$

Here, we use $g$ for good and $gl$ for goodList, to make the rules more concise. The query goal is $gl(t_0)$, where $t_0$ is a given list; it doesn't matter which. A suitable rule/goal graph for these rules is shown in Figure 13.2.

![Figure 13.2 Rule/goal graph for rules of Example 13.2.](image)

The rules generated by Algorithm 13.1 are shown in Figure 13.3. One subtlety we have not previously encountered is seen in rule (2), where the predicate $sup_{1,0}$ has no arguments. In effect, $sup_{1,0}$ denotes either true or false, and rule (2) says that its value is true if $nil$ is in the relation $m.gl$, and false otherwise. Also note that rule (5) is an example of the exception for Group IV, a rule with zero subgoals, where we obtain the head from the zeroth supplementary relation.

We can show the following by induction on the number of rounds of naive evaluation applied to the rules of Figure 13.3.

1. Relation $m.gl$ consists of those suffixes of $l_0$ that are preceded only by good elements; in particular, $nil$ is in $m.gl$ if and only if all elements of $l_0$ are good.
2. Relation $sup_{2,0}$ has tuple $(H,T)$ if and only if the list $cons(H,T)$ is a suffix of $l_0$ and is preceded only by good elements, while $sup_{2,1}$ is the subset of $sup_{2,0}$ defined by the additional condition that $H$ also be good.
3. Relation $gl$ is empty if $l_0$ does not consist exclusively of good elements. If all elements of $l_0$ are good, then $gl$ consists of all the suffixes of $l_0$.

Example 13.3: Our next example uses rules with two recursive subgoals in one body. It also illustrates an important point about how to model arithmetic.

---

2 In this case, it has no arguments because rule $r_1$ has no variables. In other situations, the rule could have variables, but none of them are bound by the head; again the magic predicate would have zero arguments.

3 Technically, we must think of "true" as meaning a relation with zero attributes, containing only the empty tuple, and of "false" as a relation with no attributes and no tuples. If we do so, then the evaluation algorithms for rules that we have been using generalize properly.
The rules in question are

$$\begin{align*}
\text{r}_1: & \ p(X,Y,D) \leftarrow a(X,Y,D). \\
\text{r}_2: & \ p(X,Y,D) :\ p(X,Z,E) \land p(Z,Y,F) \land D = E + F.
\end{align*}$$

The intent is that $a$ is an EDB predicate, and $a(X,Y,D)$ means that there is an arc from node $X$ to node $Y$, of length $D$. The IDB predicate $p(X,Y,D)$ similarly means that there is a path of length $D$ from $X$ to $Y$.

![Figure 13.4 Rule/goal graph for path-length rules.](image)

The third subgoal of $r_2$, $D = E + F$, is not of a type we have seen before, since we never incorporated arithmetic operators, like addition, into our notation for rules. However, we may think of it as an EDB subgoal $a(D,E,F)$, with the permissible binding patterns $f_{16}$ and $b_{16}$ only. The “EDB” relation $a$ consists of the infinite set of triples such that the first element is the sum of the second and third. As with built-in predicates like $<$, we can only “join” the relation for $a$ with relations for other subgoals after a finite set of values for the second and third arguments has been found, that is, after $E$ and $F$ are bound.\(^1\)

Thus, the rules may be written

$$\begin{align*}
\text{r}_1: & \ p(X,Y,D) \leftarrow a(X,Y,D). \\
\text{r}_2: & \ p(X,Y,D) :\ p(X,Z,E) \land p(Z,Y,F) \land a(D,E,F).
\end{align*}$$

Let us suppose that the query goal is $p(\overline{a})$, that is, find all the nodes reachable from a given node $x_0$ and find the length of each path to each node. A rule/goal graph for these rules and query is shown in Figure 13.4. It is not two specific rule/goal graphs, since we could have switched the order of the $r_2$. However, the $a$ subgoal can only be third, since only then

\(^1\) In principle, we could allow any two of $D$, $E$, and $F$ to be bound; for example, we could compute $E$ as $D - F$ if values for $D$ and $F$ are given. If we are willing to do the implied $f$.}

13.1: THE MAGIC-SETS RULE REWRITING TECHNIQUE

can $E$ and $F$ be bound, as required if the adornment of $a$ is to be permissible. This rule/goal graph is clearly the preferred choice, as it avoids occurrences of $a(\overline{a})$.

\begin{align*}
\text{Group I} \\
(1) & \ m.p(X) :\ sup(0)(X), \\
(2) & \ m.p(Z) :\ sup(1)(X,Z,E).
\end{align*}

\begin{align*}
\text{Group II} \\
(3) & \ sup_0(0)(X) :\ m.p(X), \\
(4) & \ sup_1(0)(X) :\ m.p(X).
\end{align*}

\begin{align*}
\text{Group III} \\
(5) & \ sup(0)(X,Z,E) :\ sup(0)(X) \land p(X,Z,E), \\
(6) & \ sup(1)(X,Z,E) :\ sup(2)(X,Z,E) \land p(Z,Y,F).
\end{align*}

\begin{align*}
\text{Group IV} \\
(7) & \ p(X,Y,D) :\ sup(0)(X) \land a(X,Y,D), \\
(8) & \ p(X,Y,D) :\ sup(2)(X,Y,E,F) \land a(D,E,F).
\end{align*}

\begin{align*}
\text{Group V} \\
(9) & \ m.p(z).
\end{align*}

![Figure 13.5 Magic-sets transformation for path-length rules.](image)

The reader should be alert to the fact that, even though the above rules are datalog, that is, there are no function symbols, we expect that the answer will be infinite if the arcs have cycles. The reason is that $a$, although treated as an EDB relation, is really infinite. When there are cycles, the bottom-up evaluation of the rules could have $r_2$ applied an infinite number of times. At each round, the relation for $p$ is finite; yet there are new length values generated at each round, and the rules never converge.\(^2\) Nevertheless, we can apply the magic-sets transformation to the above rules and get the logic program shown in Figure 13.5. If $a$ is a finite relation representing an acyclic graph, then these rules will converge bottom-up, as will the set of rules $\{r_1, r_2\}$, but if $a$ has cycles, we expect neither set of rules to converge. $\square$

\(^2\) In a sense, the addition operation plays the role of a function symbol, so the rules for path length behave as though they were not datalog.
13.2 CORRECTNESS OF THE MAGIC-SETS ALGORITHM

We need to show that the magic-sets transformation is "correct," in the sense that the new rules produce the same answer to the query as the old. The relationship between what the old rules and the new rules compute is somewhat subtle. Suppose that the query involves IDB predicate \( p \), with bindings for some of its arguments. Then the transformed rules will produce a relation for \( p \) that is generally a proper subset of what the original rules produce for \( p \). Fortunately, nothing that is lost will match the query goal, which guarantees that the answer to the query, before and after transformation, is the same. One part, that nothing is inferred by the new rules that was not inferred by the old, is isolated in the following lemma.

**Lemma 13.1**: Let \( p \) be an IDB predicate that appears in the input rules ("original rules") of Algorithm 13.1. If in the transformed rules produced by that algorithm (the "new rules"), the fact \( p(t_1, \ldots, t_n) \) can be deduced, then the same fact is deduced by the original rules.

**Proof**: In the proof of this lemma, we may ignore the magic predicates \( m.q. \). We shall prove that for each supplementary predicate \( \text{sup}_{i+1}(X_1, \ldots, X_m) \) for an original rule \( r \), each tuple \( \mu \) in the relation for \( \text{sup}_{i+1} \), \( \mu \) agrees, on common attributes, with tuples of the relationship for the first \( i \) subgoals of \( r \). More formally, \( \mu \) satisfies the following condition:

1. Let the variables \( X_1, \ldots, X_m \) be the subset of \( X_1, \ldots, X_m \) that appear among the first \( i \) subgoals of \( r \); that is, they are all the arguments of \( \text{sup}_{i+1} \), except those that appear in the head but in none of the first \( i \) subgoals.
2. Let \( \mu' \) be \( \mu \) projected onto \( X_1, \ldots, X_m \).
3. Then \( \mu' \) is in \( \text{proj}_{i+1}(Q_1 \gg \cdots \gg Q_i) \). Here, \( Q_j \) is the relation obtained by:
   i) Computing the relation for the \( j \)th subgoal of rule \( r \), according to the original rules, and then
   ii) Translating from arguments to variables, with the ATOV operator of Section 12.2.

We prove this claim, along with the claim of the lemma: tuples added to the relation for any IDB predicate \( p \) in the derived rules are also added to the relation for \( p \) in the original rules. The proof is by induction on the rounds of the bottom-up computation algorithm (Algorithm 12.3), applied to the new rules. Before beginning the induction, note that the inductive hypothesis holds vacuously for the zeroth supplementary predicates, because that hypothesis speaks of projection of the relations for those predicates onto the variables found among the first zero predicates of a rule.

**Basis**: For round 1 of bottom-up evaluation, the claim holds vacuously, because only Groups II, III, and IV can yield tuples discussed in the hypothesis. Group
evaluation on the transformed rules.

A subtlety of which we must be aware is that, in the rule/goal tree, a goal node with predicate $p$ can have more bound arguments than are indicated by the supposedly unique binding pattern of $p$. That would not be the case if the original rules are cataloged and have been processed by Algorithm 12.7 to make the bindings unique. However, in general, it is possible for some goal nodes in a rule/goal tree to have "extra" bound arguments; fortunately it is not possible that fewer arguments are bound than are expected by the binding pattern for the predicate. Also, in a way to be made precise in the next lemma, the arguments of the supplementary relations of a rule node are at least sufficient to cover all the variables of the rule that the rule/goal graph says should be bound.

**Lemma 13.2.** Suppose a set of "original" rules that satisfy the unique binding property are transformed by Algorithm 13.1 into "new" rules. Suppose we run queue-based rule/goal tree expansion (QRGT) on the original rules and Algorithm 12.3 (bottom-up evaluation) on the new rules. Then

a) If at some goal node for IDB predicate $p$, tuple $\mu$ is in the binding relation for that node, then there is some $\mu'$ in the relation for $p'$, where $\mu'$ is the projection of $\mu$ onto the arguments of $p'$ that are bound according to the unique binding pattern for $p$.

b) If tuple $\mu$ is returned as an answer at some goal node with IDB predicate $p$, then $\mu$ is in the relation for $p$ when we evaluate the new rules.

c) Suppose that at some rule node for rule $r$, say one where the substituted instance of $r$ is $\tau(r)$, tuple $\mu$ is in the $i$th supplementary relation. Let the attributes (variables) of the $i$th supplementary relation for $\tau(r)$ be $x_1, \ldots, x_n$. Let $\nu(x_1), \ldots, \nu(x_n)$ be the variables of $\tau$ (not $\tau(r)$ that are bound after $i$ subgoals of $\tau$); that is, the $X$'s are the attributes of $sup_{i-1}$ in the new rules. Then the tuple

$$\nu' = \left(\nu(\tau(x_1)), \ldots, \nu(\tau(x_n))\right)$$

is in the relation for $sup_i$.\(^6\)

**Proof:** We shall prove all three claims simultaneously, by induction on the number of steps taken by the QRGT algorithm. Before proceeding, let us note that by an easy induction on the number of steps of rule/goal tree expansion, we can infer that every goal node with some predicate $p$ has a binding pattern that

---

\(^6\) The expression for $\nu'$ takes advantage of the fact that tuple $\mu$ is really a mapping from attributes $t_1, \ldots, t_n$ to values. We have used only expressions like $\mu(t_i)$ for the value of $t_i$ in the component for the attribute $t_i$, but $\mu$, like any mapping, extends naturally to terms like $\tau(x_1)$, or even to literals and rules, provided $\mu$ has components for all the variables (attributes) mentioned in the expression to which $\mu$ is applied.

---

### 13.2 Correctness of the Magic-Sets Algorithm

bounds at least all the variables that are bound in the unique binding pattern for $p$.

**Basis:** Zero steps of rule/goal tree expansion. The only tuple given initially, is that in the binding relation for the root. The Group V rule guarantees that this tuple will be in the magic set for the query predicate, proving (a). Parts (b) and (c) hold vacuously in the basis, since no tuples fit the description before any steps of tree expansion are carried out.

**Induction:** There are three parts, corresponding to claims (a), (b), and (c) in the statement of the lemma.

(a) Suppose $\mu$ is added to the binding relation for some goal node. Then this goal node has a rule/node parent, corresponding to some substituted instance of a rule $r$, say $\tau(r)$. Let the goal in question be the $(i+1)$st subgoal of $\tau(r)$, and let that subgoal be $p(t_1, \ldots, t_k)$ in $\tau$. Thus, in $\tau(r)$, the $(i+1)$st subgoal appears as $p(r(t_1), \ldots, r(t_k))$.

Tuple $\mu$ must be derived from some tuple $\nu$ that was previously added to the $i$th supplementary relation $S_i$ at the rule node for $\tau(r)$; $\nu$ is obtained from $\nu$ by the variable-to-argument conversion operator $\nu$oa. That is, we apply the mapping $\nu$ to each of the arguments $\tau(t_1), \ldots, \tau(t_k)$ of the $(i+1)$st subgoal, where $t_1, \ldots, t_k$ are the bound arguments of $p$. Thus, we may write

$$\mu = \left(\nu(\tau(t_1)), \ldots, \nu(\tau(t_k))\right)$$

(13.1)

Now, let $x_1, \ldots, x_n$ be the variables of $r$ that are bound and relevant just before the $(i+1)$st subgoal; that is, the $X$'s are the attributes of $sup_{i-1}$. The inductive hypothesis applies to $\nu$, so by part (c) in the statement of the lemma, we conclude that

$$\nu' = \left(\nu(\tau(x_1)), \ldots, \nu(\tau(x_n))\right)$$

(13.2)

is in the relation for $sup_i$. [Note that $\nu'$ and $\nu'$ here are $\mu$ and $\mu'$ in the statement of (c)].

It is important to observe that (13.2) makes sense; in particular, $\nu(\tau(X_j))$ is defined for $j = 1, 2, \ldots, m$. The reason is that the only way $X_j$ can become bound is if it appears in a bound argument of the head or in one of the first $i$ subgoals of $r$. As we mentioned at the beginning of the proof, any argument of the head predicate $p$ that is bound in the unique binding pattern for $p$ will also be bound in the rule node for $\tau(r)$; other arguments might be bound as well. Thus, any variable that appears in $\tau(X_j)$ will be bound before the $(i+1)$st subgoal in $\tau(r)$ as well as in $\tau$. Finally, if $X_j$ appears in $r$ after the $i$th subgoal, then any variable appearing in $\tau(X_j)$ will appear in $\tau(r)$ after the $i$th subgoal, that is, if $X_j$ is relevant, so is any variable in $\tau(X_j)$. We conclude that $\tau(X_j)$ cannot involve any variables but the $U$'s mentioned in clause (c) of the lemma.
which are the bound and relevant variables after the \( i \)th subgoal.

Finally, the Group I rule defining \( m.p \) is

\[
m.p(t_1, \ldots, t_m) \supset \text{sup}(X_1, \ldots, X_m).
\]

When we substitute \( \nu' \) from (13.2) in the body of this rule, we deduce that \( \mu \), as given by (13.1), is in the relation for \( m.p \). That is, \( \nu' \) turns each \( X_j \) into \( \nu(X_j) \), so it turns the head of the Group I rule into (13.1). Thus, part (a) of the inductive hypothesis is proved.

(b) There must be some rule \( r \) for \( p \) and a rule node \( N \), with rule instance \( \tau(r) \), at which tuple \( \mu \) is produced, and that is a child of the goal node in question. Let \( r \) have \( k \) subgoals, and let the \( (k - 1) \)st supplementary relation at \( N \) be \( S_{k-1}(U_1, \ldots, U_m) \). Finally, let the \( k \)th subgoal of \( r \) be \( p(t_1, \ldots, t_j) \), and let the head of \( r \) be \( h(s_1, \ldots, s_i) \). Thus, in the substituted instance, these atoms are \( p(\tau(t_1), \ldots, \tau(t_j)) \) and \( h(\tau(s_1), \ldots, \tau(s_i)) \), respectively.

If \( \mu \) is produced in the relation for \( h \) at \( N \), there must be some tuple \( \rho \) over the variables (attributes) \( U_1, \ldots, U_m \), and whatever additional variables appear in the \( k \)th subgoal, such that

1. \( \rho(U_1, \ldots, U_m) \) is a tuple of supplementary relation \( S_{k-1} \).
2. \( \rho \) agrees with the argument-variable conversion of some tuple \( \nu \) that is in the relation for subgoal \( \tau(t_1), \ldots, \tau(t_j) \), and
3. \( \mu \) is obtained from \( \rho \) by evaluating the head predicate; that is,

\[
\mu = (\rho(\tau(s_1)), \ldots, \rho(\tau(s_i)))
\]

Condition (2) is equivalent to the statement that

\[
\nu = (\rho(\tau(t_1)), \ldots, \rho(\tau(t_j)))
\]

if we extend tuple \( \rho \), as usual, to be a mapping on terms involving the variables that are attributes of \( p \).

Now, we may apply the inductive hypothesis to the tuples known to be in \( S_{k-1} \) and to tuples returned by the goal node for the \( k \)th subgoal. By part (c) of the inductive hypothesis,

\[
(\rho(\tau(X_1)), \ldots, \rho(\tau(X_j)))
\]

is in the relation for \( \text{sup}(X_1, \ldots, X_m) \), where \( X_1, \ldots, X_m \) are the arguments for \( \text{sup}(X_1, \ldots, X_m) \).

By (b) of the inductive hypothesis and (2) above, \( \nu \) is in the relation for \( p \), if \( p \) is an EDB predicate. If \( p \) is an EDB predicate, then \( \nu \) is in its relation because that relation is the same at all goal nodes for \( p \) in the rule/goal tree and is equal to the database relation for \( p \).

In the new rules, the rule for \( h \) that is derived from \( r \) is

\[
h(s_1, \ldots, s_i) \supset \text{sup}(X_1, \ldots, X_m) \land p(t_1, \ldots, t_j).
\]

This rule's body is satisfied if we substitute \( p(\tau(X)) \) for each variable \( X \) that is among \( X_1, \ldots, X_m \), or that appears in \( p(t_1, \ldots, t_j) \). But when we make this substitution, the head becomes \( h(\mu) \), proving that \( \mu \) is in the relation for predicate \( h \).

(c) This part uses the same ideas as the first two parts, and we leave the proof as an exercise.

We can now put the previous two lemmas together and prove the correctness of the magic-set algorithms.

**Theorem 13.1:** The answer produced by the rules constructed in Algorithm 13.1 is correct.

**Proof:** Lemma 13.1 showed that every answer produced bottom-up by the new rules is also produced bottom-up by the original rules. Theorem 12.2 tells us the answers produced by QRGT and the bottom-up approach are the same. Then, Lemma 13.2(b) implies that every answer returned by the rule/goal tree expansion of the original rules is returned by the new rules, evaluated bottom-up.

13.3 EFFICIENCY OF THE MAGIC-SET RULES

We want to show that the efficiency of bottom-up execution of the new rules compares favorably with other ways to obtain the answer to the query. It is not realistic to expect that the transformed rules are as efficient as any method whatsoever on any query. However, at least for datalog rules, we can show that they are no less efficient than the QRGT algorithm applied to the original rules.

Since the latter algorithm simulates SLD-resolution (when the latter converges), and SLD-resolution is, by reason of its use in Prolog, the most successful known approach to answering general queries expressed in logic, the implication is that magic-sets is likely to be the method of choice, in general, for datalog rules. In fact, magic sets frequently improves upon SLD resolution or QRGT for nondatalog programs as well. As we shall see in Section 13.6, a generalization of magic sets, where relations can have nonground tuples, completely dominates the top-down approaches.

**Semi-Naive Evaluation**

To begin, note that we should evaluate these rules, like any rules, by the "semi-naive" evaluation method, Algorithm 12.4, rather than by the "naive" Algorithm 12.3. In the previous section, we compared QRGT to the naive algorithm. However, since Algorithms 12.3 and 12.4 produce the same answer, we could just have well shown that QRGT produces the same result as semi-naive evaluation.
A Cost Model

Let us assume that all relations $R$ are stored in such a way that, given a tuple $\mu$ over a subset of the attributes of $R$, we can find all tuples in $R$ that agree with $\mu$, in time proportional to the number of matching tuples. If there is no matching tuple, we shall assume the lookup cost is one time unit.

This assumption is not too unrealistic if we create the appropriate indices on relations. If the relations correspond to predicates of rules constructed by Algorithm 13.1, and we evaluate these rules by semi-naive evaluation, then the only times we look up a tuple are

1. When we insert it into a relation, and we wish to check that it does not already appear there.
2. When a tuple appears in $\Delta p$ for some predicate $p$, and we wish to compare it with the tuples of the (full) relation for some other predicate $q$.

Case (1) implies that we need an index on the set of all attributes; for example, we could use a hash table whose hash function maps entire tuples to buckets.$^7$

Case (2) occurs in a limited number of situations. First, note that none of the rules generated by Algorithm 13.1 have more than two subgoals, so (2) appears along with either an IDB subgoal or an EDB subgoal; each supplementary predicate appears exactly once among these rules. If it appears with an that set of its attributes that, as variables, appear in the other subgoal. Then, the second subgoal is EDB, then it never changes, so the supplementary relation the supplementary predicate needs one index, on its binding patterns. When we talk about the QRGT algorithm, we may make corresponding assumptions about indices. However, note that in QRGT the relations correspond to individual tree nodes, rather than to predicates. We shall count the same unit time to do lookups on relations in a QRGT expansion, as in a bottom-up evaluation of the predicates generated by Algorithm 13.1.

Events

When we perform semi-naive evaluation on the rules produced by Algorithm 13.1, there are two kinds of events that together account for the running time of the algorithm.

---

(A) A new tuple for some predicate $p$ is discovered; that is, the tuple is inserted into the relations for $p$ and $\Delta p$.

(B) We identify a pair of matching tuples from the relations for $\Delta p$ and $q$ for $p$ and $\Delta q$, where $p$ and $q$ are the predicates of the two subgoals of some rule in Groups III or IV.

As we shall argue below, all running time for the semi-naive evaluation algorithm can be charged to events, and all events are charged $O(1)$ time under our cost model. Thus, running time can be estimated, to within a constant factor, by counting events.

We must show that all running time is accurately accounted for. Consider first an event of type A, where we insert a tuple $\mu$ into the relation for predicate $p$. To this event we charge the cost of initiating a search for matching tuples in each rule body where predicate $p$ appears. Recall that we must charge a time unit for the lookup, even if we fail to find any matches. Each match found is charged to its own type B event. If $p$ appears as the lone subgoal of a Group I or II rule $r$, then we charge the type A event insertion-of-$\mu$-into-$p$ the cost of creating another tuple $\nu$ and checking whether it is already in the relation for the head of rule $r$, provided $\nu$ is already in that relation (and thus there is no insertion of $\nu$ to charge as a separate type A event). The number of rule bodies with predicate $p$ is a constant, depending only on the rules and not on the size of the database. Thus, the cost of a type A event is $O(1)$ under our cost model.

To an event of type B we charge the cost of inserting the tuple for the head predicate that is produced by the matching pair, in the case that this tuple already exists in the head predicate (in which case there is no type A event to charge for the insertion). We also charge unit time for consideration of the pair itself. Thus, the cost of a type B event is $O(1)$.

When we perform QRGT, we similarly can partition the work into events of types A and B. Type A events are those in which a tuple is added to

1. A binding relation at a goal node with an IDB predicate,
2. The result relation of a goal node with an IDB predicate, or
3. A supplementary relation at a rule node.

We do not count as an event the addition of a tuple to the result relation for a rule node; that tuple is immediately added to the result of the goal-node parent, and the cost can be charged to the latter type A event.$^8$

Type B events are those where tuples in supplementary relation $S$, at some rule node are matched with tuples in the result relation for the $(i+1)$st goal-node child of $N$.

Not all the costs of QRGT are accounted for by the above scheme; for

---

$^7$ See Chapter 6 of Volume I for a description of storage structures for relations.

$^8$ In an efficient implementation, we would not even create result relations for the rule nodes, but rather accumulate their tuples at the goal node above.
example, we ignore the creation of the tree itself. Also, we do not count the
processing of bindings to EDB relations as events in QRTG. However, the number
of events is a lower bound on the cost of QRTG, which is all that we shall need
in what follows.

Subgoal-Rectified Datalog

We can show that semi-naive evaluation of the rules transformed by Algorithm
13.1 executes no more events than does QRTG, in one very significant case.
Recall rules are said to be subgoal-rectified if all subgoals have lists of distinct
variables for their arguments; of course, different subgoals may use the same
variable. The output of Algorithm 12.7 is subgoal-rectified, so we know that
every set of Datalog rules can be put in this form.

The importance of the subgoal-rectified property is that in QRTG, every
instance of rule r may be taken to be r itself; that is, all unifications are trivial.
The following lemma is the heart of the proof that the transformed rules are
evaluated bottom-up at least as efficiently as QRTG evaluates them. We must
later show that Algorithm 12.7, the step that makes the rules subgoal-rectified
and gives them the unique-binding property, does not slow down QRTG. If
the transformation of Algorithm 12.7 produced rules that could not be evaluated
efficiently by any means, then our comparison would be suspect.

Lemma 13.3: Suppose we are given a set of “original” rules that are subgoal-
rectified and have the unique-binding property, and that Algorithm 13.1 pro-
duces from these, and from a certain query goal, a set of “new” rules. Then
for each event E of type A or B that occurs when we evaluate the new rules by
semi-naive evaluation, we can identify a corresponding event F at some node
of the QRTG expansion of the query goal according to the original rules, as
follows.

a) If E is the insertion of a tuple μ into the relation for some EDB predicate
p, then F is an insertion of μ into the relation returned by some goal node
with predicate p.

b) If E is the insertion of μ into a magic predicate m,p, then F is an insertion
of μ into the binding relation for some goal node with predicate p.

c) If E is the insertion of tuple μ into a supplementary predicate supb, then
F is the insertion of μ into the ith supplementary relation at some goal
node for r.

d) If E pairs a tuple μ in the relation for some supk with a matching tuple ν
in the relation for the (i + 1)st subgoal of r, then F is the pairing of the
result relation for the (i + 1)st goal-node child of N for r, with μ in
matching in the bodies of rules of Groups III and IV.

Proof: The proof is an induction on the order in which events occur during
semi-naive evaluation. In one round, all of the type B events, which pair tuples
in the bodies of the rules, occur before all of the type A events, which construct
new tuples from the bodies and insert them into the relations for the heads of
rules. We shall not give the entire induction, most of which is straightforward.
Rather, we concentrate on the most difficult situation, where we have a type
B event in which tuples from the body of a Group III or IV rule are paired.
We show that a corresponding type B event occurs during QRTG. Then we
prove that the resulting type A event, where a tuple is added to the next
supplementary relation (by a Group III rule) or to an EDB predicate (by a
Group IV rule), also corresponds to an event during QRTG.

Thus, suppose we have a rule body

\[ \text{sup}_k(X_1, \ldots, X_m) \cup p(Y_1, \ldots, Y_k). \]  

(13.3)

where the Y's are distinct variables, but there may be some overlap between
the X's and the Y's. Let the type B event consist of the pairing of μ in
the relation for \text{sup}_k with a matching tuple ν from the relation for p. Presumably,
at least one of μ and ν was added to its relation on the previous round, and
the other was added on a prior round as well (unless p is an EDB predicate);
thus, the inductive hypothesis applies to each, although it is vacuous for an
EDB predicate.

By the inductive hypothesis, there is some rule node N in the rule/goal
tree such that the ith supplementary relation, \( s_i(X_1, \ldots, X_m) \), at N contains μ.
Note that we may assume, by the subgoal-rectified property, that the instance
of r at N is μ itself. Thus, the attributes of the supplementary relations at N
are the same variables as the corresponding arguments of the predicates \text{sup}_j,
for each j. Also, the bound arguments of the EDB subgoals of N are the same as
the unique binding pattern for the predicates of those subgoals.

If p is an EDB subgoal, we need to show that ν agrees with some tuple ρ
in the binding relation for the (i + 1)st subgoal, \( p(Y_1, \ldots, Y_k) \), at node N.
For then we know, by Theorem 12.2, that ν will be placed eventually in the
relation returned by the (i + 1)st goal-node child of N, which we shall call node
M. To that end, let \( Z_1, \ldots, Z_n \) be the variables found among \( X_1, \ldots, X_m \)
and also found among \( Y_1, \ldots, Y_k \). Then there is a Group I rule

\[ m.p(Z_1, \ldots, Z_n) \cup \text{sup}_i(X_1, \ldots, X_m). \]

Since μ satisfies the body, the projection of μ onto \( Z_1, \ldots, Z_n \), which we shall
call ρ, satisfies the head.

By the same reasoning, ρ is placed in the binding relation for M. Thus,
by Theorem 12.2, ρ eventually enters the relation returned at M, and a type B
event, in which ρ is paired with matching tuple ν at node N, occurs.

If p is an EDB subgoal, there is nothing to prove. That is, ρ is always in the
COMBINING TOP-DOWN AND BOTTOM-UP LOGIC EVALUATION

relation for \( p \). We have thus shown that each type B event during semi-naive evaluation corresponds to a type B event during QRG.

Now suppose that (13.3) is the body of a Group III rule, so the head is \( H \in \text{succ}(U_1, \ldots, U_n) \), where the \( U_i \)'s are all found among the \( X \)'s or the \( Y \)'s. Then the tuple \( \psi \), which has components for each of the \( U_i \)'s, and agrees with both \( \mu \) and \( \nu \) on common attributes, is added to the relation for \( \text{sup}_{i+1} \). By the same reasoning, \( \psi \) is added to the \( (i+1) \)st supplementary relation at \( N \) as well. In terms of the inductive hypothesis, whenever we find a type A event, in which a tuple \( \psi \) is added to the relation for a supplementary predicate, such as \( \text{sup}_{i+1} \), we can identify a pair of matching tuples \( \mu \) and \( \nu \) that led to that event, and by the argument just pursued, \( \mu \) and \( \nu \) are also paired at some rule node \( N \), during QRG expansion. Thus, the matching type A event for \( \psi \) occurs at \( N \) as well.

A similar argument shows that if (13.3) is a Group IV rule, the type A event in which a tuple is added to some IDB predicate during semi-naive evaluation, is mimicked, during QRG expansion, by an insertion of the same tuple into the result relation at some rule node \( N \), and then into the result relation for its goal-node parent. The remaining inductive cases are left for the reader.

Note that QRG does some extra steps not reflected in the magic-sets rules. For example, QRG constructs final supplementary relations for rules and then computes result relations from these, rather than going directly from the head of the previous supplementary rule to the result rule. QRG computes binding relations for EDB subgoals, while magic sets avoids this step, performing lookups directly from the supplementary relations, when the joins implicit in rules of Groups III and IV are executed. None of these differences are essential. We explained QRG as we did for simplicity, but we could have added the “improvements” found in Algorithm 13.1 to QRG as well.

**Theorem 13.2:** Suppose we are given “original” rules, that are subgoal-rectified datalog, and that satisfy the unique-binding property. Let these rules be transformed into “new” rules by Algorithm 13.1. Then the running time of semi-naive evaluation on the new rules is at most proportional to the running time of QRG on the original rules.

**Proof:** It suffices to observe that two events during semi-naive evaluation are never associated, by Lemma 13.3, with the same event during QRG expansion. The reason is that each event of type A involves a unique tuple and a particular pair of predicates; each type B event involves a unique pair of tuples and a particular pair of predicates.

While Theorem 13.2 only implies that semi-naive evaluation of the new rules cannot be more than a constant factor slower than QRG expansion of the original rules, in fact the smaller constant of proportionality probably goes with the semi-naive evaluation. The latter algorithm can take advantage of economies of scale due to the fact that it deals with a few large relations rather than many small ones. Moreover, in many cases, the number of events during QRG will far exceed the number during semi-naive evaluation, since QRG may repeat the same work at many different nodes.

There is, however, another concern. Suppose we start with a datalog program \( P_1 \) and apply Algorithm 12.7 to develop an equivalent datalog program \( P_2 \) that is subgoal-rectified and has the unique-binding property. We then use Algorithm 13.1 to transform \( P_2 \) and query goal into a “magic” program \( P_3 \). Theorem 13.2 tells us that \( P_3 \) compares favorably with \( P_2 \), but does \( P_2 \) therefore compare well with \( P_1 \)? The answer is “yes,” as the following corollary to Theorem 13.2 explains.

**Corollary 13.1:** The running time of semi-naive evaluation on datalog program \( P_2 \), above, is at most proportional to the running time of QRG on \( P_1 \).

**Proof:** Consider QRG expansion, from the same query goal, applied to \( P_1 \) and \( P_2 \). We need to show that the sequences of events during the two expansions take, to within a constant factor, the same amount of time.

The first transformation of Algorithm 12.7 reduces the number of arguments of subgoals, say by replacing a subgoal \( p(X, X) \) by \( p'(X) \), for some new predicate \( p' \). This transformation changes the names of the predicates that appear, in corresponding steps of \( P_1 \) and \( P_2 \). It also changes the rules used, since the rules for \( p' \) are the rules for \( p \) with the two arguments of the head identified and merged. However, it is a simple induction on the number of steps performed by QRG, that the behaviors of \( P_1 \) and \( P_2 \) are essentially the same. The only difference is that in the expansion of \( P_1 \), we may find a relation with tuples that have two components identical, or set equal to a constant, while in \( P_2 \), only one copy of the identical columns appears, and the constant column will not appear at all.

The second transformation of Algorithm 12.7 simply renames predicates to guarantee unique bindings; it has no effect on the running time. Thus, \( P_2 \) is no slower than \( P_1 \).

**Limitations of the Magic-Set Transformation**

There are some logic programs and query goals, not covered by Theorem 13.2, for which Algorithm 13.1 transforms the rules into a set that are as efficient, or more so, than the originals. For instance, the “goodList” rules of Example 13.2 are not subgoal-rectified datalog, yet the conclusion of Theorem 13.2 holds for these rules as well. However, there are datalog programs without the subgoal-rectified property, for which semi-naive evaluation of the “magic” rules (not

---

6 Note that Algorithm 12.7 does no reordering of subgoals. If we were able to reorder subgoals while constructing \( P_2 \), then \( P_2 \) could run faster or slower than \( P_1 \). However, reordering of subgoals presumably improves the speed on the average.
proceeded by the rectification algorithm) is much slower than QRGT on the
original rules. There are also examples of non-datalog rules where, during
QRGT, the binding pattern of every goal node for an IDB predicate \( p \) is the
same, and equals the binding pattern of \( p \) in the rule/goal graph. Yet again,
semi-naive evaluation of the "magic" rules is less efficient than QRGT on the
original rules. We illustrate these points in the next two examples.

**Example 13.4:** Consider the rules

1. \( r_1 : p(X,Y) : a(X,Y) \)
2. \( r_2 : p(X,Y) : b(X,Y) \land p(X,Z) \)

These rules are similar to the rules of Example 12.32, and their rule/goal graph,
respecting the given order of subgoals, is shown in Figure 13.6. Note that \( f/b \)
is the unique binding pattern for \( p \) if we start with a query of that binding
pattern.

Let us suppose that the EDB consists of the two tuples \( \{ (1, 2, 3), (3, 4, 5) \} \)
in the relation for \( b \), and a large number of tuples of the form \( \{(m, n, 5)\} \)
where \( m \) and \( n \) are not both 4, in the relation for \( a \). Further, let the query goal
be \( p(X,Y) \). Figure 13.7 shows the rule/goal tree expanded out as far as it
goes with this query goal. For convenience, we have replaced variables that are
bound to singletons by those singletons. For example, the goal node \( p(X, X, Z) \)
is really \( p(X, X, Z) \), with \( Z \) bound to \( \{3\} \).

Evidently, each of the \( a \) goal nodes in Figure 13.7 fails to find any matches in the
EDB relation for \( a \). That is, goals \( a(X, Y) \) and \( a(X, Z) \) match nothing,
because the last component is 5 in all tuples for \( a \), and \( a(4, 4, 5) \) matches nothing
because we ruled out the possibility that the first two components of an \( a \) tuple
are both 4. More importantly, the subgoal \( b(5, 4, Z) \) in the lower-right corner
finds no matches in the EDB relation for \( b \). Thus, \( b \) is passed sideways as the
binding for \( Z \), whereas the subgoal \( p(4, 4, Z) \), shown as \( p(4, 4, b) \), is known
to return the empty set, and the rule/goal tree exploration ceases.

Now, let us consider the magic rules constructed by Algorithm 13.1, which
are shown in Figure 13.8. With the same EDB, the relations for predicates \( m \),
\( s \), and \( u \), each contain the unary tuples \( \{1, 3, 5\} \), and \( 5 \) is the only
tuple with a disjunction \( \lor \) in \( s \). Then, by rule (3), all the tuples in the
relation for \( a \) are added to the relation for \( p \).

Since the number of tuples for \( a \) of the form \( \{(m, n, 5)\} \) can be as large as we like,
we see that the time to evaluate the magic rules is arbitrarily greater than the
time to evaluate the original rules by the QRGT algorithm.

**Example 13.5:** The next example bears substantial resemblance to Example
13.4, but here we exploit function symbols to cause the rule/goal tree to termi-
nate with no answer tuples discovered, while the magic rules produce many
irrelevant tuples. The significant property that these rules have, but the rules
of Example 13.4 do not, is that here, the binding pattern at each IDB goal node

\[ r_1 : p(X,Y) : a(X,Y) \]
\[ r_2 : p(X,Y) : b(X,Z) \land p(Z,f(Y)) \]

Figure 13.6 Rule/goal graph for rules of Example 13.4.

Figure 13.7 Rule/goal tree for rules of Example 13.4.

in the rule/goal tree is the same, \( p/b \) in particular. (Note that in Example 13.4
we saw the binding pattern \( b/b \) as well as \( f/b \).) The rules are

\[ r_1 : p(X,Y) : a(X,Y) \]
\[ r_2 : p(X,Y) : b(X,Z) \land p(Z,f(Y)) \]
and their rule/goal graph is shown in Figure 13.9.

Let us suppose that the EDB relation for $b$ is $\{(1,2)\}$, while the relation for $a$ consists of many tuples of the form $(2, n)$. Here, we assume $n$ is an integer; in particular, $n$ does not match any term of the form $f(X)$ for any $X$. The rule/goal tree for query goal $p(1, Y)$ is shown in Figure 13.10. As in Figure 13.7, we show goals with arguments bounded to singletons as if that argument were the element of the singleton set, and we show the argument bound to the empty set as $\emptyset$. The rule/goal tree terminates as shown, because the goal $b(2, Z_2)$ returns the empty set for $Z_2$, which tells us that the goal $p(Z_2, f(f(Y)))$ cannot produce any tuples. Note especially that goal node $a(2, f(Y))$ does not match any of the tuples $(2, n)$ in the relation for $a$, because no constant $n$ can match.
13.4 SIMPLIFICATION OF MAGIC-SET RULES

We can eliminate any IDB predicate \( p \) that does not appear as a subgoal of one of its own rules by the steps shown in Figure 13.12. Informally, we substitute, for each subgoal with predicate \( p \), the body of each of the rules for \( p \), after unifying the rule head with the subgoal.

for each occurrence of \( p \) in a subgoal \( G \) of some rule \( s \) do begin
  for each rule \( r \) with head predicate \( p \) do begin
    rename the variables of \( r \) so \( r \) shares
    no variable with \( s \);
    let \( \tau \) be the most general unifier of the
    head of \( r \) and the subgoal \( G \);
    create a new rule \( s' \), by taking \( \tau(s) \) and replacing
    the subgoal \( \tau(G) \) by the body of \( \tau(r) \);
  end;
delete rule \( s' \);
delete the rules with \( p \) at the head

Figure 13.12 Elimination of predicate \( p \).

Example 13.6: Suppose \( p \) has two rules
\[
\begin{align*}
\tau_1: & \quad p(f(X), Y) :- t(X, Y). \\
\tau_2: & \quad p(X, a) :- u(X) \land v(X).
\end{align*}
\]
Also suppose for simplicity that there is only one use of \( p \), that in the rule
\[
\begin{align*}
\tau_3: & \quad q(X, Y) :- p(X, Z) \land s(Z, Y).
\end{align*}
\]
We begin by rewriting \( \tau_1 \) and \( \tau_2 \) so they share no variables with \( \tau_3 \); they become
\[
\begin{align*}
\tau_1: & \quad p(f(U), Y) :- t(U, Y). \\
\tau_2: & \quad p(U, a) :- u(U) \land v(U).
\end{align*}
\]
Now, we unify the head of \( \tau_1 \) with the subgoal \( p(X, Z) \), which gives us the MGU \( \tau(X) = f(U), \tau(Z) = Y, \tau(Y) = V \), and \( \tau(U) = U \). For \( Y \), which is not involved in the unification, we must assume \( \tau(Y) = Y \). Then \( \tau_3 \) becomes
\[
\begin{align*}
\tau(\tau_3): & \quad q(f(U), Y) :- p(f(U), V) \land s(V, Y).
\end{align*}
\]
and \( \tau(\tau_1) \) is
\[
\begin{align*}
p(f(U), V) :- t(U, V).
\end{align*}
\]
When we substitute the body of \( \tau(\tau_1) \) for the subgoal \( p(f(U), V) \) in \( \tau(\tau_3) \), we get the new rule
\[
\begin{align*}
r_4: & \quad q(f(U), Y) :- t(U, V) \land s(V, Y).
\end{align*}
\]
One might wonder whether \( r_4 \) is less general than \( r_3 \), since \( r_4 \) only applies if the first argument of \( q \) is of the form \( f(U) \) for some \( U \), while \( r_3 \) has no such restriction. Indeed, \( r_4 \) is less general. However, \( r_4 \) reflects the condition that must hold if \( r_3 \) is applied to some goal, and then \( r_1 \) is applied to the \( p \) subgoal of \( r_3 \). For \( q \)-goals that do not match the head of \( r_4 \), the only way to expand them by QRT (using the original rules) is for \( r_3 \) to be followed by \( r_2 \), a case that we consider next.

When we unify the head of \( r_2 \) with the \( p \)-subgoal of \( r_3 \), we get the MGU \( p(X) = p(U) = U \), and \( p(Z) = a \). We may extend \( p \) to \( Y \) by taking \( p(Y) = Y \). Then, substituting the body of \( r(p_2) \) for the \( p \)-subgoal in \( r(p_3) \), we get
\[
\begin{align*}
r_5: & \quad q(U, Y) :- u(U) \land v(U) \land s(a, Y).
\end{align*}
\]
Finally, we discard \( r_1, r_2, r_3 \), and \( r_4 \), leaving the equivalent pair of rules, \( r_4 \) and \( r_5 \).

Theorem 13.3: The transformation of Figure 13.12 does not change the relation computed for any of the predicates that remain after transformation.

Proof: We shall leave the details of this proof as an exercise. The key idea is to look at the rule/goal trees produced for some query, using both the old rules and the new. We may transform the rule/goal tree for the old rules into one using the new rules, as follows. In the tree for the old rules, each time we have a goal node \( N \) with predicate \( p \), we combine \( N \)'s rule-node children with \( N \)'s rule-node parent; that is, we combine two levels of rule nodes into one, making the appropriate substitution indicated by the algorithm in Figure 13.12.\(^{10}\) The effect is that in the revised tree, rules from the new set have replaced the corresponding rules from the old set, as suggested in Figure 13.13. If we make this change everywhere a goal node with predicate \( p \) appears, we shall create a tree using the new rules, which produces the same relation at each of its nodes as was produced at the corresponding node of the old tree.

\( ^{10} \) Note that \( p \) is assumed not to appear as a subgoal in its own rules.
Predicate Elimination Applied to the Magic-Set Rules

Ordinarily, the transformation of Figure 13.12 causes extra work; there is an increase in the number of rules, and the rules get longer. However, in two situations it makes sense to eliminate a predicate p. As before, we assume that p does not appear as a subgoal of its own rules. It is then advantageous to eliminate p whenever

1. p has only one rule, and p appears in only one subgoal, or
2. p has only one rule, and its body has only one subgoal; p may appear as a subgoal any number of times.

Let us consider when these conditions apply to the predicates and rules generated by the magic-sets algorithm. First, each zeroth supplementary predicate has only one rule, in Group II, and its body has length one. Thus, case (1) always applies to a zeroth supplementary predicate, and we can eliminate them by substituting the appropriate magic predicate for the zeroth supplementary predicate in a rule of Group III or IV.

The other supplementary predicates also have only one rule, but these rules, in Group III, have bodies of length two. Thus, only (2) could apply. If the jth subgoal of rule r_j is an EDB subgoal with predicate p, then there will be two uses of sup_j,-1: once in a rule of Group I for the magic predicate m,p, and the other in Group III or IV, when sup_j,-1 is used to compute either sup_j, or the head of r_j. If the jth subgoal is EDB, then the use of sup_j,-1 in Group I is missing. Thus, we can eliminate each supplementary predicate that corresponds to a point just before an EDB subgoal.

There are often other opportunities to apply (1) or (2). For example, a magic predicate may have only one rule (don’t forget to count the Group V rule, however). All rules for magic predicates in Group I have bodies consisting of a single subgoal with a supplementary predicate, so (2) applies if the magic predicate has a single rule.

**Example 13.7:** Let us consider the magic rules of Figure 13.1 (Section 13.1) for the same-generation rules. These rules are repeated in Figure 13.14, for convenience.

1. m.sg(X) :- sup_1,1(X,Xp).
2. sup_1,0(X) :- m.sg(X).
3. sup_2,0(X) :- m.sg(X).
4. sup_1,1(X,Xp) :- sup_2,0(X) & par(X,Xp).
5. sup_2,1(X,Yp) :- sup_1,1(X,Xp) & ag(X,Yp).
6. sg(X,X) :- sup_2,1(X) & person(X).
7. sg(X,Y) :- sup_2,2(X,Yp) & par(Y,Yp).
8. m.sg(a).

**Figure 13.14** Magic-sets transformation of same-generation rules.

We begin by eliminating the zeroth supplementary predicates, which are defined by rules (2) and (3). We substitute m.sg(X) for sup_1,0(X) and sup_2,0(X) in rules (4) and (6), so the latter become:

4. sup_1,1(X,Xp) :- m.sg(X) & par(X,Xp).
5. m.sg(X,X) :- m.sg(X) & person(X).

Rules (2) and (3) can then be eliminated.

Notice that in Figure 13.14, all occurrences of a predicate have exactly the same arguments, so it appears as if no renaming of variables, as required in the algorithm of Figure 13.12, has been done when we constructed the new rules (4) and (6). However, strictly speaking, the new rules (4) and (6) were derived by following the algorithm of Figure 13.12 exactly.

We may eliminate the predicate sup_2,2, which is defined by rule (5), if we substitute the body of (5) in rule (7). The remaining five rules are shown in Figure 13.15. Note that further applications of the algorithm of Figure 13.12 are legal, but would make the program slower. For example, we could eliminate sup_1,1 if we substituted the body of rule (4) in rules (1) and the rewritten (7).

We would now interrogate the EDB relation for par twice for each value of X that appears in the magic predicate m.sg(X). □

**Example 13.8:** A similar sequence of transformations applies to the “good list” rules of Example 13.2 (See Figure 13.3). The resulting set of five rules is shown in Figure 13.16.

We may also perform a similar sequence of steps on the “paths” rules of Example 13.3. However, we should note that rules (1) and (6) define the predi-
(8) \( m\text{.sg}(a) \).
(1) \( m\text{.sg}(Xp) := \text{sup2}_1(X,Xp) \).
(4) \( \text{sup2}_1(X,Xp) := m\text{.sg}(X) \& \text{par}(X,Xp) \).
(6) \( \text{sg}(X,X) := m\text{.sg}(X) \& \text{person}(X) \).
(7) \( \text{sg}(X,Y) := \text{sup2}_1(X,Xp) \& \text{sg}(Xp,Yp) \& \text{par}(Y,Yp) \).

Figure 13.15 Improved magic rules for same-generation.

(7) \( m\text{.gl}(h) \).
(1) \( m\text{.gl}(T) := \text{sup2}_1(H,T) \).
(4) \( \text{sup2}_1(H,T) := m\text{.gl}(\text{cons}(H,T)) \& g(H) \).
(5) \( \text{gl}(\text{nil}) := m\text{.gl}(\text{nil}) \).
(6) \( \text{gl}(\text{cons}(H,T)) := \text{sup2}_1(H,T) \& \text{gl}(T) \).

Figure 13.16 Simplified "good list" rules.

In cases \( m.p \) and \( \text{sup2}_0 \) to be identical. When we eliminate \( \text{sup2}_0 \) by substituting the body of rule (6) into rule (1), the latter rule becomes
\[
m.p(X) := m.p(X).
\]

Such a rule evidently serves no purpose, and we may eliminate it. The final set of rules is shown in Figure 13.17.

(9) \( m.p(x_0) \).
(2) \( m.p(Z) := \text{sup2}_1(X,Z,E) \).
(5) \( \text{sup2}_{1}(X,Z,E) := m.p(X) \& p(X,Z,E) \).
(7) \( p(X,Y,D) := m.p(X) \& a(X,Y,D) \).
(8) \( p(X,Y,D) := \text{sup2}_1(X,Z,E) \& p(Z,Y,F) \& s(D,E,F) \).

Figure 13.17 Simplified "paths" rules.

Direct Generation of the Simplified Rules

The following useful observation can be made about the transformations discussed above. Suppose we have a rule \( r \) with subgoals \( G_1, \ldots, G_j \). Let \( G_{i+1}, \ldots, G_j \) be EDB subgoals; \( G_i \) may be either an EDB or an IDB sub-

13.5 PASSING BINDINGS THROUGH VARIABLES ONLY

Recall from our discussion in Section 12.6, especially Example 12.23, that when there are function symbols, we can lose information by converting supplementary predicates to magic predicates and then back to zeroth supplementary predicates of other rules. The motivation for doing so anyway is that fewer rules are generated if we follow Algorithm 13.1, than if we avoid using the magic predicates. Let us briefly discuss the modifications to Algorithm 13.1 that are needed if we go directly between supplementary predicates, rather than using the magic predicates.

First, we must revise the notion of a rule/goal graph to avoid the IDB goal nodes altogether. If a rule \( r \)'s ith subgoal has an IDB predicate \( p \), then the rule node \( r_{i-1} \) has successor \( r_i \) as usual, but also has a successor \( s_0 \) for each rule \( s \) whose head predicate is \( p \). Successor \( s_0 \) has whatever bound variables can be deduced by binding the variables that \( r_{i-1} \) says are bound, unifying the ith subgoal of \( r \) with the head of \( s \), and making bound for \( s_0 \) any variable that unifies with a term all of whose variables are bound by \( r_{i-1} \). The details were covered in Section 12.6.

There is one exception to the avoidance of goal nodes for the IDB predicates. If \( p \) is the query predicate, then we need a goal node \( p^* \), where \( a \) is the appropriate adornment. This adornment determines the bound variables for
the rule nodes \( s_9 \) that are added to the rule/goal graph initially; here, \( s \) ranges over all rules with head predicate \( p \). Nodes representing other sets of bound variables, for these or other rules, are added to the rule/goal graph on demand, as in the usual construction described in Section 12.8.

Now, when we generate the magic rules, we define \( \text{sup}_{p,0} \) directly in terms of \( \text{sup}_{p,1-1} \). The necessary rule is constructed thusly. After renaming variables of \( s \), if necessary, unify the \( i \)th subgoal of \( r \), say \( p(t_1, \ldots, t_k) \), with the head of rule \( s \), say \( p(u_1, \ldots, u_k) \), and let \( \tau \) be the MGU. Then determine the bound variables for the head as described at the end of Section 12.6. That is, we know

1. The variables \( X_1, \ldots, X_n \) that are bound by (are arguments of) \( \text{sup}_{p,1-1} \) are bound.
2. If \( \tau(Y) \) contains \( X \), and \( Y \) is bound, then \( X \) is bound.
3. If all variables appearing in \( \tau(X) \) are bound, then \( X \) is bound.

Let \( Y_1, \ldots, Y_m \) be the variables of the head of \( s \) that are determined to be bound if we apply this process until all bound variables are found. Then we generate the rule

\[
\text{sup}_{p,0}(\tau(Y_1), \ldots, \tau(Y_m)) := \text{sup}_{p,1-1}(\tau(X_1), \ldots, \tau(X_n)).
\]

Finally, we must replace the Group V rule that initializes the magic set for the query predicate, since this magic predicate will no longer be present. If the query predicate is \( p \), we term-match the head of each rule \( r \) for \( p \) with the query. Thus, we obtain a tuple \( \mu \) of bindings for the variables that are arguments of \( \text{sup}_{p,0} \), and we generate the bodyless rule

\[
\text{sup}_{p,0}(\mu).
\]

for each rule \( r \).

**Example 13.9:** Let us reconsider the same-generation rules of Example 13.1, with the query goal \( s^M \). As these are datalog rules, the passing of bindings is no different from what was discovered in Example 13.30 (Section 12.8). However, the rule/goal graph, shown in Figure 13.18, differs from the "standard" rule/goal graph, which was given in Figure 12.23, in that the successors of rule node \( r_2 \) are now rule nodes \( r_{1,0} \) and \( r_{2,0} \), instead of the goal node \( s^M \); \( r_2 \) remains a successor of \( r_{2,1} \).

When we generate the magic-set rules the Group I rules are eliminated. The Group II rules define the zeroth supplementary predicates in terms of other supplementary predicates, rather than magic predicates. Groups III and IV are unchanged, and Group V now has one basis rule for each of the \( s_9 \) rules. These rules are shown in Figure 13.19. □

**Example 13.10:** Now, let us consider part of a more complicated example based on Example 12.24. Let the \( i \)th subgoal of rule \( r \) be

![Figure 13.18 New form of rule/goal graph.](image)

**Figure 13.19 Magic-set rules with bindings passed from rules to rules.**

\[
p(U, V, Y)
\]

and let the head of rule \( s \) be

\[
p(f(U, V), Y)
\]

As in Example 12.24, let us suppose that \( Y \) is an argument of \( \text{sup}_{p,1-1} \), and \( X \) is not. In the MGU, from Examples 12.13 and 12.24, we have \( \tau(X) = \tau(Y) = g(U) \), \( \tau(U) = U \), and \( \tau(V) = g(U) \). Thus, the rule defining the bindings of
U and V in terms of the binding for Y is
\[
\text{sup}_{r,\circ} (U, g(V)) = \text{sup}_{r,\circ} (g(U)).
\]
(13.5)

Note that even though V is not mentioned in (13.5), it receives a value, which is given by the second argument of \(\text{sup}_{r,\circ} \). Intuitively, (13.5) says that for each value assumed by Y, if it is not of the form \(g(t)\), then unification of the head of s with the rth subgoal of r yields no tuple for the head. If Y is \(g(t)\) for some ground term t, then U becomes t, and V becomes \(g(t)\).

Let us again observe that this modification to Algorithm 13.1 offers the advantage that it may, with rules that have function symbols, provide some additional bound expressions beyond what Algorithm 13.1 itself can provide. The disadvantage is that the modified method can produce approximately the sum of the number of rules produced by Algorithm 13.1 itself. That is, for each subgoal, say the rth subgoal of rule r, Algorithm 13.1 produces one Group I rule that defines the magic predicate for p (the predicate of r's rth subgoal) in terms of \(\text{sup}_{r,\circ} \). That algorithm also produces one rule of Group II for each of p's original rules; those rules define the zeroth supplementary predicates of the rules for p in terms of \(\text{sup}_{r,\circ} \). Thus, the total number of Group I and II rules equals the number of IDB subgoals plus the number of rules.

In comparison, if we go directly from each supplementary predicate, such as \(\text{sup}_{r,\circ} \) in Example 13.10, to each of the zeroth supplementary predicates for the rules of p, then the number of rules we generate is the sum, over all the IDB subgoals in all the rules, of the number of rules that the predicate for that subgoal has. In the extreme case, where there is one IDB predicate \(p \) with \(n \) rules, and the average rule has one IDB subgoal, then Algorithm 13.1 generates \(n \) Group I rules for \(m_p \) and \(n \) Group II rules. In comparison, the modified algorithm generates \(n \) rules for each IDB subgoal, or \(n^2 \) rules.

13.6 GENERALIZED MAGIC SETS

When rules are not subgoal-rectified datalog, we saw in Examples 13.4 and 13.5 (Section 13.3) that the magic-sets rules may not be efficiently evaluatable bottom-up. If we examine these examples, we see two problems. In Example 13.4, we saw that computing the bindings for the third argument of p did not tell the full story. The top-down exploration of Figure 13.7 shows how it is possible to deduce more than a binding for the third argument. The rightmost grandchild of the root, the goal node \(p(X, X, 3)\), tells us not only a binding for the third argument; it also tells us that the first two arguments are equal, although we do not know which. When we follow the rightmost path in the tree for two more levels, we come to goal node \(p(4, 4, 5)\), and here we see that the equality between arguments deduced two levels earlier has led to actual binding of the first and second arguments.

The weakness of the magic-sets technique is that, while it records bindings for bound arguments, it doesn't record other useful relationships, such as the equality of two arguments. Example 13.5 illustrates another important relationship not represented by magic sets—the structure of terms as arguments. That is, the rightmost grandchild of the root in Figure 13.10 has a subgoal \(p(2, f(V))\), which, while it doesn't bind the second argument, imparts to it a special structure. That structure, which is also ignored by magic sets, is important when we reach the leaf \(a(2, f(V))\), because it enables us to rule out all of the tuples in the EDB relation for \(a\) that have a second component not of the form \(f(V)\).

Suppose we regard the magic predicate \(m_p\) as having arguments corresponding to all of the arguments of \(p\). We can then let the tuples in \(m_p\) correspond to a goal node in the rule/goal tree, together with one possible binding found in the binding set for that goal node. Tuples of this nature carry all the information that the goal nodes and binding relations carry during QRTG execution. The only problem is that these "tuples" are of a kind we have never seen before; they may have variables appearing as components or parts of components.

Example 13.11: Consider the predicate \(p\) of Example 13.4. The magic predicate \(m_p\), like \(p\), has three arguments. The query \(p(X, Y, 1)\) yields the tuple \((X, Y, 1)\) for the relation corresponding to \(m_p\). Recall that in the magic rules of Figure 13.8, \(m_p\) has only one component, corresponding to the third component of \(p\), and the initial tuple in \(m_p\) is just 1. In practice, there is little difference between a tuple of constants and the same tuple padded out in other attributes with distinct variables. However, the bindings for other \(p\)-nodes in the rule/goal tree of Figure 13.7 provide tuples that are not related so simply to the tuples that are placed in \(m_p\) by the magic rules of Figure 13.8.

When we pass bindings for \(Z_i\) sideways at the second level of goal nodes, we get only the binding \(Z_i = 3\) for the goal node \(p(X, X, Z_i)\). That yields the tuple \((X, X, 3)\) for the relation of \(m_p\), which is not the same as the tuple \(3\) found by the rules of Figure 13.8, because the dual occurrence of \(X\) tells us more than that the third component is 3. Note also that if the EDB predicate \(b\) binds \(Z_i\) to a set of several values, rather than to the singleton \(\{3\}\), there would be a tuple \((X, X, i)\) in \(m_p\) for each value \(i\) provided for \(Z_i\).

Finally, at the third level of goal nodes, we pass the binding \(Z_i = 5\) to the goal node \(p(X, X, Z_i)\), which yields the tuple \((4, 4, 5)\) for \(m_p\). Lower levels of the rule/goal tree pass the empty set of bindings to the \(p\)-subgoals, so there are no more tuples for \(m_p\). Thus,

\[
m_p = \{(X, Y, 1), (X, X, 3), (4, 4, 5)\}
\]

should be the complete relation for \(m_p\).
We also need to modify our notion of a supplementary predicate, so their
relations can carry information other than bound values for variables. Since
the magic predicates now have components for all arguments of the head, we
need to consider all variables appearing in the head as "bound," even though
the "binding" may be to another variable appearing in a tuple of the magic
predicate's relation. The extra information carried by the supplementary
relations is the same as the information carried by a particular rule instance in
the rule/goal tree. These modifications to the magic and supplementary predicates,
along with the modifications to the rules described below, will be called the
generalized magic-sets construction.

Example 13.12: For the rules of Example 13.4,
\[
\begin{align*}
  r_1: & \quad p(X,Y,W) : \neg a(X,Y,W), \\
  r_2: & \quad p(X,Y,W) : \neg b(X,Y,Z) \land p(X,X,Z).
\end{align*}
\]
If we follow Algorithm 13.1, with query \(\theta/p\), the predicate \(sup_{p,1}\) has only
arguments \(W, Y, Z\) and \(X\) in the rules of Figure 13.8, and these provide a binding
for only the third argument of \(p\). In the generalized magic-sets technique, we
include \(X\) among the attributes of \(sup_{p,1}\), simply because \(X\) appears in the
head of \(r_2\).

Modification of the Magic-Set Rules

To construct the generalized magic-set rules, we make the following modifications
to Algorithm 13.1. First, as mentioned, the magic predicate \(m.p\) now has arguments identical to those of \(p\) and its arguments to \(IDB\) predicate \(p\) itself. The supplementary
predicate \(sup_{p,1}\) has arguments corresponding to all variables that appear in the
head of rule \(r\) and all variables that appear in any of the first \(i-1\) subgoals of
and also appear in any of the \(i\)th and subsequent subgoals. The changes to
the rules themselves are listed by group.

1. \(Group I\) rules define the magic predicates. If rule \(r_j\) has an \(IDB\) subgoal
\(\sigma(t_1, \ldots, t_n)\) appearing as the \(i\)th subgoal, and \(X_1, \ldots, X_m\) are the variables
that are arguments of \(sup_{p, j-1}\), then there is a \(Group I\) rule
\[
\neg m.p(t_1, \ldots, t_n) : sup_{p, j-1}(X_1, \ldots, X_m).
\]
Note that this rule differs from the rule given by Algorithm 13.1 in that
the head is an exact copy of the subgoal, rather than the subgoal restricted
to the bound arguments.

2. \(Group II\) rules define the zeroth supplementary predicates. If rule \(r_j\) has
head \(\sigma(t_1, \ldots, t_n)\), and \(X_1, \ldots, X_m\) are the variables appearing in the head,
then there is the \(Group II\) rule
\[
sup_{p,0}(X_1, \ldots, X_m) : m.p(t_1, \ldots, t_n).
\]

Again, the difference is that Algorithm 13.1 restricts this rule to bound
arguments and variables, while here we include all arguments and variables.

3. There is no change to the \(Group III\) rules, except that the supplementary
predicates may have additional arguments, as indicated above.

4. Likewise, the \(Group IV\) rules are changed only by augmenting the set of
arguments of the supplementary predicate appearing in the rule.

5. The \(Group V\) rule is changed to include all of the arguments in the query,
whether free or bound. That is, if the query is \(q(t_1, \ldots, t_n)\), then the \(Group V\)
rule is
\[
m.p(t_1, \ldots, t_n).
\]
Note that this rule is not safe, if there are any variables in the query. However,
the generalized magic-sets technique deals quite easily with unsafe
rules; we simply put variables in tuples. In this case, we put the tuple
\((t_1, \ldots, t_n)\) in the relation for \(m.p\). In situations where the rules defining
the answer are unsafe, we shall find nonground tuples in the answer
relations.

Example 13.13: Let us consider the rules of Example 13.4,
\[
\begin{align*}
  r_1: & \quad p(X,Y,W) : \neg a(X,Y,W), \\
  r_2: & \quad p(X,Y,W) : \neg b(X,Y,Z) \land p(X,X,Z).
\end{align*}
\]
with the query \(p(X,Y,1)\). The generalized magic-set rules are shown in Figure
13.20. These rules should be compared with Figure 13.8. Notice, for example,
that \(X\) has been added as an argument of \(sup_{p,1}\). The predicate \(m.p\) now
has three arguments, and in rule (1), the arguments of \(m.p\) are a copy of the
arguments of the \(p\)-subgoal in \(r_2\).

Example 13.14: Now consider the rules from Example 13.5,
\[
\begin{align*}
  r_1: & \quad p(X) : \neg a(X). \\
  r_2: & \quad p(X) : \neg b(Z) \land p(Z,f(Y)).
\end{align*}
\]
with query \(p(X)\). The rules constructed by the generalized magic-set algo-
rithm are shown in Figure 13.21. These should be compared with the rules
of Figure 13.11.

Evaluating the Generalized Magic-Set Rules

When we apply one of the bottom-up algorithms, naive or semi-naïve evaluation,
to the generalized magic rules, we must be prepared to deal with variables and
nontrivial terms in tuples. The easy part is computing relations for the head of a
rule from the relation for the body. Recall the relation for the body is a relation
over all variables in the rule; it is constructed for ground tuples by Algorithm
12.2. We may perform the same VTOA construction to evaluate the relation
for the head that we did in Section 12.2 (bottom-up evaluation), realizing that variables in tuples of the body's relation are treated like constants when we perform VTOA. In fact, we can even allow rules to be unsafe. If a variable \( X \) appears in the head, but nowhere else, we simply leave it as a variable when constructing the relation for the head.

**Example 13.15**: Suppose the head of our rule is \( p(X, f(Y)) \), and the relation for the body of this rule is

\[
\begin{array}{ccc}
X & Y & Z \\
\text{a} & \text{B} & c \\
g(A) & A & d \\
\end{array}
\]


Note that here, and elsewhere in this section, we adopt the convention that variables of rules are represented by letters near the end of the alphabet, while variables in tuples are represented by letters near the beginning of the alphabet.

For the first tuple \((a, B, c)\), we substitute \( a \) for \( X \) and \( B \) for \( Y \) to obtain the tuple \((a, f(B))\) for the head relation. For the second tuple, we substitute \( g(A) \) for \( X \) and \( A \) for \( Y \), yielding the head tuple \((g(A), f(A))\).

The more difficult part is constructing the relation for the body of a rule by "joining" tuples that have variables. We shall explain the algorithm for the case of two subgoals; the generalization will then be obvious. Suppose the subgoals

\[
\begin{align*}
\text{Group I} \\
(1) & \quad m.p(X, Y, Z) : (-) \sup_{2.1} (W, X, Y, Z). \\
\text{Group II} \\
(2) & \quad \sup_{1.0} (X, Y, W) \quad - m.p(X, Y, W). \\
(3) & \quad \sup_{2.0} (X, Y, W) \quad - m.p(X, Y, W). \\
\text{Group III} \\
(4) & \quad \sup_{2.2} (X, Y, W, Z) \quad - \sup_{2.6} (X, Y, W) \land b(W, Y, Z). \\
\text{Group IV} \\
(5) & \quad p(X, Y, W) \quad - \sup_{2.0} (X, Y, W) \land a(X, Y, W). \\
(6) & \quad p(X, Y, W) \quad - \sup_{2.1} (X, Y, W, Z) \land p(X, Y, Z). \\
\text{Group V} \\
(7) & \quad m.p(X, Y, 1). \\
\end{align*}
\]

**Figure 13.20** Generalized magic rules for Example 13.13.

are \( p(t_1, \ldots, t_m) \) and \( q(s_1, \ldots, s_m) \), and suppose the relations for \( p \) and \( q \) have tuples \((u_1, \ldots, u_k)\) and \((v_1, \ldots, v_m)\), respectively. The situation is suggested by Figure 13.22. We may assume that the variables of tuple \((u_1, \ldots, u_k)\) do not appear in \((v_1, \ldots, v_m)\) or in either subgoal, and similarly for the other tuple. That is, the only sharing of variables occurs between the two subgoals.

**Figure 13.21** Generalized magic rules for Example 13.14.

\[
\begin{align*}
\text{Subgoals} & \quad p(t_1, \ldots, t_m) \land q(s_1, \ldots, s_m) \\
\text{Tuples} & \quad (u_1, \ldots, u_k) \quad (v_1, \ldots, v_m) \\
\end{align*}
\]

**Figure 13.22** Subgoals and tuples.

Intuitively, a tuple with variables represents all ground tuples that could be created by substituting ground terms for the variables. However, not all ground tuples so created necessarily will match the corresponding subgoal. To tell which ground tuples will match, we unify the tuple \( \mu \) with the subgoal, because the MGU \( \tau \) thus created represents the minimal constraints necessary to assure that a substitution for the tuple \( \tau(\mu) \) will match the subgoal.

However, it is not sufficient that the tuples are unified with their corresponding subgoals, because the subgoals generally share variables, and we must
be sure that the substitution for a variable \( X \), as we unify both tuples with their corresponding tuples, is consistent. Therefore, we must unify both subgoals with a pair of their corresponding tuples simultaneously. That is, we choose a "dummy" predicate name \( d \), and create atom

\[
d(t_1, \ldots, t_n, s_1, \ldots, s_m)
\]

from the subgoals of Figure 13.22 and atom

\[
d(u_1, \ldots, u_n, v_1, \ldots, v_m)
\]

from the tuples of Figure 13.22. We then unify these atoms. If the unification is unsuccessful, then this pair of tuples yields nothing for the relation of the body. If there is an MGU \( \tau \), then the tuple \( (\tau(X_1), \ldots, \tau(X_k)) \) is added to the relation for the body, where \( X_1, \ldots, X_k \) are all of the variables appearing in either subgoal.

The relation for the rule body is the set of all tuples so constructed, as we pair the tuples from the relation for one subgoal with the tuples in the relation for the other subgoal, in all possible ways. The resulting relation is called the **unification join** of the relations for the subgoals.

**Example 13.16:** Let us follow the bottom-up evaluation of the magic rules generated in Example 13.13. First, note that we can simplify the rules as we did in Section 13.4; the resulting rules are shown in Figure 13.23.

\( (7) \quad m.p(X,Y,1) \)
\( (1) \quad m.p(X,k,2) \)
\( (4) \quad s.u.p_2_1(W,X,Y,Z) \)
\( (6) \quad p(X,Y,Z) \)

Figure 13.23 Simplified magic rules for Example 13.16.

Figure 13.24 shows the EDB relations \( A \) and \( B \) for predicates \( a \) and \( b \), respectively, that we assumed in Example 13.4. That is, \( B \) contains two particular tuples, and \( A \) consists of a large number of tuples, each with third component 5 and first two components not both equal to 4.

Let us simulate bottom-up evaluation, computing relations \( M \) for predicate \( m.p \), \( P \) for \( p \), and \( S \) for \( s.u.p_2_1 \). On the first round, only rule (7) generates any tuples. Note that the body of rule (7) is empty, so the relation for its body should be regarded as consisting of the empty tuple. That tuple can be padded with components for the variables \( X \) and \( Y \), which appear only in the head of this unsafe rule. The lone tuple in the relation for the body is thus

\[
\begin{array}{c|c|c}
X & Y & A \\hline
A & B & 5
\end{array}
\]

Note that we follow the convention of using letters at the beginning of the alphabet for variables in tuples. The particular names used in tuples do not matter; we could have used \( X \) and \( Y \), of course.

\[
\begin{array}{c|c|c|c|c}
A & B & \\hline
m & n & 5
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
A & B & \\hline
1 & 2 & 3 & 3 & 4 & 5
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
M & S & \\hline
A & B & \\hline
1 & 2 & 3
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
A & B & \\hline
3 & 4 & 4 & 5
\end{array}
\]

**Figure 13.24** Relations used in Example 13.16.

When we translate the tuple \((A, B)\) to the head relation, we get the tuple \((A, B, 1)\) for the relation \( M \), as indicated in Figure 13.24. On the next round, we can use this tuple in rule (4), along with the tuples of the relation \( B \), to create tuples for the relation of the body of rule (4); these tuples have arguments corresponding to variables \( WXYZ \).

Let us begin by matching the tuple \((A, B, 1)\) from \( M \) with the tuple \((1, 2, 3)\) from \( B \). The dummy atom constructed from the subgoals of rule (4) is

\[
d(X, Y, W, W, Y, Z)
\]

Note that we simply copy the arguments of the two subgoals of rule (4), in the order of appearance. The dummy atom for the tuples is

\[
d(A, B, 1, 1, 2, 3)
\]

The MGU \( \tau \) for this atom and (13.6) exists and is

\[
\tau(W) = 1 \quad \tau(X) = \tau(A) = A \quad \tau(Y) = \tau(B) = 2 \quad \tau(Z) = 3
\]

Thus, the resulting tuple for the body relation is

\[
\begin{array}{c|c|c|c|c}
W & X & Y & Z & 1 & 2 & 3
\end{array}
\]

and this tuple is also a tuple of the head, \( s.u.p_2_1 \), of rule (4), as indicated by the relation \( S \) in Figure 13.24.
We now must match tuple \((A, B, 1)\) against the other tuple in the relation for predicate \(b\), which is \((3, 4, 5)\). That is, we must unify (13.6) with the atom \(d(A, B, 1, 3, 4, 5)\).

Evidently, \(W\) cannot match both 1 and 3, so no tuple for the rule is created. Now we can use the tuple \((1, A, 2, 3)\) for \(sup_2\) in rule (1) to create the tuple \((A, A, 3)\) for \(m.p\). That, in turn, is used in rule (4) to match with the tuple \((3, 4, 5)\) from relation \(B\). We must unify atom (13.6) with the atom \(d(A, A, 3, 3, 4, 5)\)

and we find that the MGU \(\tau = \tau(X) = \tau(Y) = \tau(A) = 4, \tau(W) = 3, \) and \(\tau(Z) = 5\). The resulting tuple for \(sup_1\) is \((3, 4, 5)\), and this tuple yields \((4, 5)\) for \(m.p\), by rule (1).

We claim that there are no more tuples deduced. None of the three tuples in \(M\) match any of the tuples in relation \(a\). The first two, \((A, B, 1)\) and \((A, A, 3)\), do not match any, because all tuples in relation \(a\) have third component 5. The last tuple, \((4, 4, 5)\), cannot match because this tuple was explicitly ruled out of \(a\). Thus, rule (5) yields nothing. Rule (6) cannot yield any tuples as long as \(P\) remains empty. Finally, there are no undiscovered matches between the tuples of \(M\) and the tuples of relation \(B\), as the reader may check. We conclude that the relation \(P\), and therefore the answer to the query, is empty. □

Example 13.17: A similar sequence of events happens when we execute the rules of Example 13.14 bottom-up. We show the rules in Figure 13.25 with the simplifications of Section 13.4 applied.

\[
\begin{align*}
(7) & \quad m.p(1, y) . \\
(1) & \quad m.p(2, f(y)) : - sup_2 : (X, Y, Z) . \\
(4) & \quad sup_2 : (X, Y, Z) : - m.p(X, Y) & b(X, Z) . \\
(5) & \quad p(X, Y) : - m.p(X, Y) & a(X, Y) . \\
(6) & \quad p(X, Y) : - sup_2 : (X, Y, Z) & p(Z, f(Y)) .
\end{align*}
\]

Figure 13.25 Simplified magic rules for Example 13.17.

Suppose we have the EDB relations of Example 13.14, where the relation for \(b\) has only the tuple \((1, 2)\), and the relation for \(a\) consists of tuples of the form \((2, n)\), for various values of constant \(n\), none of which are of the form \(f(Y)\) for any \(Y\). Let \(M\) be the relation for \(m.p\) and \(S\) the relation for \(sup_2\). Then (7) on the second round, we add \((1, A, 2)\) to \(S\) by rule (4), and on the third round we add \((2, f(A))\) to \(M\), by rule (1). At that point, none of the rules yield any more tuples, so we conclude that the answer to the query is empty. Notice particularly that in rule (5), neither \((1, A)\) nor \((2, f(A))\) unifies with any tuple from any \(M\).

Subsumption

When adding tuples with variables to relations, we must be careful to recognize that two distinct tuples may actually be the same. For instance, in Example 13.16, we added the tuple \((A, A, 3)\) to relation \(M\). If we later tried to add the tuple \((B, B, 3)\) to \(M\), we must realize that these are really the same tuple, with different variable names, and not add the latter tuple. More importantly, if we add the tuple \((B, C, 3)\) to \(M\), we must delete the tuple \((A, A, 3)\), because the former is more general than the latter. That is, \((A, A, 3)\) in \(M\) says we are interested in tuples for predicate \(p\) where the third component is 3 and the first two components are equal; \((B, C, 3)\) says we are interested in \(p\)-tuples with a third component of 3 and any first and second components. Hence, \((A, A, 3)\) will not result in any \(p\)-tuples beyond what \((B, C, 3)\) produces, and we may delete \((A, A, 3)\) from \(M\) without changing what the rules compute.

We say tuple \(\mu\) subsumes tuple \(\nu\) if there is some substitution for the variables of \(\mu\) that turns it into \(\nu\). If \(\mu\) and \(\nu\) each subsume the other, then they are equivalent. Any tuple that is subsumed by another tuple in the same relation may be removed from that relation, and either of two equivalent tuples in the same relation may be removed without changing what the relation represents. That is, if \(\mu\) is subsumed by \(\nu\), then any substitution of ground terms for the variables of \(\mu\) can be mimicked by a substitution on \(\nu\).

Testing for subsumption is easy. To test whether \(\mu\) subsumes \(\nu\), treat each of the variables of \(\nu\) as a distinct constant, and see whether \(\mu\) term-matches the resulting tuple. For example, \((B, C, 3)\) subsumes \((A, A, 3)\), because of the substitution \(\tau(B) = \tau(C) = A\). However, there is no substitution for \(A\) that makes \((A, A, 3)\) become \((B, C, 3)\), so the former does not subsume the latter.

Use of subsumption to eliminate tuples is often essential for efficiency, or even for convergence of the query evaluation. The next example illustrates the point.

Example 13.18: Consider the rules
\[
\begin{align*}
\text{p}(X) & \leftarrow \text{p}(f(X)) . \\
\text{p}(X) & \leftarrow a(X) .
\end{align*}
\]

Note that, as long as the EDB relation for \(a\) is finite, the relation for \(p\) will be finite; it consists of all the tuples in \(a\) plus those that can be obtained by stripping away the unary function symbol \(f\) one or more times. For example, if \(a\) has the one-component tuple \(f(f(1))\), then \(p\) has the three tuples \((1, f(1), f(f(1)))\).

If we apply the generalized magic-sets algorithm to these rules, with the query \(\text{p}(X)\), that is, find all \(p\) facts, and then simplify according to Section...
3.4. We get the four rules shown in Figure 13.26. Suppose we do not use sub-
sumption. Rule (1) tells us tuple A, that is, a single component consisting of a
variable, is in M, the relation for m.p. Then rule (2) tells us f(A), f(f(A)),
and so on, are in M, and the computation never stops.

However, note that f(A) is subsumed by A, and so is each of the tuples we
try to place in M, since A is the most general possible one-component tuple.
Thus, we should not add any of these tuples to M; the correct value of M is
{A}, that is, a single tuple, whose lone component is a variable. If we do, then
rule (4) simply tells us that anything in a is in p, and rule (3) lets us strip f’s
off tuples in p to get new tuples in p. This process terminates after a finite time
if we start with a finite number of tuples, each with a finite number of f’s, as
the EDB relation for a.

Analysis of the Generalized Magic-Set Algorithm

We contend that the rules constructed by the generalized magic-set algorithm are evaluated by semi-naive evaluation in time no worse than proportional to
the time taken by QCGT on the original rules. However, to make this claim,
we need an assumption that is open to question: as for Theorem 13.2, we
must suppose that we can find the tuples in a given relation that match given
values in some of their components, in constant time. For the basic magic-sets
algorithm, Algorithm 13.1, that assumption was fairly realistic, as “match”
meant to have identical constants, and index structures are known to provide
close to constant-time performance. However, for the generalized magic-sets
algorithm, “match” means “unify,” and there is no reason to believe that we
can select the tuples that unify with a given tuple quickly. There are certain
data structures for “partial-match retrieval” (see Sections 6.12 through 6.14 of
Volume I) that can help, and they have even been implemented and shown to
offer reasonably good performance for the problem of finding unifying tuples.11

If we are willing to ignore this retrieval problem, then we can claim the following

Theorem 13.4: The generalized magic-set rules, executed by semi-naive eval-

uation with subsumption, never take time that is more than proportional to
the time taken by QCGT applied to the original rules.

Proof: We shall only outline the proof, which parallels the proof of Lemma 13.3
and Theorem 13.2. We must again identify “events,” which are the insertion of
a tuple into a relation or the matching of tuples from the two subgoals in the
body of a Group III or IV rule. For each tuple μ added to a magic predicate, say
m.p, there is a goal node N for predicate p in the rule/goal tree during QCGT
and a binding tuple passed down to N, such that μ is the result of substituting
these bindings for the appropriate arguments in the instance of p at node N.
Similar claims are made for the tuples for the supplementary predicates and
the answer tuples. Then, an induction like that of Lemma 13.3 proves that
every event during the bottom-up evaluation of the magic rules is mirrored by
at least one event during QCGT.

While Theorem 13.5 only claims that the magic rules are not worse (by
more than a constant factor) than the original rules executed top-down, we
should be aware that the generalized magic rules can be much better than the
rules from which they came. For instance, the rules of Example 13.18, when
transformed by the generalized magic-sets algorithm into the rules of Figure
13.26, execute as efficiently as one could imagine possible for the problem they
solve. However, the original rules (or even the transformed rules of Figure 13.26)
will go into an infinite loop, where the infinite sequence of goals

\[ p(X), p(f(X)), p(f(f(X))), \ldots \]

is generated, if we execute either QCGT or Prolog on these rules.

Comparison of Methods

It appears that the development of database applications for logic processing,
where all solutions, rather than a single solution, is wanted, has caused rapid
evolution in the way we think about processing logic. The situation may be
analogous to that in parsing, where top-down methods, such as recursive de-
scent, were for a long time thought the preferred way to parse, but more recently
it has come to be realized that bottom-up parsing, for example, LR-parsing,
offers everything that top-down offers, both in terms of language recognition
and in terms of translation.12 In logic processing, it appears that top-down
or resolution-based methods are likewise dominated by bottom-up methods, as
concepts like magic-sets become fully known and explored.

At the least, the basic magic-set algorithm is usually the method of choice
for datalog, and for some nondatalog examples. However, there are some special
cases of rules where magic-sets are beaten by other bottom-up methods; these

11 See the references to the NU-Prolog system in Chapter 16.

12 See Aho, Sethi, and Ullman [1986] for a discussion of these issues.
techniques are discussed in Chapter 15. It appears likely that for many, if not all, realistic examples, the generalized magic-sets technique will allow us to find the appropriate data structures so that unification joins, where we find pairs of

tuples in two relations that unify with each other, can be taken efficiently. As

an important special case, if the given rules are subgoal-rectified datalog, then

the tuples in the relations computed by the generalized algorithm are just the
tuples of the basic algorithm, padded out with distinct variables in the extra

components. In that case, it is easy to show that the unification joins are really

ordinary joins, and the usual index data structures provide adequate support,
as was discussed in Section 13.3.

EXERCISES

13.1: Transform the rules

\begin{align*}
  \text{path}(X,Y) & : = \text{red}(X,Y). \\
  \text{path}(X,Y) & : = \text{path}(X,U) \& \text{blue}(U,V) \& \text{path}(V,Y). \\
\end{align*}

assuming a query of the form \text{path}\^M, and using

a) Algorithm 13.1 (the basic magic-sets algorithm).

b) The modified technique of Section 13.5, where magic predicates are

omitted.

c) The generalized magic-sets technique of Section 13.6.

If we think of \text{red} and \text{blue} as representing red and blue arcs in a graph,

then \text{path} represents paths of alternating red and blue arcs, beginning and

ending with a red arc.

13.2: Repeat Exercise 13.1 on the rules of Figure 13.27 with the query goal

\text{expression}\^M. These rules represent the construction of arithmetic expres-
sions with operators + and \times, and parentheses to control order of eval-
uation.

\begin{align*}
  \text{expression}(\text{plus}(E,T)) & : = \text{expression}(E) \& \text{term}(T). \\
  \text{expression}(E) & : = \text{term}(E). \\
  \text{term}(\text{times}(T,F)) & : = \text{term}(T) \& \text{factor}(F). \\
  \text{term}(T) & : = \text{factor}(T). \\
  \text{factor}(\text{paren}(E)) & : = \text{expression}(E). \\
  \text{factor}(F) & : = \text{identifier}(F). \\
\end{align*}

Figure 13.27 Rules for arithmetic expressions.

EXERCISES

13.3: Repeat Exercise 13.2 with the query \text{expression}\^M. What happens when

you try to evaluate the magic rules bottom-up?

13.4: Repeat Exercise 13.1 with the rules of Example 12.2 (Figure 12.1) and

the query \text{sum}\^M.

13.5: It is possible to write the "same-generation" rules as

\begin{align*}
  \text{sg}(X,X). \\
  \text{sg}(X,Y) & : = \text{par}(X,Xp) \& \text{sg}(Xp,Yp) \& \text{par}(Y,Yp). \\
\end{align*}

Here, the first rule is evidently unsafe. Repeat Exercise 13.1 on these rules,

with the query \text{sg}\^M. Does the unsafety of the given rules create problems

when we evaluate the magic rules bottom-up?

13.6: Simplify the rules you generated for each of parts (a) to (c) in (i) Exercise


13.5.

13.7: Prove the claim of Example 13.1 about what \text{m.sg}, \text{sup}_{2,1}, \text{sup}_{2,2} and \text{sg}

represent.

13.8: Prove the claim of Example 13.2 about what \text{m.gl}, \text{sup}_{0,0}, \text{sup}_{2,1} and \text{gl}

represent.

13.9: For the paths rules of Exercise 13.3, determine what each of the predicates

in the magic rules of Figure 13.5 represent and prove your conclusion.

13.10: Suppose we construct a rule/goal tree for rules that have the unique binding

property with respect to a given query goal. Show that in the rule/goal tree,
every goal node has at least the bound arguments that are indicated for
its predicate by the unique binding for that predicate.

13.11: Suppose we have an "EDB" predicate \text{equals}(A, B, C, D) that is true whenever
\begin{align*}
  A & = B \\
  C & = D. \\
\end{align*}

What adornments for \text{equals} should be permissible if we want to pretend the relation for \text{equals} is finite (as for predicate \text{s in}

Example 13.3)?

13.12: Prove part (c) of Lemma 13.2.

13.13: Complete the proof of Lemma 13.3.

13.14: Simulate semi-naïve evaluation on the magic rules of

a) Example 13.4.

b) Example 13.5.

In each case, show that a large number of tuples are placed in the relation

for \text{p}, even though none of these answer the query.

13.15: Complete the proof of Theorem 13.3.
3.16: Prove that the simplification rule given at the end of Section 13.4 always works. That is, we can eliminate the ith supplementary predicate for a rule if i = 0 or if i > 0 and the (i + 1)th subgoal is EDB.

3.17: Show that if Algorithm 13.1 is applied to safe rules, then the resulting magic rules are safe.

3.18: Define a tuple with variables to be *null* if each component is either a ground term or a variable that appears nowhere else in the tuple. Thus, \((a, A, b, B)\) is null, but \((a, A, b, f(B))\) are not. Define \(\text{GROUND}(\mu)\) to be the components of tuple \(\mu\) with ground terms. For example, \(\text{GROUND}(aAbB) = ab\). If \(R\) is a relation that is homogeneous, in the sense that the components with ground terms in each tuple of \(R\) are the same, then let \(\text{GROUND}(R)\) be the union of \(\text{GROUND}(\mu)\) for each tuple \(\mu\) in \(R\). Suppose that we use the unification join of Section 13.6 to compute the relation \(R\) for the body of a rule with subgoals \(G_1\) and \(G_2\), whose relations are \(R_1\) and \(R_2\), respectively. Let \(R_1\) and \(R_2\) be homogeneous, and let \(G'_1\) and \(G'_2\) be \(G_1\) and \(G_2\) restricted to components where \(R_1\) and \(R_2\), respectively, have ground terms. Finally, assume all tuples of \(R_1\) and \(R_2\) are null. Prove the relationship

\[
\text{GROUND}(R) = \text{ATOV}(G'_1, \text{GROUND}(R_1)) \cup \text{ATOV}(G'_2, \text{GROUND}(R_2))
\]

Less formally, for null tuples, the unification join gives us what we would get if we threw away all nonground components, evaluated the body as if they were ordinary tuples (tuples with no variables), and then picked the resulting tuples with variables that appear only once, to add components for those variables of the rule that appear only in unbound components of \(R_1\) and \(R_2\).

3.19: Suppose we perform the basic magic-sets construction (Algorithm 13.1) on a datalog program with the subgoal-rectified and unique-binding properties, and that the resulting logic program is \(P_1\). Also, we perform the generalized magic-sets construction of Section 13.6 on the same datalog rules to produce the logic program \(P_2\).

a) Show that the relations defined by \(P_2\) will have only null tuples.\(^{12}\)

b) Show that for each predicate \(q\) appearing in \(P_2\) and \(P_2\) (note the sets of predicates are the same), if \(Q\) is the relation for \(q\) produced by \(P_2\), then the relation for \(q\) produced by \(P_1\) is \(\text{GROUND}(Q)\).

3.20: Complete Example 13.17 by showing that no more tuples can be added to the relations for \(p, m, p\), or \(\text{sup}_{21}\).

---

\(^{12}\) The terms "null" and \(\text{GROUND}\) used in parts (a) and (b) are defined in Exercise 13.18.

---

13.21: Suppose we have the rules of Example 13.18,

\[
p(X) :: = p(f(X))
\]

\[
p(X) :: = a(X)
\]

with a finite EDB relation for \(a\)?

a) What happens if naive or semi-naive evaluation (without subsumption) is performed on these rules?

b) What happens if QRGT is performed on these rules with a query goal \(p^d\)?

c) What happens when naive or semi-naive evaluation is performed on the corresponding magic rules of Figure 13.26, again assuming no subsumption?

d) What happens when QRGT is performed on the rules of Figure 13.26 with the query \(p^d\)?

13.22: In Figure 13.28 are five tuples with variables in some components. Which of these are subsumed by others?

<table>
<thead>
<tr>
<th>(X)</th>
<th>(Y)</th>
<th>(Z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i)</td>
<td>(a)</td>
<td>(f(B))</td>
</tr>
<tr>
<td>(ii)</td>
<td>(A)</td>
<td>(f(B))</td>
</tr>
<tr>
<td>(iii)</td>
<td>(a)</td>
<td>(B)</td>
</tr>
<tr>
<td>(iv)</td>
<td>(f(A))</td>
<td>(f(B))</td>
</tr>
<tr>
<td>(v)</td>
<td>(A)</td>
<td>(f(A))</td>
</tr>
</tbody>
</table>

Figure 13.28 Tuples containing variables.

13.23: Compute \(\text{VTOA}(p(W, X, Y), R)\), where \(R\) is the relation of Figure 13.28.

13.24: Write an algorithm to take the unification join of \(k\) relations for any \(k \geq 2\).

---

**BIBLIOGRAPHIC NOTES**

The "magic sets" technique of rule rewriting was expressed for linear rules by Barcelo, Maier, Sagi, and Ullman [1986]. Independently, Rohmer, Lescoeur, and Kepis [1986] described a method similar to Algorithm 13.1. Algorithm 13.1 itself is from Boer and Ramakrishnan [1987], where it is called "generalized, supplementary magic sets." The simplified rules described at the end of Section 13.4, where supplementary predicates are used only for IDB predicates, is based on the "minimagic" algorithm of Sacca and Zaniolo [1987a].

The generalized magic-sets construction of Section 13.6 was first published by Ramakrishnan [1988]. It was also discovered independently by Seki [1988].
Theorem 13.2, showing bottom-up dominates top-down for datalog, is from Ullman [1988].

Stratified Logic

When a logic program has stratified negation, the straightforward magic-sets construction sometimes leads to nonstratified rules. Algorithms for dealing with this problem have been considered by Balbin, Port, and Ramamohanarao [1987], Port, Balbin, Meenakshi, and Ramamohanarao [1988], Kemp and Topor [1988], Seki and Itoh [1988], and Balbin, Meenakshi, and Ramamohanarao [1988].

Memoing

There is a long list of seemingly unrelated papers that, in essence, express the same idea, often called “memoing.” The concept is to reduce the work of bottom-up computation by remembering when a query has been asked (i.e., remembering the goal nodes of the rule/goal tree) and setting up a network of producers and consumers, where answers to one query are passed to other queries that may be able to use those answers in inferences.

McKay and Shapiro [1981] was an early implementation of this idea. Pereira and Warren [1983] propose “Earley deduction,” a technique for memoing patterned after the parsing algorithm due to Earley [1970]; the technique was further explored by Porter [1986].

The following papers examine this idea in its various guises: Lozinskii [1985], Kifer and Lozinskii [1985], Dietrich and Warren [1985], Neiman [1986], Tamaki and Sato [1986], Van Gelder [1986b], Dietrich [1987], and Vincze [1987, 1988]. It appears that all these techniques are subsumed by the generalized magic-sets method mentioned earlier.

Bancilhon and Ramakrishnan [1986] survey and compare many of these papers, as well as the magic-set papers discussed above.

Other Optimization Techniques

There are a number of interesting approaches that should be mentioned, but are not related to the material found in other chapters. Gangopadhyay [1987] explores rule-rewriting schemes that are independent of the binding pattern of the query. Ioannidis and Wong [1987a] investigate local improvements to expressions for queries by the technique known as “simulated annealing.”

The MRS system of Genesereth [1983] allows programmer control of evaluation techniques, with different methods used for different predicates. Treitel and Genesereth [1987] and Treitel and Smith [1988] explore the combination of top-down and bottom-up techniques, with the appropriate technique applied to each predicate.