July 1984

Report No. STAN-CS-84-1009

Complexity of A Top-Down Capture Rule

by

Yehoshua Sagiv and Jeffrey D. Ullman

Department of Computer Science

Stanford University
Stanford, CA 94305

COMPLEXITY OF A TOP-DOWN CAPTURE RULE

Yehoshua Sagiv
Hebrew University

Jeffrey D. Ullman[†]
Stanford Univ.

ABSTRACT

Capture rules were introduced in [U] as a method for planning the evaluation of a query expressed in first-order logic. We examine a capture rule that is substantiated by a simple top-down implementation of restricted Horn clause logic. A necessary and sufficient condition for the top-down algorithm to converge is shown. It is proved that, provided there is a bound on the number of arguments of predicates, the test can be performed in polynomial time; however, if the arity of predicates is made part of the input, then the problem of deciding whether the top-down algorithm converges is NP-hard. We then consider relaxation of some of our constraints on the form of the logic, showing that success of the top-down algorithm can still be tested in polynomial time if the number of arguments is limited and in exponential time if not.

I. PRELIMINARIES

We consider the question of capture rules and their substantiation as outlined in [U]. The purpose is to plan the evaluation of queries expressed in logical terms. Suppose we are given a "logic program" in the form of first-order Horn clauses, which we shall write in the Prolog form:

$$A(x_1, \ldots, x_n) \text{ :- } B_1(y_{11}, \ldots, y_{1m_1}), \ldots, B_k(y_{k1}, \ldots, y_{km_k})$$

The meaning of such a statement is that B_1 and B_2 and . . . and B_k imply A. Initially, we shall make the following restrictions on the form of the clauses.

1. Arguments may not be constants.
2. Arguments may not involve functions; e.g., $A(x, y)$ is legal, but $A(f(x), y)$ is not.
3. A variable may appear only once on the left side of a rule. Thus, $A(x, x)$:- $B(x, y)$, $C(y, y)$ is illegal because of the

† Work partially supported by AFOSR grant 80-0212 and an Einstein Fellowship of the Israeli Academy of Arts and Sciences. The work was performed at Hebrew University.

arguments of A, although C, being on the right side, is permitted to have repeated arguments.

Conditions (1) and (3) will eventually be relaxed. Condition (2) is a familiar simplification made by people investigating logic applied to databases, such as [HN]. In Section IV we shall briefly consider applications of our results to rules that do not obey (2).

When discussing databases with a collection of logical ruler; that affect the interpretation of the data, it is normal to identify certain predicate symbols as representing relations of the database; the remaining predicates take as values the relation consisting of all facts about that predicate that can be deduced from the database. Database predicates are assumed not to appear on the left sides of rules; that is, they are not defined in more primitive terms. The use of databases with associated logic was discussed in many of the articles in [GM], for example.

A query to such a database is a "goal," or predicate symbol with given arguments, which may each be either bound or free. A query $A(x_1, \ldots, x_n)$ is intended to produce a relation over those of the x_i's that are free. It is formed by taking the relation for A and performing a selection in which, if x_i is a bound variable, then the i^{th} component of the relation must have value x_i.

Example 1: Let us introduce our running example: computing the transitive closure of a graph. We take $E(x, y)$ to be a database relation, which we may interpret as the arcs of a directed graph. The transitive closure of E may be defined by the two rules:

$$r_1\text{: } T(x, y) \text{ :- } E(x, y)$$
$$r_2\text{: } T(x, y) \text{ :- } E(x, z), T(z, y)$$

Then the relation associated with T is easily seen to be the transitive closure of E.

A possible query is $T(1, w)$, that is, list all the nodes that can be reached from node 1. In principle, the answer to this query is found by computing T, selecting for the first component equal to **1**, and then projecting onto the second component. There are more efficient ways to answer that query, and one of the purposes of this paper is to investigate a more efficient algorithm and a test for whether that algorithm works for a given set of rules. .

Rule /Goal Trees

One way to define the relation denoted by a query is to expand it in a tree of rules and goals. This is the methodology followed by Prolog, and it has been used in a number of systems recently, such as

[MS] and [T*]. The nodes of the tree are either ***goal nodes***, corresponding to predicates with specific arguments, or they are ***rule*** nodes, corresponding to Horn clauses, also with a particular substitution for its variables. The root node is a goal node corresponding to the query.

The children of a rule node are goal nodes corresponding to the terms on the right side of the rule. The children of each goal node correspond to the rules whose heads (left sides) unify with the goal. Whatever substitution for variables is implied by the unification applies not only to the rule node, but to its goal children.

Example 2: Consider the goal $T(1, w)$ from Example **1.** The upper levels of the rule/goal tree for this query are shown in Fig. **1.** Conventionally, we represent rule nodes by their left sides and the :- symbol only; the complete rule can be deduced from the children of the rule node. Since variables are local to a rule node and its goal children, we have used arbitrary values for the free variable z appearing in rule r_2; z and $z\,1$ are the values chosen. Note that the tree is infinite, since the goal $T(z\,1, w)$ is equivalent, after a substitution of variables, to its ancestor goal $T(z, w)$. •

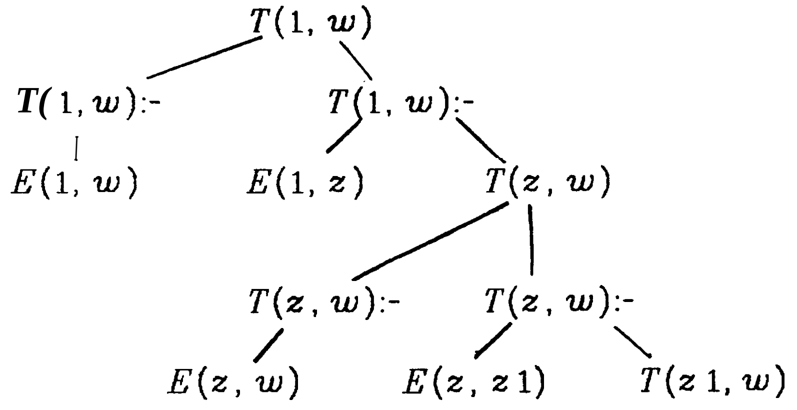


Fig. 1. Rule/goal tree.

We may define relations for the nodes of a rule/goal tree as follows. Relations for goal nodes are over those arguments of tne predicate symbol for that goal that are free variables, and relations for rule nodes are over all variables appearing in the rule. To compute the relation for a goal, take the relations for its rule children, project them onto the variables of the goal, and take the union. To compute the relation for a rule, take the natural join of the relations for its rule children. There is a technicality that if there is a variable appearing on the left side of the rule but not the sight, we must take the Cartesian product of the result with a copy of the "domain" of the variable. If that domain is infinite, this operation may not make sense, so we shall assume that the component for any such variable is represented by a placeholder and that we do not actually compute the Cartesian product. See [U] for details of the construction of relations as we work up the tree.

Even though the tree is infinite, the above construction makes sense. (See [CH], e.g.) The reason is that all the operations on relations that we use are monotone, meaning that if we add tuples to one of the arguments, the effect on the result of the operation is only the addition of zero or more tuples. For example, the set difference A-B of relations is not monotone, because we could add tuples to B and as a consequence delete tuples from the result.

We may therefore define the relation associated with the root of an infinite rule/goal tree to be the limit of the relation we get by cutting off the tree at the top n levels, computing the relation of the root by assuming that nondatabase relations at the bottom level are empty, and taking the limit as $n \rightarrow \infty$. Of course the resulting relation could be infinite, even if the database relations are finite.

Rule/Goal Graphs

A joke told by mathematicians goes as follows. The professor was teaching the formula

$$\sum_{i=1}^{n} i = n(n+1)/2$$

The clever student was asked to give a concrete value of n and to tell what the formula said. The student replied "let $n = n_0$. Then the formula says"

$$\sum_{i=1}^{n_0} i = n_0(n_0+1)/2$$

Well we don't think that's very funny either, but it does serve to introduce an important point. It is necessary to distinguish between free variables and "bound variables." The latter are symbols representing values that will be known as we implement the query, but are not necessarily known at the planning stage. The former represent attributes in the relation printed as a response to the query, or in intermediate relations that are computed to help form the response to the query.

In planning how to respond to a query, we may use the fact that a bound variable, like n_0, is fixed and finite in any situation, while free variables can assume any of an unlimited number of values in the same relation. As a simple example, if answering a query requires us to answer questions of the form "what is the sum of the first n_0 squares," we know we can write an iterative program that will sum the first n_0 squares. The loop will eventually terminate, although we cannot predict how long it will take until we know n_0. In contrast, the query "what is the sum of the first n squares" asks for an infinite set of pairs $(n, (n(n+1)/2)$, and there is no algorithm that can compute it.

To take advantage of this distinction between bound and free variables, and to help with the planning of efficient strategies to answer complex queries, [U] introduced "rule/goal graphs," in which the nodes correspond to predicates and rules, with a specified selection of variables bound. We shall assume that the order of variables is fixed, and that order will be the lexicographic one by default. Then a goal like $T(x, y)$ is represented in the rule/goal graph by four nodes T^q, where $q = q_1q_2$, and each q_i is either b (bound) or f (free). Thus, T^{bf} corresponds to the case where x is bound and y is free. Similarly, rules are represented by nodes with superscripts to indicate which variables are bound and which are free. Thus, r_2^{bff} represents r_2 of Example **1** with x bound and y and z free.

The predecessors of a goal node correspond to all the rules whose heads have the same predicate as the goal. Similarly, the predecessors of a rule node are the goals that appear on the right of the rule. In both cases, the superscripts must be correct, and the rule to follow is that X^q has Y^p as a predecessor only if p makes a variable bound exactly when q makes that variable bound.

Example 3: Figure 2 shows that portion of the rule/goal graph for the rules of Example **1** that is relevant to the node T^{bf} . That is, we show only T^{bf} , its predecessors, their predecessors, and so on. ▪

Capture Rules

Also introduced in [U] is the notion of "capturing" a node of the rule/goal graph. When we capture a node like T^{bf} we must have an algorithm that, given any bound value, say x_0, for the first argument of T, will produce the relation that is the set of y for which $T(x_0, y)$ is true, i.e., the relation for T, selected for $x = x_0$, and projected onto the y-component. This algorithm, called the ***substantiation,*** may only make use of algorithms that produce the

$$\begin{tikzcd}
T^{bf} & r_1^{bf} \arrow[l] & E^{bf} \arrow[l] \arrow[dll] \\
r_2^{bff} \arrow[u] & & \\
T^{ff} \arrow[u] \arrow[d, bend right] & r_1^{ff} \arrow[l] & E^{ff} \arrow[l] \arrow[dll] \\
r_2^{fff} \arrow[u] & &
\end{tikzcd}$$

Fig. 2. Rule/goal graph.

relations for certain other, already-captured nodes, given values for any variables that the superscripts for these nodes indicate are bound.

[U] discusses several capture rules and the requirements that a capture rule must follow. Chief among these is the requirement that whenever the rule applies to capture a node (or set of nodes) of the rule/goal graph, supported by the fact that another set of nodes is already captured, there is an algorithm that for any node among the set just captured, takes the values of the bound variables for that node, and computes the relation for that node and bindings, calling only on hypothetical routines that do the same for the nodes supporting the capture. As a simple example, we may capture any node whose predicate is a database relation, and if the predecessors of a node in the rule/goal graph are all captured, then we may capture that node. However, there are other, more complex rules that work under a variety of conditions.

Capture rules help us to plan the implementation of a query, and they are analogous to query optimization strategies in ordinary relational database theory. They must be independent, in the sense that the test for applicability of a rule is local; it depends only on the set of nodes supporting the capture, and not on the particular algorithms that substantiate the capture of the supporting nodes. Different capture rules will typically have substantiations of different costs, and it would be natural to make passes over the rule/goal graph using capture rules of increasing cost, until the goal node corresponding to the query is captured. By unraveling the sequence of captures, we have a method for computing the query, and that method is likely to be as efficiently implementable as any algorithm for answering that query.

However, while it is easy to state capture rules and their substantiations' it is much harder to answer the question: given an algorithm for computing relations, what is the most general capture rule that it substantiates? Put another way, can we provide a necessary and sufficient condition for a given query-implementation algorithm to work? We shall next discuss a particular, very simple algorithm and show exactly the conditions that will allow it to be used to capture a set of nodes.

II. AN ALGORITHM FOR SUBSTANTIATING CERTAIN CAPTURE RULES

In this section we shall first introduce the notion of a "downwards dependency" in database relations. This idea is used to define a variant of top-down construction of the rule/goal tree. In the next section we give a test for whether this construction converges when applied to a given set of rules.

Downwards Dependencies

Let $R(X_1, \ldots, X_n)$ be a database relation. We shall assume that the domain of each attribute X_i is the nonnegative integers. Our real requirement is that there be a partial order < on each domain with no infinite descending chains, i.e., infinite sequences $a_1>a_2> \cdots$. However, our proofs are easily seen to generalize to such partial orders, and all our counterexamples, will use the nonnegative integers.

We shall take the dependency $X_i => X_j$ to mean that in every tuple of R, the jth component is strictly less than the ith component.

Example 4: If *E (X, Y)* represents the edges of a finite acyclic directed graph, then we may assume that if there is an arc $n \rightarrow m$, then $n > m$. If that is the case, then there is a dependency $X => Y$. We could also assume that the nodes were numbered so that if $n \rightarrow m$ then $n < m$, which would give us the dependency Y => X. However, we cannot assume both at once.

For another example, in the relation

EMPS(EMP, SALARY, MGR)

we might assume that *EMP => MGR*. That would make sense if there were no cycles in the managerial hierarchy (typical organizations seem to have cycle-free hierarchies) and the hierarchy had a finite number of levels (even corporations with an infinite number of employees seem to have a finite number of levels of manager). Again, we could also choose to order the names of employees so that the dependency ***MGR*** => *EMP* held, but we could not have both. .

In addition to the three constraints on the form of rules mentioned in Section I, we shall add two more. In what follows, we make the natural translation of => dependencies on the attributes of relations to the same relationship on variables. That is, if $R(X, Y, Z)$ is a relation' with X => Y, and there is a term $R(u, v$, w) in some rule, then we shall say u => v in that rule.

4. In no rule do we have z => z and y => z for $x \neq y$.

5. If in some rule we have x => y , and y is an argument of some nondatabase relation on the right, then x appears on the left side of the rule, and y does not.

Example 5: The rules from Example 1 satisfy the above conditions, if we take the dependency in the relation $E(X, Y)$ to be X => Y. Rule (5) does not hold if we instead assume Y => X. •

The Algorithm

The algorithm we shall consider is a simple modification of top-down expansion of the rule/goal tree. The change applies to the case in which we have just expanded a rule node, and there are one or more database relations among the goal children. Moreover' there is at least one dependency X => Y that applies to a goal node whose X-component is a constant x_0.

We consult the relation corresponding to that node for all the Y-values, say $\{ y_1, . . . , y_k \}$, that appear in tuples with X-value x_0. Suppose that the free variable corresponding to attribute Y is y. (Note that y must be free by condition (5) on the form of rules.) Then for each goal node that involves variable y , we create k children, and in the i^{th} child, the goal has y_i in place of y. This process of expansion is repeated for each free variable y that can be replaced by a set of constants. Note that by condition (4), if y can be expanded' there is a unique way to do so. In cases where more than one application of => is possible for a goal, we shall expand the variables such as Y at once, in all possible ways, so the original goal has children with these variables replaced by all possible combinations of constants.

The construction of the rule/goal tree continues, until all leaves are either

1. database relations,
2. rule nodes with empty right sides,
3. goal nodes that have been "expanded" by the rule above, but where the set of constants $\{y_1, . . . , y_k\}$ was actually empty, or
4. goal nodes corresponding to nodes of the rule/goal graph that have already been captured. These tree nodes will, in the remainder of the paper, be treated exactly as if they were

database relations, and the term "database relation" will be construed to include them.

If and when the tree construction terminates, we work up the tree, computing the relations for each node in the normal way.

It is important to note that this algorithm uses one specific "trick," which we call ***sideways information passing,*** to help convergence. There are other tricks we could use; for example, if a goal leaf is known to return an empty relation, then its parent will also return empty, which may lead to its parent returning empty, and so on. Also, we could use sideways information passing to enumerate the possible values for a free variable whenever that variable appears in a database relation.† These improvements on the algorithm might or might not be made in practice; the decision whether it is worthwhile to replace free variables by sets of constants is not clearcut. Unfortunately, these other variants of the top-down algorithm do not appear to have simple decision procedures that will tell us whether they converge for a given set of rules. The decision procedure we propose is, of course, a sufficient condition for the more powerful variants.

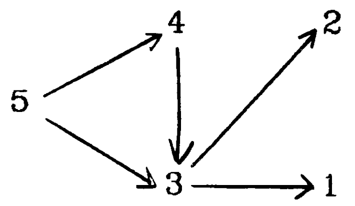


Fig. 3. Directed acyclic graph.

Example 6: Suppose we use the rules of Example 1 on the graph of Fig. 3. We shall assume that there is a => dependency from the first component of E to the second. In Fig. 4 we see the rule/goal tree for the goal $T(5, w)$, i.e., the tree of nodes accessible from 5. We use arbitrary names of the form yi for free variables corresponding to y in rule r_2. When a variable is bound to a set of constants by the sideways information rule, we show that set equated to the variable. The goal node $T(3, y2)$ is equivalent to the goal $T(3,$ y). We show a dotted line from the former to the latter, which may be interpreted in one of two equivalent ways.

† However, limiting sideways information passing to the direction of => arrows has some good intuition behind it. Since the values on the right are less than the values on the left of the =>, we might expect the set of values on the right corresponding to a given value on the left to be rather small. That would typically be true if the dependency were *EMP* => *MGR*, for example, but not if it were the other way around.

1. A subtree equivalent to that dangling from $T(3, y)$ will appear below $T(3, y2)$.
2. The system will identify the equivalence of the two goals, and the relation for $T(3, y2)$ will be copied from the relation for $T(3, y)$ when the latter is computed. This strategy is used, for example, in [MS], but it doesn't affect convergence of the algorithm, even though it may make some trees that would be infinite become finite. The reason is that when the tree would otherwise be infinite, there must be a pair of identical goals that are ancestor and descendant. For these nodes, we shall never be able to compute the desired relation working up the tree.†

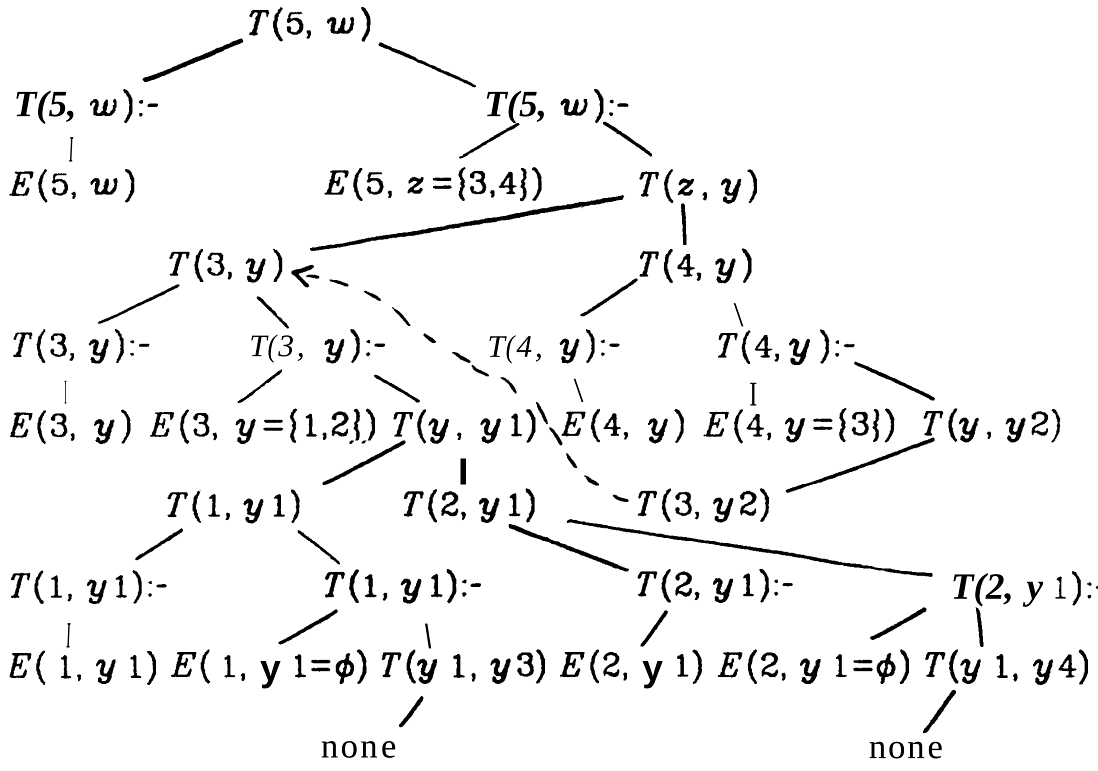


Fig. 4. Expansion of rule/goal tree with sideways information passing.

† Iterative bottom-up methods of evaluating such relations may work ([R], [HN], [U], e.g.), but bottom-up evaluation is often a far more expensive process than top-down expansion of the tree, and the use of these more powerful methods will not be considered here.

III. TESTING WHETHER THE TOP-DOWN ALGORITHM CONVERGES

In essence, given our five restrictions on the form of rules, very limited things can happen as we trace any path down the rule/goal tree. In particular, let us follow what happens to the arguments in a goal node. Suppose $A(x_0, y_0, z)$ is a goal, where x_0 and y_0 denote specific constant values, and z is a free variable. A rule head that unifies with this goal will bind one of its variables to x_0 and one to y_0. Condition (3) assures that the same variable could not be bound to both x_0 and y_0, or to one of these and to z.

Possibly, the right side of the rule has a term or terms with dependencies that make some variable(s) be strictly less than x_0 or y_0 (but not both). In that case, the sideways information feature of the algorithm of Section II will create descendant goals in which these variables are replaced, in all possible ways, by constants.

Thus, all variables appearing in the rule are either set equal to x_0 or y_0, are bound to values strictly less than one of these, are bound to z (the third argument of *A),* or are new free variables. The same, therefore, applies to the goal nodes that are children of the rule node (or grandchildren in the case that expansion of variables into sets of constants occurs). We shall see that the presence of free variables as arguments of goals does not affect convergence of the algorithm, so we need only trace the positions in which bound arguments occur.

Argument Mappings

It helps to express the way bound arguments are passed down the rule/goal tree by directed, bipartite graphs, which we shall call ***argument mappings.*** In these graphs, the two sets of nodes will be called the ***domain*** and ***range*** sets, and all arcs will go from the domain to the range. The domain and range sets are each *identified with a class, which is a predicate symbol with a superscript indicating which arguments are bound. For example, the class corresponding to the term $A(x_0, y_0, z)$ would be A^{bbf} .

The domain and range sets each have nodes for each bound variable in their class. If $A\ (x_1, \ldots, x_k)$ appears on the left of a rule and $B(y_1, \ldots, y_m)$ appears on the right, then there is an argument mapping from class A^q to B^p if the following conditions are met.

1. p makes an argument y_i bound if and only *if it is either equal to an argument that q makes bound, or $x_j \Rightarrow y_i$ for some variable x_j that q makes bound.

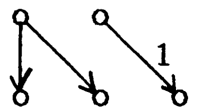
2. There is an arc from x_i to y_j labeled 0† if and only if x_i is bound according to q, and y_j and x_i are the same variable.
3. There is an arc from x_i to y_j labeled **1** if and only if x_i is bound by q and x_i => y_j.

According to condition (4) on the form of rules, there can be only one arc into any node. Since argument mappings only use nodes for bound variables, there will be exactly one arc into each node. Also, condition (5) tells us that there are no other => relationships; each one must be from a domain node to a range node.

Example 7: Consider the rule

$$A(x, y) \text{ :- } B(x, x, z), C(y, z)$$

and suppose that a dependency on C tells us y => z . Then the following is the argument mapping from class A^{bb} to B^{bbb}:



That is, the first two arguments of B each come from the first argument of A, while the third argument of B is related by => to the second argument of A. ▪

Argument mappings can be composed in an obvious way, and the result will be an argument mapping, although the labels on arcs may become larger than 1. The mappings still have the property that there is exactly one arc into each of the range nodes, and the domain and range sets of nodes are each identified with specific classes.

We can see a composition as representing a path in the rule goal tree. If the classes of the domain and range sets are A^* and B^p ,respectively then there is a path in some rule/goal tree from a goal node labeled A, with bound arguments where q indicates, to a goal node labeled B, with bound arguments where p indicates. If there is an arc from x to y, labeled 0, then argument y of the latter node is identical to argument x of the former. If the arc has label m > 0, then the value of y is at least m less than the value of

† We omit 0 labels on arcs, as a default.

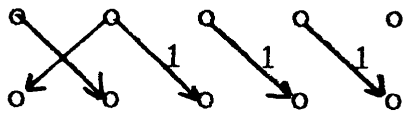
Fixpoints of Argument Mappings

We can now begin to see how argument mappings relate to the convergence of the algorithm of Section II. If arbitrarily long compositions of argument mappings exist, but the labels on their arcs remain bounded, then we can start the rule/goal construction off with bound arguments that exceed the bound on the labels. We can load the database with tuples guaranteeing that whenever we are forced by a => dependency to take a step downward, WE can step downward by 1. It is then possible to show that some path in the rule/goal tree grows forever, corresponding to the arbitrarily long sequence of compositions of argument mappings with limited arc labels.

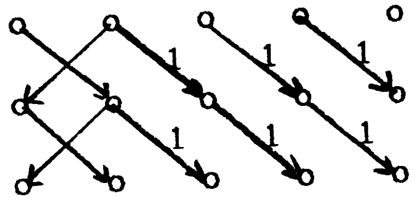
Conversely, if the only growing sequences of compositions of argument mappings also have growing labels on the arcs, then no matter what value we start our bound arguments with at the root of the rule/goal tree, we shall eventually find that some argument must become negative as we follow any path in the tree. However, there are no negative values in domains, which means that at some point along the path, we applied the sideways information transfer rule, but there were no values in the database far the variable on the right of the =>. In that case, the path in the rule/goal tree terminated after some finite length.

To test whether compositions of argument mappings have growing arc labels or not, we can ask about *fixpoints* of mappings. Consider a mapping whose domain and range sets are in the same class. Then there is an obvious correspondence between the domain and range nodes, so we can identify corresponding pairs of nodes and draw the mapping on a single set of nodes, with exactly one labeled arc entering each node. We call this graph the *collapsed graph* for the mapping. Then this mapping is said to have a fixpoint if there is an assignment of numerical values to the nodes such that if there is an arc labeled m from node v to node u, then the value of u is exactly m less than the value of v .

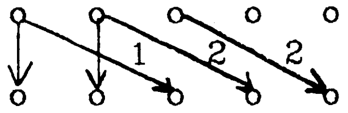
Example 8: Consider the following argument mapping, which we assume goes from some class to the same class.



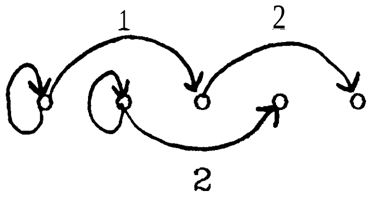
We can compose it with itself as follows:



The result of this composition, as a bipartite graph, is



We may identify the nodes in a column, and show the composition as a collapsed graph:



■

Lemma 1: The following three conditions are equivalent far the compositions of any finite set of argument mappings.

a) There is an infinite sequence of mappings $\mu_1, \mu_2, \ldots$ and a constant c such that for all i, the composition $\mu_i \circ \mu_{i-1} \circ \cdots \circ \mu_1$ is legal (the class of the domain of μ_j equals the class of the range of μ_{j-1} for all j), and in its argument mapping, no arc has label greater than c .

b) There is an argument mapping with a fixpoint.

c) There is an argument mapping whose collapsed graph has no cycle with a positive sum of edges.

Proof:

(c) implies (b). As each node has exactly on entering arc, all collapsed graphs are collections of cycles, with trees fanning out from some of the nodes in the cycles. Let k be the largest weight (sum of edge labels) of any path in the graph. Since there are no positive weight cycles, k is finite. Assign k to each node that is in a cycle. Then, work down from the roots of the trees, assigning to each node the value of its parent minus the label of the arc entering that node. For example, the nodes in the collapsed graph of Example 8 would be assigned values 3, 3, 2, 1, 0, from the left.

(b) implies (a). A fixpoint of any mapping is also a fixpoint of that mapping composed with itself any number of times. But the fixpoint has specific values, whose differences are bounded, so if the arc labels grow as we compose the mapping with itself many times, we cannot meet the condition of a fixpoint, that the values of the nodes differ by the label of the arc between the nodes. Thus, if there is a mapping with a fixpoint, there is an arbitrarily long sequence of compositions with a bound on arc labels.

(b) implies (c). Suppose we have a mapping with a fixpoint, but its collapsed graph has a positive-weight cycle, involving nodes $n_1 \to n_2 \to \cdots \to n_k \to n_1$. Then if we compose the mapping with itself k times, the resulting mapping has the same fixpoint, but the collapsed graph for the mapping has a positive-weight loop at each of the nodes $n_1, \ldots, n_k$, an impossibility for a mapping that has a fixpoint.

(a) implies (b). Suppose such an infinite sequence and constant c exists. Consider what happens if we start with arguments $(c, c, \ldots, c)$ and apply the mappings $\mu_1, \mu_2, \ldots$ in turn. First, note that some class A must occur an infinite number of times as the range class of these compositions. As the composition of mappings at each step has no arc of label greater than c , no value ever gets below 0. Thus, we can find two steps at which the range class is A, and the values of the arguments are the same. The mapping consisting of the composition of the μ's between these two steps evidently has a fixpoint. ▪

An Algorithm to Test the Conditions of Lemma 1

Condition (c) is easiest to test. We iteratively find the set of argument mappings that are compositions of the given set of mappings, S_0. However, we shall not distinguish between arc labels greater than 0, since we care only whether a positive-weight cycle exists, not what the exact weight of the cycle is. The details of the iteration are given in Fig. 5.

Complexity of Testing Condition (c)

Let there be r argument mappings and let m be the maximum size of a domain or range set. Then r is also an upper limit on the number of classes. Hence the number of different mappings, with labels above 1 replaced by 1, is no more than $r^2(2m)^m$; the ractor r^2 represents the r choices for the domain and range classes, while $(2m)^m$ represents the fact that each of the m or fewer range nodes has an entering arc from any of at most m domain nodes, each arc labeled 0 or 1. Thus, the number of mappings ever marked "new" is no greater than this quantity. For each "new" mapping, we look at no more than r V's, spending $Q(m)$ time on each pair. Thus, a bound on the running time of the algorithm is

```
s := $S_0$;
mark all members of S "new";
while changes to S occur do begin
     for all new $\mu$ in S do begin
          mark $\mu$ "old";
          let $A$ be the range class of $\mu$;
          for all mappings $\nu$ in $S_0$ with domain class A do
               begin
                    $\rho := \nu \circ \mu$;
                    let $\sigma$ be $\rho$ with arc labels greater
                         than 1 replaced by 1;
                    i f a i s not in S then begin
                         mark $\sigma$ "new";
                         $S := s \cup \{\sigma\}$
                    end
               end
     end
end
```

Fig. 5. Algorithm to test Lemma **1.**

$O(r^3(2m)^{m+1})$.

This figure looks formidable, but it is typical for ***m,*** which corresponds to the number of arguments of predicates in the logical rules, to be small, perhaps limited by 3 or 4. If we regard m as a constant, the algorithm is actually polynomial in the size of the input (the argument mappings written out). Moreover, in typical cases we do not expect to generate anything like the full set of possible mappings, nor shall we typically generate only one new mapping per iteration of the loop; thus the actual time consumed will probably be acceptable. In case ***m*** must be considered a variable, we can still show the following.

Theorem 1: Deciding whether condition (c) of Lemma **1** holds is in PSPACE.

Proof: We shall give a nondeterministic polynomial space algorithm to decide; this can be converted to a deterministic polynomial space algorithm by Savitch's theorem.7 Guess a starting class *A* and a mapping with domain class *A* to get an initial mapping and range class for that mapping. Repeatedly guess a mapping whose domain equals the range of the current mapping, and compose the

† See [HU] for notions of nondeterministic algorithms, Savitch's theorem, etc.

current mapping with the chosen one to get a new current mapping. If the range class of the current mapping is A, **check** whether the collapsed graph is free of positive-weight cycles, and say "yes" if so.

Throughout this process we have only to record one mapping at any time, and no arc labels above 1 are ever needed, since they may be replaced by 1. Thus, the required space is of the same order as that needed to write the largest argument mapping in the input, and is therefore linear in the input size. .

A Capture Rule for the Algorithm of Section II

We shall now describe the conditions under which we can capture a set of nodes of the rule/goal graph using as substantiation the algorithm of Section II. First, we need to modify the construction of the rule/goal graph slightly to accommodate the sideways information passing inherent in that algorithm. The change occurs when we consider the superscript on a goal node A^p that is the predecessor of some rule node r^q, where A is not a database relation. In the modified graph, p will make a variable x bound not only if q makes x bound, but also if there is some other variable y such that q makes y bound, and there is a dependency $y \Rightarrow x$ coming from some term on the right side of r. That term could have the same predicate A.

Example 9: Consider the rules

$$r_1\colon A(x, y, z) \text{ :- } B(y, z, w), C(x, w)$$
$$r_2\colon A(x, y, z) \text{ :- } B(z, x, w), C(y, w)$$
$$r_3\colon B(x, y, z) \text{ :- } A(x, x, y), A(y, y, z)$$

We shall assume a dependency from the first argument of C to the second.

The modified rule/goal graph, or at least the portion that is required to capture A^{bbb}, is shown in Fig. 6. The important point about the modification is that B^{bbb}, rather than B^{fbb}, is a predecessor of r_1^{fbbb} and r_2^{fbbb}. The reason is that the dependency on C tells us $x \Rightarrow w$ in r_1 and $y \Rightarrow w$ in r_2. •

The capture rule allows us to capture any set S of nodes provided

1. All predecessors of S are either in S or already captured, and
2. When the nodes of S are converted to argument mappings in a manner to be described, the set of mappings does not satisfy the conditions of Lemma 1; that is, every composition of the mappings from a class to the same class has a positive-weight

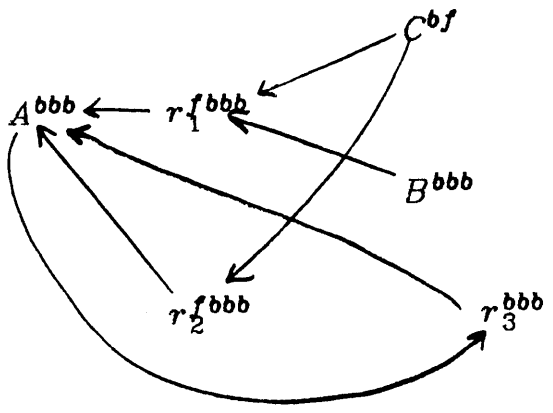


Fig. **6**. Modified rule/goal graph.

cycle.

For example, we may let

$$S = \{A^{bbb}, r_1^{fbbb}, r_2^{fbbb}, B^{bbb}, r_3^{bbb}\}$$

in Fig. 6, and condition **(1)** will be satisfied since C is a database relation, and therefore C^{bf} can be captured by elementary means [U].

Conversion of Rules to Argument Mappings

The conversion process is as follows. Suppose A^q is a node of S and r^p is one of its predecessors (i.e., r is a rule with head A). Let B be a nondatabase predicate appearing on the right side of r, and suppose that corresponding to this occurrence of B on the right is the predecessor B^t of r^p.† Then there is an argument mapping from class A^q to B^t. For each node in the range set, say corresponding to variable x, there is an entering arc from

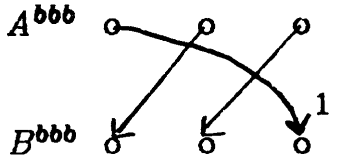
a) The node of the domain set corresponding to x if x is bound according to q; in this case the arc has label 0, or

b) The node corresponding to y of the domain set if y is bound according to q and there is a dependency $y => x$ due to some other predecessor of r^p.

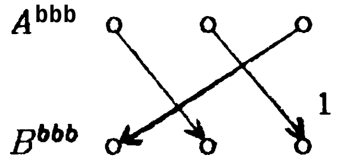
Note that the definition of the modified rule/goal graph guarantees that all and only the variables made bound by t will satisfy (a) or (b).

Example 10: For the set S consisting of all the nodes in Fig. 6 except for C^{bf}, we get the mapping

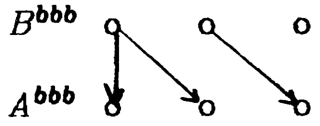
† Recall that if B^t is already captured, then this occurrence of B is a "database relation."



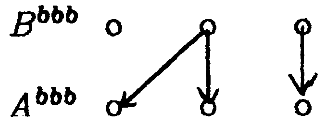
from the path through r_1^{fbbb} from B^{bbb} to A^{bbb}, and we get the mapping



from the path through r_2^{fbbb}. For paths through r_3^{bbb}, we must realize that A^{bbb} represents both terms on the right of r_3, $A(x, x, y)$ and $A(y, y, z)$. For the former, we get the mapping



and for the latter we get



■

Validation of the Capture Rule

We can now show that the capture rule given above exactly characterizes the conditions under which the algorithm of Section II substantiates the capture of set of nodes S.

Theorem 2: Suppose we have a set of nodes S of a rule/goal graph meeting condition (1) of the capture rule. Then the algorithm of Section II attempts to construct an infinite tree when started with some database and with some goal $A(x_1, \ldots, x_k)$ corresponding to some node A^q in S (i.e., the x_i's are bound exactly when q says they should be) if and only if the conditions of Lemma 1 are met.

Proof:

If: Suppose that a sequence $\mu_1, \mu_2, \cdots$ and constant c as described in condition (a) of Lemma **1** exist. Let A^q be the domain class of μ_1. Then we shall show that for some database there is an infinite sequence of nodes constructed in the rule/goal tree with root $A(a_1, \ldots, a_k)$, where a_i is either c, if q makes argument x_i bound, or $a_i = x_i$, a free variable, if not. The database we have in mind is any one where for each dependency $X \Rightarrow Y$ all relations with attributes X and Y have, for any pair (i, j) with $c \geq i > j \geq 0$, a tuple with X-value i and Y-value j.

We shall construct a path in the rule/goal tree that has nodes corresponding to each of the mappings in the given sequenae. In particular, corresponding to μ_n will be a goal node $B(y_1, \ldots, y_m)$, if the range class of μ_n is B^p for some p . We shall call this node N_n. The y_i's will be as follows. Let ν be the composition $\mu_n \circ \cdots \circ \mu_1$. If p says that argument i of B is free, then y_i could be any variable. If argument i is bound, then look at the arc entering the range node for position i in ν. If that arc has label d, let $y_i = c - d$. By our assumptions, $d \leq c$, so no constant beaomes negative.

As a special case, let the goal node at the root be N_0. Then we can perform an induction on n to construct each of the nodes N_i, $i \geq 1$. Suppose we have constructed the node $N_n = B(y_1, \ldots, y_m)$, and we want to construct the node $N_{n+1} = C(z_1, \ldots, z_h)$. Then there must be some rule r with B on the right and a term C on the left, such that μ_{n+1} was constructed from r to reflect the transition from B on the left to this occurrence of C on the right. There can be only free variables appearing once among the arguments of the left side of r by conditions (1)—(3) regarding the form of rules. Thus, the left side unifies with the goal at N_n , and it has a child corresponding to this occurrence of C on the right of r.

This child has some occurrences of constants that are copies of constant arguments of B. By the way μ_{n+1} was constructed from r, z_j will be a copy of y_i if and only if μ_{n+1} has an arc with label 0 from the domain node corresponding to y_i to the range node corresponding to z_j.

If μ_{n+1} has no arcs labeled **1** we are done; the child corresponding to C serves as N_{n+1}. Suppose there is some arc, say to the range node for z_j, that has label **1.** Then there must be some term D on the right of r that has variables y_i and z_j, and $y_i \Rightarrow z_j$. Our database relations, of which D is one, have been constructed so that if y_i has constant value e in N_n, then there is in D a tuple with e in the component for y_i and e **-1** in the component for z_j. Thus, there will be a grandchild of N_n in which z_j has the value e **-1.** By the inductive hypothesis, in the composition of $\mu_n \circ \cdots \circ \mu_1$, the arc into y_i has some label d , so in $\mu_{n+1} \circ \cdots \circ \mu_1$ the arc into z_j has label **d** +1. Thus, e $=$ **c** $-d$ and **e-l** = c -(**d** +1), so the inductive hypothesis about the value of z_j is proven. Note that **d** +1 $\leq$ c must hold by condition (a) of Lemma

1, so we can never face a situation where **e -1** does not exist in the database.

If there are several arcs labeled 1 in μ_{n+1}, then we can find a grandchild of N_n that has the proper value for each argument that is the target of one of those arcs. This grandchild is N_{n+1}. We have now completed the induction on n and see that an infinite path in the rule/goal tree exists.

Only *if:* The converse also requires some attention to details, but the ideas are the same as for the first part of the proof, and we shall only sketch them. Suppose we are given an infinite path in the rule/goal tree. We look at the goal nodes along the path, skipping those whose children are also goal nodes; that happens only when sideways information transfer occurs. Then we can construct a sequence of argument mappings corresponding to this sequence of goal nodes in the obvious way. Suppose c is the largest constant appearing among the arguments at the root of the tree (if there are no constants then we may take $c = 0$).

Every arc of label 1 in an argument mapping corresponds to a situation in which a constant at some goal node is set equal to a value that is strictly less than a certain constant at the previous goal node. Thus, we may prove by an easy induction down the path that no arc in the composition of the first n argument mappings has a label higher than c, for if it did, then in the rule/goal tree there would be a negative constant. Thus, condition (a) of Lemma **1** is seen to hold. ▪

Complexity of Deciding Whether the Capture Rule Applies

As discussed in [U], we can assume that we apply the capture rule only to minimal sets S, and these can be found in time that is linear in the number of nodes of the rule/goal graph by finding strong components in the reverse graph.

Let there be s rules, at most t terms on any right side, and let the maximum number of arguments in a predicate be m. The number of argument mappings constructed from these rules is no more than $st2^m$, that is, no more than the number of rules, multiplied by the number of terms on the right of each rule, and the number of combinations of bound and free variables among the arguments of the left side of the rule.

By the reasoning given prior to Theorem 1, we may test condition (c) of Lemma 1, which is equivalent to testing condition (a), in time $O(s^3t^32^{3m}(2m)^{m+1})$, or $O(s^3t^3(16m)^{m+1})$. Suppose we let n be the size of the input, i.e., the rules written down. Then surely s,

t , and m are all no larger than n. Hence, we may observe the following.

Theorem 3: If m, the maximum number of arguments in a predicate, is constant, then we can find all applications of the capture rule of Section II in polynomial time, If not, the problem is in EXPTIME, that is, time 2 to a polynomial. ▪

IV, APPLICATION TO RULES WITH FUNCTIONS IN ARGUMENTS

The ideas of the previous sections can be applied in certain situations to rules in which there are functions. Without loss df generality, we shall speak of a "cons" function used as a list former; $[x|y]$ denotes the list with head x and tail y; [] denotes the empty list.

In [U], there is a discussion of an efficient test for a subset of the cases in which we can capture a set of nodes using as a substantiation the straightforward top-down construction of the rule /goal tree. This capture rule applies to collections of rules in which all the cons operators appear on the left sides of rules The motivation behind the capture rule is that in some cases we could detect that for a particular argument, say the i^{th}, whenever we had a goal node $A(x_1, \ldots, x_k)$ in the rule/goal tree, with a descendant $A(y_1, \ldots, y_k)$, then y_i could be proved to be a proper sublist of x_i (both these arguments would be bound variables, of course). The test for this condition is polynomial in the size of the set captured; that is, if there are r nodes, and m is the maximum number of variables in the rule or in a predicate corresponding to any node, then the algorithm is polynomial in r and m .

In [N], a similar idea is proposed, but it is more general, in that convergence of the top-down algorithm can be proved by finding any set of the arguments of A, say $\{i_1, \ldots, i_m\}$ such that in any ancestor-descendant situation described in the previous paragraph' there is at least one i_j for which y_{i_j} can be shown a proper subpart of x_{i_j}; j can vary for different ancestor-descendant pairs. The algorithm proposed in [N] for detecting such situations involves considering all possible paths from A to A in the rule/goal graph, as well as all possible subsets of arguments' so it can be exponential in both r and m.

Example 11: The following example taken from [N] shows how looking for subsets of arguments that together guarantee convergence can be an advantage. The merger of two lists to form a third can be expressed as:

$merge(x, [], x)$:-
$merge([], y, y)$:-
$merge([a|x], [b|y], [a|z])$:- $a \leq b$, $merge(x, [b|y], z)$
$merge([a|x], [b|y], [b|z])$:- $a > b$, $merge([a|x], y, z)$

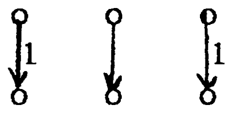
Certainly, if the third argument is bound, then whenever ***merge*** calls itself recursively, the third argument is a proper subpart of its initial value. But in addition, if the first and second arguments are both bound, then one or the other (but not both) will become a proper subpart of its initial value, so we are able to oapture not only $merge^{ffb}$, but also merge ^{bbf} with a capture rule substantiated by top-down expansion of the rule/goal tree. ▪

Of course, in more general sets of rules, involving the mutual recursion of many predicates, and the shifting of arguments from one position to another, detecting all such opportunities can be time consuming — exponential in the size of the problem as we mentioned. Fortunately, the methods of the previous section carry over to this case, and we can adapt them to provide a test that is exponential in m but polynomial in r. Moreover, sinoe our algorithm looks at only those argument mappings that it is forced to look at, while [N] looks at all possible paths through the set of nodes being captured regardless of whether or not they are generating different argument mappings, we expect our approach to be far more efficient in practice.

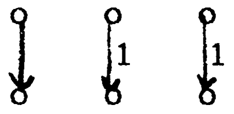
From rules with structured arguments on the left we can construct argument mappings that are analogous to those used in Theorem 2. Here, an arc labeled 1 from x to y corresponds to the notion that y is a proper subpart of x, rather than being numerically less than x. An arc labeled 0 means that y is x itself. As long as there are no structured arguments on the right, there till be an argument mapping in the sense of Section ITT between any rule head and any nondatabase term on the right. That is, each argument appearing on the right side, whose value is bound, will either be a copy of a bound variable on the left, or a subpart of one of these. The test of Theorem 2 will then answer the question of [N]: will a given set of rules lead to infinite recursion when the root goal has a certain set of arguments bound.

The proof of the "if" portion can be modified so that instead of starting with constant c for each bound argument at the root of the rule/goal tree, we start with a constant list that is a complete tree of depth c , i.e., the head and tail of the list are each complete trees of depth c -1. Moreover, Theorem 3 will apply; if there is a constant upper bound on the number of arguments in a term, then the test is polynomial in the size of the data.

Example 12: Strictly speaking, the rules of Example **11** violate our constraint that the arguments on the right be without structure. However, it is easy to see that the argument *[b |y]* in the right side of the third rule and [a |*x*] in the fourth are copies of arguments on the left side of those rules, so we can use arcs of label 0 to reflect the copying of arguments in each case.† Then the argument mappings corresponding to rules **(3)** and **(4)** of Example 11 are (assuming all three arguments are bound):



and



respectively. The class is *merge*^{bbb} for each domain and range, of course.

It is easy to see that any composition of these two mappings will have a positive-weight cycle, so top-down expansion is guaranteed to converge; sideways information passing as described in Section II is not needed, and in fact doesn't make sense, since we are using the structure of the list arguments to force progress downwards, rather than using any numerical inequalities. Notice also that if the third argument of *merge* is free rather than bound, the two mappings are changed only by having the rightmost column disappear. The remaining mappings still have no composition without a positive-weight cycle, so we can capture *merge*^{bbf} , as [N] observed. Similarly, if only the third argument is bound, we have argument mappings consisting only of the rightmost column; this set has no composition without a positive-weight cycle, so we can capture *merge* ^{ffb}. .

† Incidentally, it is possible to avoid structured arguments on the right if we define a "car" function that extracts the first element of a list. Then we could write: *car*([*a*|*y*], *a*):-, *merge*([*a*|*x*], *y*, [*a*|*z*]) :- *a* ≤ *b*, *merge*(*x*, *y*, *z*), *car*(*y*, *b*), and *merge*(*x*, [*b*|*y*], [*b*|*z*]) :- a > b, *merge*(*x*, *y*, z), *car*(*x*, u).

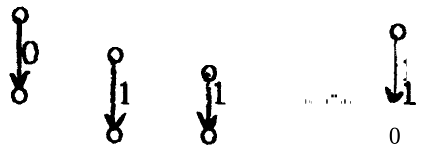
V. AN INTRACTABILITY RESULT

While the capture rule of Section III can be applied in polynomial time if the maximum number of arguments in any predicate is fixed, the same problem turns out to be intractable if we let the number of arguments be part of the input. We begin by showing the following intractability result, and then show how the difficulty of deciding properties of argument mappings translates into problems about logical rules.

Lemma 2: It is NP-hard to determine, given a set of argument mappings, whether some composition of those mappings is free of positive-weight cycles.

Proof: Our reduction is from 3SAT. Suppose we are given an instance of 3SAT in which the variables are $x_1, \ldots, x_v$, and there are f factors, $F_1, \ldots, F_f$. Initially, let us construct a set of argument mappings in which there are v +1 classes, which we denote $A_0, A_1, \ldots, A_v$, and each class has f +1 arguments, which we number 0, 1, . . . , f . Variable i corresponds to A,, and factor j corresponds to argument j; A_0 and argument 0 play special roles as we shall see.

There will be one argument mapping from A_0 to A_1, in which all arcs have both head and tail corresponding to the same argument, as:

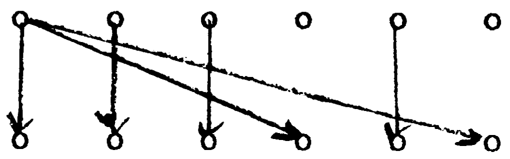


All arcs but that for argument 0 have label 1.

Next, for each variable x_i there are two argument mappings, one corresponding to x_i and the other to $\bar{x}_i$. Each takes class A_i to class A_{i+1} (to class A_0 if $i = v$). All arcs in these mappings have label 0. In the mapping for x_i, range node 0 has an arc from domain node 0. Range node j, for $j \geq 1$, has:

1. An arc from domain node 0 if x_i is a term in F_j.
2. An arc from domain node j otherwise.

The pattern is suggested below.



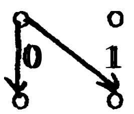
For the argument mapping corresponding to $\bar{x}_i$, the same construction is used, but the arc comes from domain node 0 if and only if $\bar{x}_i$ is in F_j.

Consider a composition of mappings that takes us once from A_0 to A_1 to $\cdots$ to A_v and back to A_0. The composition uses one of the two possible mappings for each variable, and therefore corresponds in a natural way to a truth assignment. If that truth assignment is satisfying, then every range node has an arc that comes from domain node 0, with label 0. But if some factor F_j is not satisfied, then range node j has an arc from domain node j, and the label is 1. Thus, the existence of a composition taking us from A_0, once around the cycle, and back to A_0, with no positive-weight cycle, is equivalent to the existence of a satisfying assignment.

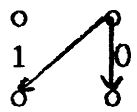
It is also easy to show that if we start the cycle at some class other than A_0, then the absence of a positive-weight cycle implies that, had we started at A_0 and made the same choices of mappings, we would again have gotten no positive-weight cycle.

However, there is a bug in this construction: we must consider compositions that go around the cycle of classes more than once. In particular, we could go around once, destroy the loops of weight 1 for some of the arguments, using one truth assignment, then go around a second time and kill the rest with another truth assignment. It would be nice if we could arrange that every time we went more than once around the cycle of classes there was guaranteed to be a positive-weight cycle, but that does not seem possible. Fortunately, we can do something different; we can guarantee that there is a positive-weight cycle whenever we go around the cycle more than once and we pick a different truth assignment for one or more variables.

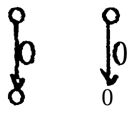
To this end, for each variable x_i we introduce two more arguments for all classes. In the mapping for x_i, the arcs for these arguments are



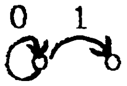
while in the mapping for $\bar{x}_i$ they are



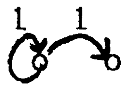
In all other mappings, the arcs for these two arguments are



If the first time around the cycle, the mapping for x_i is chosen, then the collapsed graph for the composition is



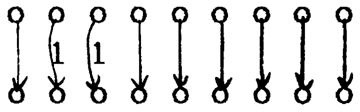
If some subsequent time around the cycle, the mapping for $\bar{x}_i$ is chosen, then the collapsed graph becomes



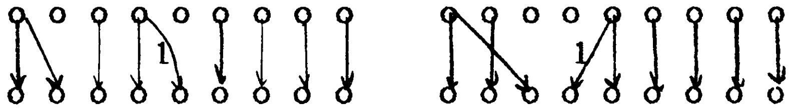
and now, whatever mappings are chosen, there will always be a loop of positive weight at the first of these two arguments.

Symmetrically, if the mapping for $\bar{x}_i$ is chosen the first time around the cycle, and later the mapping for x_i is chosen, there will always be a positive-weight loop at the second argument. Now, we know that if there is a composition of argument mappings with no positive-weight cycle, it cannot involve both the mappings for x_i and $\bar{x}_i$. Thus, if there is no loop of positive weight at the node for any of arguments 1 through f, then the truth assignment corresponding to the mappings actually chosen must cover all factors and is therefore a satisfying assignment. ▪

Example 13: To give an illustration, we shall consider the boolean expression $(x_1 + x_2)(\bar{x}_1 + x_3)$, which strictly speaking is not an instance of 3SAT. The mapping from A_0 to A_1 is

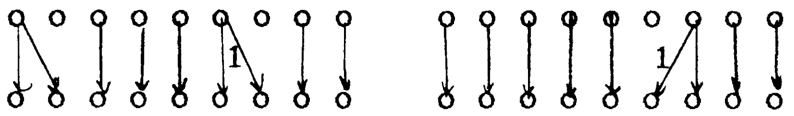


The two mappings from A_1 to A_2 are



The one on the left corresponds to x_1 and represents the fact that x_1 covers the first factor but not the second; the mapping on the right is for $\bar{x}_1$ and expresses the fact that $\bar{x}_1$ covers the second term only.

The mappings from A_2 to A_3 are



while the mappings from A_3 to A_0 are

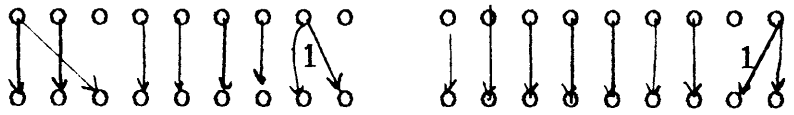
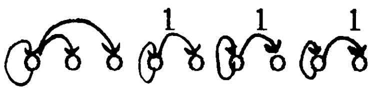


Figure 7 shows the paths in the composition that represents the truth assignment $x_1 = x_2 = x_3 = 1$. Its collapsed graph,



is seen to have no positive-weight cycle, because the assignment chosen is satisfying. ▪

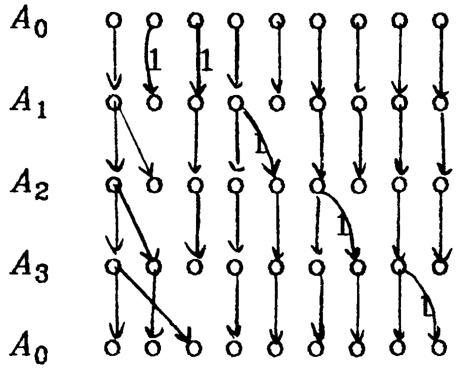


Fig. 7. Composition of argument mappings.

Theorem 4: The question of deciding whether a given set of logical rules obeying conditions (1)—(5) permits the capture of a given set of nodes by the capture rule of Section III is NP-hard.

Proof: By Lemma 2, all we must do is show that given any set of argument mappings, we can construct a set of rules yielding those and only those mappings. Of course, the size of the set of rules must be polynomial in the size of the set of argument mappings, and in fact it will be linear.

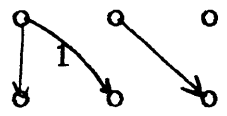
For each class A we shall have a predicate A, and the node we shall attempt to capture is $A^{bb \cdots b}$. Suppose there is am argument mapping from class A to class B, with k domain nodes and m range nodes. Then there will be a rule of the form

$$A(x_1, \ldots, x_k) \text{ :- } B(y_1, \ldots, y_m), \text{ other terms}$$

If there is an arc from domain node i to range node j, labeled 0, then $y_j = x_i$. If there is such an arc, but it is labeled **1,** then y_j is a new variable, and among the "other terms" alluded to above, we introduce a new database relation $C(x_i, y_j)$, with a dependency $x_i => y_j$.

When constructing argument mappings from rules, this rule yields only one mapping, and it is exactly the mapping from which it was constructed. It is easily seen to satisfy conditions (1)—(5) on the form of rules. ▪

Example 14: From the argument mapping



we construct the rule

$$A(x_1, x_2, x_3) :- B(x_1, y_2, x_2), C(x_1, y_2)$$

with dependency $x_1 => y_2$ enforced by C. ▪

VI. SOME GENERALIZATIONS

In this section we shall briefly cover relaxation of our conditions **(1)** and (3) on rules.

Duplicate Arguments on the Left Sides of Rules

Condition (3), that no variable appear twice among the arguments of any one term on the right side of a rule, is fairly straightforward to eliminate. First, let us observe where the test for convergence might break down when rules can have left sides like $A(x, x)$:-. There is a possibility that, although an infinite sequence of argument mappings whose compositions have no arcs of weigh: above c exists, when we try to convert this sequence into an infinite path in the rule/goal tree, we fail because $A(x\ , x)$ does not unify with a goal $A(y_0, z_0)$. Here, y_0 and z_0 are two distinct bound variables.

A simple solution is to replace terms with identical arguments by terms with different predicates, in fewer variables, so that no term whatsoever has duplicate arguments. The merger of arguments must be propagated from the left sides of rules to the right. sides, as discussed in the lemma that follows.

Lemma 3: Every set of rules satisfying conditions (1) and (2) is equivalent to one in which, if predicate *A* appears with duplicate arguments on the left of some rule then *A* does not appear on the right of any rule.

Proof: For each predicate symbol A and each partitian π of the arguments of A , we introduce a new predicate symbol A_π. A_π has one argument for each block of π. Suppose rule r has A *on* the left, and any two arguments of A that are the same variable are in the same block of π. Then we generate a rule r_π with A, *on* the left, and each term B on the right replaced by B_ρ, where ρ groups two arguments into a block of the partition only if they are the same variable in term B, or their variables appear in arguments of A that are in the same block of π.

So that we can get, in the modified set of rules, the same answer to any query that we wold get in the originall rules, we retain the original predicate symbols, and for each one we introduce rules of the form

$$A(x_1, \ldots, x_n) :- A_\pi(y_1, \ldots, y_m)$$

for all partitions π. Here, each y_i corresponds to the i^{th} block of π, and $x_j = y_i$ if and only if j is in block i of π. ▪

Example 15: Consider the rule

$$r: A(x, x, y, z) :- B(x, y, z, w), C(w, w, x)$$

For partition {12, 34}, i.e., the partition that equates the first two arguments and equates the last two arguments, we get the rule

$$r_{12,34}: A_{12,34}(x, Y) :- B_{1,23,4}(x, y, w), C_{12,3}(w, x)$$

We use y as a representative for the block 34, which equates y and z.

The partition {13, 24} cannot be used with rule r; the reason is that the left side of r equates arguments **1** and 2 of A, *so* these positions cannot appear in different blocks of a partition.

As another example, suppose we were given the query $A(a_0, w, a_0, b_0)$. One of the starting rules

$$A(x, x, y, z) :- A_{12,3,4}(x, y, z)$$

unifies with this goal, producing the goal $A_{12,3,4}(a_0, a_0, b_0)$. However, this goal will produce only a subset of the relation returned by the starting rule

$$A(x, y, x, z) :- A_{13,2,4}(x, y, z)$$

and its first goal, $A_{13,2,4}(a_0, w, b_0)$.

On the other hand, the starting rule

$$A(x, x, y, y) :- A_{12,34}(x, y)$$

does not unify with the goal, because the last two arguments, a_0 and b_0, which we presume are different constants, cannot be unified. •

Theorem 5: We can decide whether or not the algorithm of Section II will lead to infinite loops even if terms are allowed to have duplicate arguments. If the number of arguments in predicates is bounded, then this test can be performed in polynomial time; if not, then the decision may take exponential time.

Proof: We modify the set of rules as in Lemma 3. If the number of arguments of predicates is bounded, then the number of rules is multiplied by some constant factor. We already know we can use the argument mapping test of Lemma 1 to decide whether there is an infinite loop with root A, for any partition π. Thus, given a query with predicate symbol A, we have only to find which start rules unify with the query and determine, using condition (c) of Lemma 1, whether the corresponding A_π's can lead to infinite loops. •

Constants in Rules

Constants in rules are different from bound variables in the sense we have been using the latter term. Constants appearing in rules never change, so their particular values may be used when deciding whether a capture rule may be applied. In contrast, the bound variables will always have a particular value when we use the substantiation algorithm for a capture rule, but in planning what strategy to use, we may not assume we know the value of a bound variable. In fact, the algorithm used to substantiate the capture rule may involve the same calculation with a variety of different values for the bound variable.

When we allow constants in rules, we come up against the same problem that we face with duplicate arguments on the left: infinite sequences of argument mappings may not translate into infinite paths in the rule/goal tree, because unification with a particular constant is impossible. When selecting our database to allow the infinite path, we need not choose only tuples with pairs $(i, i-1)$ in situations where the second component is constrained to be less than the first. Rather, we can arrange that all pairs (i, j), where $j < i$, appear in these relations.

However, that is not sufficient if we are faced with a situation where we have a constant, say 1, in a rule, and in the sequence of argument mappings there are inequalities that imply this argument must be at least 2. The following example illustrates the problem.

Example **16**: Suppose we have the following rules, where some terms have been elided, and symbol $\bar{x}$ denotes a variable that (because of the elided terms and dependencies that we shall not state) is forced to be strictly less than x.

$A(x, y)$:- $B(x, \bar{x}), \ldots$
$B(x, \ldots$:- $C(x, \bar{y}), \ldots$
$C(x, \ldots \ldots D(\bar{x}, x), \cdots$
$D(z, 1)$:- $\cdots$

Figure 8 shows the composition of the three argument mappings implied by the first three rules. The dashed lines, labeled i, among nodes on one level represent the fact that that the value of the argument at the tail must exceed the value of the head by at least i. We also see absolute lower limits on the value of certain nodes, that may or may not be implied by a dashed arc. In particular, the second argument of D is forced to be at least 2, because it is identical to the first argument of C, which in turn was forced to be at least 2 greater than the second argument of C. Thus, the rule for D cannot be applied after the three rules above it are applied, **but if we** simply constructed argument mappings as in Section III, **we would** have no way of knowing that. .

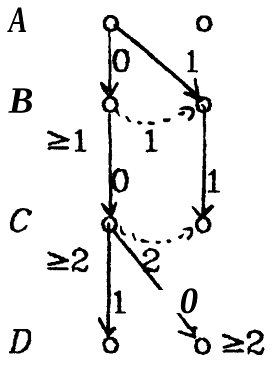


Fig. **8.** Composition of argument mappings with inequality constraints.

We therefore propose the following modification to the argument mapping construction of Section III. Include in the collapsed graph:

1. Dashed arcs, with labels.
2. Lower bounds for the nodes.

The meaning of these two additional notations is as described in Example 16.

Formally, suppose we have a collapsed graph G with set of nodes $v_1, \ldots, v_m$, with a collection of solid and dashed arcs and **lower bounds for nodes.** Also, suppose that there is an argument mapping μ from $\{v_1, \ldots, v_m\}$ to range node set $\{w_1, \ldots, w_n\}$. Let H be the collapsed graph for the composition of the mapping represented by G with the mapping μ. Then in H there is a dashed arc from w_i to w_j with label k if either:

1. There is some node v_p with an arc in μ labeled 0 to w_i and an arc to w_j labeled 1; in this case, $k = 1$.
2. There are nodes v_p and v_q, with a dashed arc in G from v_p to v_q labeled s , an arc in μ labeled 0 from v_p to w_i, and an arc labeled r from v_q to w_j. Here, r is 0 or 1, and $k = r + s$.

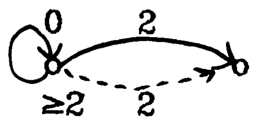
Note that if μ has an arc labeled 1 entering w_i, then w_i will not be the tail of a dashed arc in H. This observation reflects the fact that if the argument corresponding to w_i is only constrained to be strictly less than the value of some other argument, then the value of w_i has no lower limit (except 0, since all values are assumed nonnegative).

The lower bound on a node w_i is determined as follows.

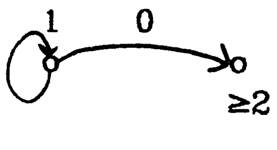
1. If there is a lower bound r on some node v_p in G, and there is an arc in μ labeled 0 from v_p to w_i, then r is a lower bound on w_i.
2. If w_i is the tail of a dashed arc labeled s in H, and r is a lower bound on the node at the head of this arc, then $r+s$ is a lower bound on the value of w_i.

Of course, we need only record the highest of several lower bounds on a node. It can be checked that no dashed cycles ever result from this construction.

Example 1'7: The collapsed graph for the composition of the first two mappings in Fig. 8 is:



and the collapsed graph for the composition of all three is:



■

We may also attach to the domain nodes of the argument mappings themselves an indication if that node represents a constant. Thus, in Example 16, the second domain node for the mapping from D would have the associated constant value 1. When composing argument mappings, we do not allow the composition if a constant node must be matched with a node whose lower bound exceeds that constant. If we do so, then we claim that all legal compositions of argument mappings with the additional information described above will correspond to paths in the rule/goal tree. Thus, we shall claim without detailed proof the following theorem.

Theorem 6: We can decide whether or not the algorithm of Section II will lead to infinite loops even if rules are allowed to have duplicate arguments and constants. If the number of arguments is bounded and there is bound on the size of constants that appear in rules, then this test can be performed in polynomial time; if not, then the decision may take exponential time.

Proof: When computing all compositions of a set of argument mappings with dashed arcs and with lower bounds, we need not distinguish lower bounds or **arc** labels that exceed the maximum constant mentioned in the rules. Thus, if the maximum number of arguments and the maximum constant are both fixed, independent of the instance of the problem, then the number of different argument mappings is still polynomial in the space it takes to write down the rules.

If there are no a ***priori*** limits on these quantities, then the number of arguments still cannot exceed the problem size n, and the constants appearing in rules cannot be larger than c^n. The number of arc labels and limits in a given mapping is no more than $O(n^2)$, so the number of different argument mappings is no more than $(c^n)^{dn^2})$ for some d. This quantity, $(c^d)^{n^3}$, is still limited by 2 to a polynomial. ▪

BIBLIOGRAPHY

[CH]Chandra, A. K. and D. Harel, "Horn clauses and the fixpoint hierarchy," *Proc. ACM Symp. on Principles of Database S&stems,* pp. 158-163, 1982.

[GM]Gallaire, H. and J. Minker, Logic and Databases, Plenum, New York, 1978.

[HN] Henschen, L. J. and S. A. Naqvi, "On compiling queries in recursive first-order databases," *JACM* 31:1, pp. 47-85, 1984.

[HU] Hopcroft, J. E. and J. D. Ullman, *Introduction to Automata, Languages, and Computation,* Addison-Wesley, 1979.

[MS] McKay, D. and S. Shapiro, "Using active connection graphs for reasoning with recursive rules," *Proc. 7th IJCAI,* pp. 368-374, 1981.

[N] Naish, L., "Automatic generation of control for logic programs," TR 83/6, Dept. of CS, Univ. of Melbourne, 1983.

[R] Reiter, R., "Deductive question answering in relational databases," in [GM], pp. 149-177.

[T*] Taylor, S. et al., "Logic programming using parallel associative operations," *Intl. Symp. on Logic Programming,* pp. 58-68, 1984.

[U] Ullman, J. D., "Implementation of logical query languages for databases," unpublished manuscript, Dept. of Computer Science, Hebrew Univ., Jerusalem, Israel, May, 1984. Available from Dept. of CS, Stanford Univ.