DISTRIBUTED PROCESSING OF LOGIC PROGRAMS

by

Ouri Wolfson and Avi Silberschatz

Technical Report #466
October 1987

Distributed Processing of Logic Programs

Ouri Wolfson
Computer Science Department
The Technion - Israel Institute of Technology
Haifa 32000, Israel

Avi Silberschatz†
Computer Science Department
University of Texas at Austin
Austin, TX 78712

ABSTRACT

This paper is concerned with the issue of parallel evaluation of logic programs. To address this issue we define a new concept of predicate decomposability. If a predicate is decomposable, it means that the load of evaluating it can be divided among a number of processors, without a need for communication among them. This in turn results in a very significant speed-up of the evaluation process.

We completely characterize three classes of single rule programs (sirups) with respect to decomposability: nonrecursive, linear, and simple chain programs. All three classes were studied previously in various contexts. We establish that nonrecursive programs are decomposable, whereas for the other two classes we determine which ones are, and which ones are not decomposable. We also establish two sufficient conditions for sirup decomposability.

† This research was partially supported by ONR Contract N00014-86-K-0161 and NSR Research Grant DCR-8507224
1. Introduction

We propose a new method of evaluating logic programs in parallel. The method is suitable for sharing the computation load among an arbitrary number of processors, which either have common memory or communicate by message passing. This makes it applicable to a large class of hardware architectures. Let us demonstrate the method using the classical example of the program computing the transitive closure of a graph. The arcs of the graph are given by the tuples of a database relation $A$. The program is written in DATALOG (see [MW]):

$$T(x,y);= T(x,z)A(z,y)$$

$$T(x,y);= A(x,y).$$

If the relation $A$ is replicated at two different processors, $p_1$ and $p_2$, we can partition the work of computing (the relation for) the predicate $T$ as follows. Processor $p_1$ executes the program:

$$T(x,y);= T(x,z)A(z,y)$$

$$T(x,y);= A(x,y), even(x).$$

and processor $p_2$ executes the program:

$$T(x,y);= T(x,z)A(z,y)$$

$$T(x,y);= A(x,y), odd(x).$$

In other words, $p_1$ computes the tuples $(x,y)$ of the transitive closure, in which $x$ is even, and $p_2$ computes those tuples in which $x$ is odd.

A moment of reflection will reveal that the above partitioning of the work has several nice properties. First, no processor computes a tuple which is also computed by the other processor. Second, if the relation computed by each processor is output to the same device, or stored in the same file, the result is always the complete transitive closure, regardless of the input graph. Third, no communication between the two processors is required during the computation. Fourth, the work-partitioning does not require complicated program transformations, only adding evaluable predicates to the body of some rules of the original program.

Assume that the whole relation for $T$ has to be evaluated, and $p_1$ and $p_2$ start at the same time and execute their programs in parallel. Assume further that at the same time a single processor, using the original program, starts the evaluation of $T$. It is quite intuitive that, for an "average" (large enough) graph, the partitioned evaluation of $T$ will complete much sooner than the single-processor evaluation. Furthermore, note that the evaluation can be divided among $k$ processors, for any $k \geq 2$. The only difference from the above example is that processor $pi$ executes a copy of the program with the predicate $i \mod k(x)$ added to the nonrecursive rule. The exact time-speedup achieved by the work-partitioning scheme depends on many parameters outside the scope of this paper, however, here we are interested in a qualitative issue.

We postulate that in general, a work-partitioning scheme with the properties enumerated above, is very desirable. If it can be applied to the evaluation of a predicate in a program, then we say that the predicate is decomposable. Not every predicate is decomposable. Even for the same problem of computing the transitive closure, we will prove that the predicate $T'$ in the program:

\[ T(x,y);= T(x,z)A(z,y) \]

\[ T(x,y);= A(x,y). \]
is not decomposable. The proof of this fact will be given in section 6. Therefore, we feel that it is both practically and theoretically important to first formally define decomposability, and then characterize the decomposable predicates.

In this paper we completely characterize three subclasses of single rule programs (sirups) with respect to decomposability: nonrecursive, linear programs, and simple chain programs. Sirups were first studied as a syntactically restricted class of programs by Cosmodakis and Kanelakis ([CK]). They have only one output predicate, therefore we interchangeably use the term decomposability of a predicate or of a program. We also provide two sufficient conditions for any sirup to be decomposable. Linear programs and simple chain programs are important subclasses of sirups from the practical point of view. Simple chain programs were completely characterized with respect to membership in the complexity class NC by Afrati and Papadimitriou ([AP]). Linear sirups were studied as a distinct subclass in the context of bounded recursion ([I], [N1], [NS]) and one sided recursion ([N2]).

This work is related to the general subject of parallel evaluation of logic programs. The subject has recently emerged as a very important and active area of research ([K], [U]). However, as far as we know, existing research is concerned with membership in the complexity class NC. This class is a mathematical tool for analyzing parallel algorithms in general. Here we show that for analyzing parallel evaluation of logic programs, a different tool can be used. Loosely speaking, if a logic program is in NC it does not guarantee that it has all the nice properties of a decomposable predicate. In particular, the processors executing an NC type algorithm usually have to communicate extensively, and therefore communication is assumed to take place through common memory. Also, a speedup for such an algorithm is not guaranteed unless the number of processors is polynomial in the size of the input. The technique of program modification that we discuss here is also related to the magic sets technique ([BMSU]). Magic sets and decomposability, both aim at increasing the efficiency of query evaluation. However, the means of magic sets is selection propagation, whereas the means of decomposability is parallel evaluation.

The remainder of the paper is organized as follows. In section 2 we introduce the necessary definitions and notations used throughout the paper. In section 3 we prove that any nonrecursive sirup is decomposable. In section 4 we provide two sufficient conditions for a general sirup to be decomposable, and in section 5 we show that one of these conditions, called pivoting, is also necessary for decomposability of a linear sirup. In section 6 it is proven that a simple chain program is decomposable if and only if it is regular. In section 7 we discuss future work.

2. Preliminaries

An atom is a predicate symbol with a constant or a variable in each argument position. We assume that the constants are the natural numbers. An R-atom is an atom having R as the predicate symbol. A rule consists of an atom, \( Q \), designated as the head, and a conjunction of one or more atoms, denoted \( Q_1, ..., Q_k \), designated as the body. Such a rule is denoted \( Q : \) \( Q_1, ..., Q_k \), which should be read "\( Q \) if \( Q_1 \) and \( Q_2 \), and, ..., and \( Q_k \)." A rule or an atom is an entity. If an
entity has a constant in each argument position, then it is a ground entity. For a predicate symbol, \( R \), a finite set of \( R \)-ground-atoms is a relation for \( R \).

A DATALOG program \( P \), or a program for short, is a finite set of rules whose predicate symbols are divided into two disjoint subsets: the base predicates, and the derived predicates. The base predicates are distinguished by the fact that they do not appear in any head of a rule. An input to \( P \) is a relation for each base predicate. An output of \( P \) is a relation for each derived predicate of \( P \). A substitution applied to an entity, or a sequence of entities, is the replacement of each variable in the entity by a variable or a constant. It is denoted \( x_1/y_1, x_2/y_2, ..., x_n/y_n \) indicating that \( x_i \) is replaced by \( y_i \). A substitution is ground if the replacement of each variable is by a constant. A ground substitution applied to a rule is an instantiation of the rule.

A database for \( P \) is a relation for each predicate of \( P \). The output of \( P \), given an input \( I \), is the set of relations for the derived predicates in the database, obtained by the following procedure, called bottom up evaluation:

(1) Start with an initial database consisting of the relations of \( I \).

(2) If there is an instantiation of a rule of \( P \) such that all the ground atoms in the body are in the database generated so far, and the one in the head is not, then: add to the database the ground atom in the head of the instantiated rule, and reexecute (2).

(3) Stop.

This procedure is guaranteed to terminate, and produce a finite output for any given \( P \) and \( I \) ([VEK]). The output is unique, in the sense that any order in which bottom up evaluation adds the atoms to the database will produce the same output. For simplicity we assume that the rules of a program are range restricted, i.e. every variable in the head also appears in the body. Furthermore, we assume that the rules do not have constants, and each query is to evaluate a whole relation for a predicate.

An evaluable predicate is an arithmetic predicate (see [BR]). Examples of evaluable predicates are sum, greater than, modulo, etc. A rule \( re \) is a restricted version of some rule \( r \), if \( r \) and \( re \) have exactly the same variables, and \( r \) can be obtained by omitting zero or more evaluable predicates from the body of \( re \). In other words, \( re \) is \( r \) with some evaluable predicates added to the body, and the arguments of these evaluable predicates are variables of \( r \). For example, if \( r \) is:

\[
S(x,y,z) :- S(w,x,y), A(w,z)
\]

then one possible \( re \) rule is:

\[
S(x,y,z) :- S(w,x,y), A(w,z), x-y=5.
\]

A program \( P_i \) is a restricted version of program \( P \) if each one of its rules is a restricted version of some rule of \( P \). Note that \( P_i \) may have more than one restricted version of a rule \( r \) of \( P \). To continue the above example, if \( P \) has the rule \( r \), then \( P_i \) may have the rule \( re \) as well as the rule \( re' \):

\[
S(x,y,z) :- S(w,x,y), A(w,z), x-y=6.
\]

Throughout this paper, only restricted versions of a program may have evaluable predicates.
The input of a program with evaluable predicates, i.e. a restricted version, is defined as before. The output is also defined as before, except that step (2) also verifies that the substitution satisfies the evaluable predicates in the ground rule; only then the atom in the head is added to the database and step (2) is reexecuted. For example, for the rule \( re' \) above, the substitution \( x/14, y/8 \) satisfies the evaluable predicate \( x-y=6 \), whereas the substitution \( x/13, y/9 \) does not.

A predicate \( Q \) in a program \( P \) derives a predicate \( R \), if \( Q \) occurs in the body of a rule whose head is a \( R \)-atom. \( Q \) is recursive if \( (Q; Q) \) is in the nonreflexive transitive closure of the "derives" relation. A program is recursive if it has a recursive predicate. A rule is recursive if the predicate in its head transitively derives some predicate in its body.

A single rule program (see [CK]) is a DATALOG program having a single derived predicate, denoted \( S \) in our paper, and consisting of:

1. A nonrecursive rule,

\[
S(x_1, \ldots, x_n) :- B(x_1, \ldots, x_n)
\]

where the \( x_i \)'s are distinct variables.

2. One other, possibly recursive, rule in which the predicate symbol \( B \) does not appear.

### 3. Decomposability

Let \( P \) be a program with \( P_1, \ldots, P_r \) restricted copies of \( P \), and let \( T \) be a derived predicate of \( P \). We denote by \( T_i \) the relation output by \( P_i \) for \( T \). (Observe that this is a somewhat unconventional notation, since for \( P_i \) the relation name is different than the predicate name).

We say that predicate \( T \) is decomposable in \( P \) with respect to \( P_1, \ldots, P_r \) if the following two conditions hold:

1. For each input \( I \) to \( P, P_1, \ldots, P_r \)
   
i. \( \bigcup_i T_i \supseteq T \) (completeness).
   
   ii. \( T_i \) and \( T_j \) are disjoint for each \( i \neq j \); furthermore, if some derived predicate \( Q \) transitively derives \( T \) in \( P \), then \( Q_i \) and \( Q_j \) are disjoint (lack-of-duplication).

2. For some input \( I \) to \( P_1, \ldots, P_r \), each \( T_i \) is nonempty (nontriviality).

The above definition is central to this paper, and we shall discuss it next.

Requirement 1.i states that no output is lost by evaluating the relation for \( T \) in each \( P_i \) rather than the relation for \( T \) in \( P \); the fact that no additional output is generated is implied by the fact that each \( P_i \) is a restricted version of \( P \). Requirement 1.ii states that in the process of evaluating \( T \), each new ground atom (or intermediate result) is computed by a unique processor. Assume that, along the lines suggested in [BR section 4], we measure the cost of evaluating the relation \( T \), in terms of the number of new ground atoms generated in the evaluation process. Then, loosely speaking, requirement 1 says the following. For every input (i.e. set of base relations replicated at each processor), the evaluation by \( r \) processors is equivalent, in terms of the output produced and the total evaluation cost, to the single-processor evaluation.

The strength of requirement 1 enables the relaxed form of requirement 2. It is enough that for "some" inputs each \( T_i \) is nonempty, since for those inputs the evaluation cost incurred by
each processor is smaller than that of a single processor executing the program \( P \). Then the evaluation of \( T \) completes sooner in the distributed case. In other words, since there is nothing to lose by distributing the computation, it is enough that we gain only in some cases to make the scheme worthwhile. However, for the decomposable predicates that we discuss in this paper, nontriviality holds for more than an isolated case input.

For instance, in the transitive closure example nontriviality holds for any input graph in which arcs exit both, even and odd nodes. Specifically, for the class of predicates that we prove decomposable in this paper, decomposability is shown using the \( odd-even \) predicates alone. This has two implications. First, the work performed by each processor for an arbitrary input, is roughly equal (e.g. for an arbitrary graph, the number of odd and even nodes is roughly equal). In these cases we expect the distributed evaluation to be faster than the single-processor evaluation, by a factor which is close to two, i.e. the number processors. Second, note that the \( odd \) and \( even \) predicates are a special case of the \( i \mod r \) predicates, for \( r=2 \). When we show that \( T \) is decomposable in \( P \) with respect to \( P_1 \) and \( P_2 \), then it should be easy for the readers to convince themselves that for any \( r \), there are restricted copies \( P_1, \ldots, P_r \) such that \( T \) is decomposable in \( P \) with respect to \( P_1, \ldots, P_r \). This means that the work can be divided among any number of processors. For instance, in the transitive closure example, in order to do so processor \( i \) evaluates \( T_i \) where

\[
P_i: \quad T(x,y) :\neg T(x,z), A(z,y).
\]

\[
T(x,y) :\neg A(x,y), x = i \mod r.
\]

These facts stress the robustness of the decomposability definition.

We say that predicate \( T \) is decomposable in \( P \) if it is decomposable with respect to some restricted copies \( P_1, \ldots, P_r \) for \( r > 1 \).

**Theorem 1:** If a sirup \( P \) is nonrecursive, then its derived predicate is decomposable.

**Proof:** Assume that \( P \) is:

\[
S(x_1, \ldots, x_n) : \neg Q_1(\ldots), \ldots, Q_k(\ldots)
\]

\[
S(x_1, \ldots, x_n) : B(x_1, \ldots, x_n).
\]

where \( B \) and each \( Q_i \) are base predicates. Consider the following restricted copies of \( P \):

\[
P_1: \quad S(x_1, \ldots, x_n) : \neg Q_1(\ldots), \ldots, Q_k(\ldots), \text{even}(x_1)
\]

\[
S(x_1, \ldots, x_n) : B(x_1, \ldots, x_n), \text{even}(x_1)
\]

\[
P_2: \quad S(x_1, \ldots, x_n) : \neg Q_1(\ldots), \ldots, Q_k(\ldots), \text{odd}(x_1)
\]

\[
S(x_1, \ldots, x_n) : B(x_1, \ldots, x_n), \text{odd}(x_1).
\]

It is easy to see that \( S \) is decomposable in \( P \) with respect to \( P_1 \) and \( P_2 \), by noticing the following facts. For an arbitrary input \( I \), if the ground atom \( S(c_1,\ldots,c_n) \) is in the output of \( P \), then \( c_1 \) is either even or odd. If it is even, then given the input \( I \), \( S(c_1,\ldots,c_n) \) is in the output of \( P_1 \); however, given \( I \), \( S(c_1,\ldots,c_n) \) is not in the output of \( P_2 \). Similar arguments can be made if \( c_1 \) is odd. Therefore completeness and lack-of-duplication hold. To realize that nontriviality holds, notice that for any input \( I \) of \( P \) for which in the output there are two ground atoms \( S(c_1,\ldots,c_n) \) and
4. Sufficient Conditions for Decomposability

In this section we provide two sufficient conditions for decomposability of a general sirup. The first one is motivated by the next example, which also merits attention for the following reason. From the preceding discussion one might suspect that our notion of decomposability is equivalent to "naive" propagation of variable bindings (see introduction of [BKBR]). The latter notion means simply substituting a constant for a variable in some rules. The constant is usually taken from a query. For example, in order to find all the arcs exiting the node 2 in the transitive closure of a graph, the constant can be naively propagated into the program as follows:

\[
T(2,y) :- T(2,x), A(x,y) \\
T(2,y) :- A(2,y).
\]

It is quite clear that if a sirup is amenable to naive propagation of variable bindings, then it is decomposable. However, the reverse is not true. For example, consider the program:

\[
S(x,y) :- S(y,x) \\
S(x,y) :- A(x,y).
\]

which outputs an arc in both directions for every arc of an input graph. It is easy to see that a binding cannot be naively propagated into this program, but the sirup is decomposable; one restricted copy has the nonrecursive rule:

\[
S(x,y) :- A(x,y), \text{even}(x+y).
\]

and the other:

\[
S(x,y) :- A(x,y), \text{odd}(x+y).
\]

Our first sufficient condition for decomposability, introduced below, is based on the preceding observation.

Let \( R \) be a set of atoms, each of which has a variable in each argument position. The set \( R \) is **pivoting** if there is a subset \( d \) of argument positions, such that in the positions of \( d \):

1. The same variables appear (possibly in a different order) in all atoms of \( R \), and
2. Each variable appears the same number of times in all atoms of \( R \).

A member of \( d \) is called a **pivot**. Note that a variable that appears in a pivot may or may not appear in a nonpivot position.

The recursive rule of a sirup is **pivoting** if all the occurrences of the recursive predicate in the rule constitute a pivoting set. For example, the rule

\[
S(w,x,x,y,z) :- S(u,y,x,x,w), S(v,x,y,x,w), A(u,v,z)
\]

is pivoting, with argument positions 2, 3 and 4 of \( S \) being the pivots.

**Theorem 2:** If the recursive rule of a sirup is pivoting, then the sirup is decomposable.

**Proof:** Assume that argument positions \( i_1, \ldots, i_k \) of \( S \) are the pivots. Consider restricted copy \( P_1 \) of \( P \) which has the same recursive rule as \( P \), and a nonrecursive rule
S(x₁, ..., xₙ) :- B(x₁, ..., xₙ), even(xi₁ + xi₂ + ..., + xiₖ).

Restricted copy $P_2$ of $P$ is the same, except that the nonrecursive rule is

$S(x₁, ..., xₙ) :- B(x₁, ..., xₙ), odd(xi₁ + xi₂ + ..., + xiₖ)$.

Assume that for input $I$, the ground atom $a = S(c₁, ..., cₙ)$ is in the relation $S$ output by $P$. Assume further that $c = c_{i₁} + ..., + c_{iₖ}$ is even. Denote by $t$ the necessary and sufficient number of iterations of step (2) of the bottom up evaluation for adding $a$ to the database, in evaluating $P$. We will show by induction on $t$, that $t$ iterations are necessary and sufficient to add $a$ to $S_1$.

**Basis:** For $t=1$ the instantiation must be of the nonrecursive rule. Then obviously the claim is true, since $c$ is even.

**Inductive Step:** Assume that the claim is true for each 1 ≤ $t$ ≤ $t₀$ and that $t₀+1$ iterations are necessary and sufficient for adding $a$ to the database in evaluating $P$. Consider a sequence of $t₀+1$ iterations after which $a$ is added to the database. The last iteration must be an instantiation of the recursive rule of $P$, otherwise $t₀+1$ iterations are not necessary to derive $a$. Consider the $S$-ground-atoms in the body of this instantiation. In each one of them, the sum of the constants in argument positions $i₁$, ..., $iₖ$ is equal to $c$. For each one of these $S$-ground-atoms, the necessary and sufficient number of iterations for adding it to the database is at most $t₀$. Therefore, the claim being proven holds for each one of them, by inductive assumption. This in turn implies that $t₀+1$ iterations are necessary and sufficient to add $a$ to the database in evaluating $P₁$.

This completes the inductive proof, and we conclude that $a ∈ S₁$. Now we prove that $a$ is not in $S₂$. Assume that it is, and consider a minimal-length sequence of iterations of step (2) of the bottom up evaluation which results in $a$ being added to the database, in the evaluation of $P₂$. For each $S$-ground-atom in the sequence, the sum of constants in the pivots is $c$, thus even. At least one of the iterations instantiates the nonrecursive rule of $P₂$. However, for this instantiation, the evaluable predicate is not satisfied, since $c$ is not odd. Contradiction.

Similarly we can show that if $c$ is odd then $a ∈ S₂$ and $a$ is not in $S₁$. Therefore completeness and lack-of-duplication hold. Finally, note that for an input consisting of two atoms, $B(c₁, ..., cₖ)$ and $B(d₁, ..., dₖ)$, for which the sum of constants in the pivots is even and odd respectively, nontriviality holds. □

Theorem 2 can be extended to general programs, not necessarily sirups, provided that we extend the pivoting definition properly. Since in this paper we concentrate on sirups, we shall be informal about general programs. A rule in a program is pivoting, if all its derived-predicate-atoms (in the head or the body) constitute a pivoting set. A program is pivoting if each one of its rules is pivoting, with the same argument positions being the pivots in all the rules; additionally it is required that the heads of rule do not have repeated variables. For example, the program

$S(x₁, y, z) :- R(y, x, w), A(w, z)$

$R(x₁, y, z) :- R(x, y, w), B(w, z)$

$R(x, y, z) :- C(x, y, z)$

is pivoting, with positions 1 and 2 being the pivots. It can be shown that a predicate in a general
program is decomposable if the rules which derive the predicate constitute a pivoting program. For example, predicate $S$ in the program above is decomposable (add \textit{odd–even} $(x+y)$ to the body of the third rule).

The condition of theorem 2 is not necessary for decomposability. For example, the sirup
\begin{align*}
S(x,x) &: S(y,y), A(x,y) \\
S(x,y) &: B(x,y)
\end{align*}
is obviously not pivoting, but it is decomposable. Again, \textit{odd–even} $(x+y)$ is added to the body of the nonrecursive rule. The intuition indicates that in this example the computation load for an arbitrary input is not evenly divided between the processors executing the two restricted versions of the program (because only the processor executing the copy with the \textit{even} evaluable predicate can output an atom as a result of instantiation of the recursive rule). The example is unique (throughout the paper) in this respect. Expectedly, the last example motivates our next sufficient condition for decomposability of a sirup. It is defined as follows. Assume that $R$ is a set of atoms with each atom having the same predicate symbol, $Q$, and a variable in each argument position. The set $R$ is \textit{repeating} if there are at least two argument positions of $Q$, $i$ and $j$, such that the same variable appears in position $i$ and position $j$, and this is true for each member of $R$ (note that the variable of one member of $R$ may be different than the variable of another). The recursive rule of a sirup is \textit{repeating} if all the occurrences of the recursive predicate in the rule constitute a repeating set. For example, the rule
\begin{align*}
S(x,z,x) &: S(z,z,z), S(x,x,x)
\end{align*}
is repeating because of argument positions 1 and 3.

\textbf{Theorem 3:} If the recursive rule of a sirup is repeating, then the sirup is decomposable.

\textbf{Proof:} Very similar to the proof of theorem 2 thus omitted. The only difference between the proofs is that \textit{odd–even} $(x_i + x_j)$ replaces \textit{odd–even} $(x_1 + \cdots + x_k)$, where $i$ and $j$ are the positions of the repeated variable. $\Box$

Obviously, the condition of theorem 3 is not necessary for decomposability either.

5. Linear Sirups

In this section we completely characterize the class of linear sirups with respect to decomposability. A sirup is \textit{linear} if it is recursive, and in the body of the recursive rule there is exactly one occurrence of the recursive predicate. We also require that a linear sirup does not have repeated variables in an occurrence of the recursive predicate.

The characterization of linear sirups with respect to decomposability is done by proving that the sufficient condition of theorem 2 is also necessary. We assume that the recursive rule is:
\begin{align*}
S(x_1, \ldots, x_n) &= S(Y_1, \ldots, Y_n), A_1(\ldots), \ldots, A_k(\ldots)
\end{align*}
where the $A_i$'s are base predicates. Observe the notation used in this section to distinguish between two types of variables. The ones starting with a lowercase letter are logic program variables, or variables for short, as before. The ones starting with an upper case letter (e.g. $Y_1$), are \textit{metalinguistic-variables}. They denote program variables. For example, $Y_1$ may denote the
variable $x_n$.

If the predicate $S(x_1, ..., x_n)$ in a not necessarily linear sirup $P$ is decomposable with respect to $P_1, ..., P_r$, then we define the home-site of a sequence of $n$ constants, $\bar{c} = c_1, ..., c_n$. It is the $S_i$ to which the output atom $S(\bar{c})$ belongs, if each $P_i$ is given the input consisting of a unique atom, $B(\bar{c})$. Note that the home-site of a sequence is unique (lack-of-duplication), every sequence of $n$ constants has a home-site (completeness), and each $S_i$, $1 \leq i \leq r$, has a sequence of constants for which $S_i$ is the home-site. Let $\bar{c} = c_1, ..., c_n$ and $\bar{d} = d_1, ..., d_n$ be two sequences of constants. The ordered pair of ground atoms $<S(\bar{d}), S(\bar{c})>$ is a one-step-derivation if there is an instantiation of the recursive rule of $P$, in which the first atom is in the head and the second is in the body.

Lemma 1: If the derived predicate, $S$, of a linear sirup $P$ is decomposable, and there are two sequences of constants $\bar{d} = d_1, ..., d_n$ and $\bar{c} = c_1, ..., c_n$ such that $<S(\bar{d}), S(\bar{c})>$ is a one-step-derivation, then the home-site of $\bar{d}$ and $\bar{c}$ is identical.

Proof: Let the instantiation of the recursive rule which results in the one-step-derivation be:

$$S(\bar{d}) = S(\bar{c}), A_1(\bar{a}_1), ..., A_k(\bar{a}_k)$$

Consider the input:

$$I_1 = \{A_1(\bar{a}_1), ..., A_k(\bar{a}_k), B(\bar{c})\}.$$ 

Assume that $P$ is decomposable with respect to $P_1, ..., P_r$. Note that $S(\bar{c})$ must be in the output of some restricted version of $P$. Assume that $S(\bar{c})$ is in $S_j$. By completeness, $S(\bar{d})$ is also in the output of some restricted version. This output must be of $P_j$, for the following reason. In the input $I_1$ there is only one $B$-ground-atom, therefore the output of any restricted version other than $P_j$ is empty. Now consider the input $I_2 = I_1 \cup \{B(\bar{d})\}$. By lack of duplication, for the input $I_2$ the ground atom $S(\bar{d})$ is still in $S_j$. Therefore, for the input consisting of the single atom $B(\bar{d})$, the output ground atom $S(\bar{d})$ must be in $S_j$. $\square$

Next we discuss a procedure, called Derive-New-Variables ($P$) and given in Figure 1. We shall prove that the recursive predicate $S$ of a linear sirup $P$ is not decomposable, if the procedure Derive-New-Variables ($P$) halts (see step 3). Then we shall prove that it halts if the recursive rule of $P$ is not pivoting.

The procedure Derive-New-Variables ($P$) iteratively substitutes for the variables in the recursive rule of $P$. It starts by subscripting all variables by 1, and then at each iteration it increases the subscript of the variables and unifies the $S$-atom in the body with the atom in the head of the previous iteration. In Figure 2 we demonstrate the procedure Derive-New-Variables for the recursive rule: $S(x, y, z) :- S(w, x, y), A(w, z)$.
Derive-New-Variables ($P$).

1. Last-Rec-Rule := The recursive rule of $P$ with all the variables given the subscript of one.
2. Last-Deriv := $S(x_{10}, ..., x_{n0})$
3. Do until none of the variables of the atom in Last-Deriv is equal to one of the variables $x_{10}, ..., x_{n0}$.
4. Assume that Last-Rec-Rule = $S(x_{1}, ..., x_{n})$ := $S(Y_1, ..., Y_n), A_1(...), ..., A_k(...)$ and Last-Deriv = $S(Z_1, ..., Z_n)$.
   Let Last-Deriv := The atom in the head of the rule obtained by applying the substitution $Y_1/Z_1, ..., Y_n/Z_n$ to Last-Rec-Rule.
5. Let Last-Rec-Rule := Last-Rec-Rule with the subscript of the variables increased by one.
6. END;

Figure 1: Procedure Derive-New-Variables

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Last-Rec-Rule</th>
<th>Last-Deriv</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$S(x_1,y_1,z_1) :- S(w_1,x_1,y_1), A(w_1,z_1)$</td>
<td>$S(x_{10},y_{10},z_{10})$</td>
</tr>
<tr>
<td>2.</td>
<td>$S(x_2,y_2,z_2) :- S(w_2,x_2,y_2), A(w_2,z_2)$</td>
<td>$S(y_{10},z_{10},z_{10})$</td>
</tr>
<tr>
<td>3.</td>
<td>$S(x_3,y_3,z_3) :- S(w_3,x_3,y_3), A(w_3,z_3)$</td>
<td>$S(z_{10},z_{10},z_{10})$</td>
</tr>
<tr>
<td>4.</td>
<td>$S(x_4,y_4,z_4) :- S(w_4,x_4,y_4), A(w_4,z_4)$</td>
<td>$S(z_{10},z_{10},z_{10})$</td>
</tr>
</tbody>
</table>

Figure 2: Example of an execution of the procedure Derive-New-Variables
Lemma 2: Let $P$ be a linear sirup, and assume that $S(Y_1, ..., Y_n)$ and $S(Z_1, ..., Z_n)$ are two consecutive values of Last-Deriv in the execution of the procedure Derive-New-Variables($P$). Furthermore, assume that there is a ground substitution $p$ of the program variables in the sequence $S(Y_1, ..., Y_n), S(Z_1, ..., Z_n)$, resulting in the sequence of ground atoms $S(c_1, ..., c_n), S(d_1, ..., d_n)$. Then the pair $<S(d_1, ..., d_n), S(c_1, ..., c_n)>$ is a one-step-derivation.

Proof: By step 4 of the procedure, there is a substitution of the variables in the recursive rule, such that the rule obtained, $r_0$, has $S(Z_1, ..., Z_n)$ in the head, and $S(Y_1, ..., Y_n)$ in the body. Then the substitution $p$ can be extended to a ground substitution, $p'$, such that: when $p'$ is applied to $r_0$, it results in an instantiation of $r_0$ for which $S(c_1, ..., c_n)$ is in the body and $S(d_1, ..., d_n)$ is in the head. \[\square\]

Lemma 3: If for a linear sirup $P$, the procedure Derive-New-Variables($P$) halts, then $P$ is not decomposable.

Proof: Assume that $P$ is decomposable with respect to $P_1, ..., P_r$, and let $c_1, ..., c_n$ and $d_1, ..., d_n$ be two arbitrary sequences of constants. We will show that both have the same home-site, contradicting nontriviality. Consider the sequence of atoms $s: S(x_{10}, ..., x_{n0}), ..., S(Z_1, ..., Z_n)$, representing the consecutive values of the variable Last-Deriv, from the beginning to the end of the Derive-New-Variables execution. Observe that all the argument positions in the first and last atoms of the sequence have distinct program variables. In other words, the $x_i$'s were initially chosen to be different, the $Z_i$'s must denote distinct variables by definition of a linear sirup, and any $Z_j$ denotes a variable which is not in the set $\{x_{10}, ..., x_{n0}\}$ because Derive-New-Variables halts. Therefore, the substitution $\rho = x_{10}/c_1, ..., x_{n0}/c_n, z_1/d_1, ..., z_n/d_n$ is valid for any values of $c_1, ..., c_n, d_1, ..., d_n$. Let $\rho'$ be an extension of $\rho$ to a ground substitution of the sequence $s$. The sequence of ground atoms $s \rho'$ has the property that any two consecutive atoms in it constitute a one step derivation (by Lemma 2). Therefore, by Lemma 1, the constant-sequences $c_1, ..., c_n$ and $d_1, ..., d_n$ have the same home-site. \[\square\]

Lemma 4: If the recursive rule of a linear sirup $P$ is not pivoting, then the procedure Derive-New-Variables($P$) halts.

Proof: Construct a graph $G$ in which the nodes are the argument positions of $S$, and there is an additional node called "new". There is an edge from $i$ to $j$ if the same variable appears in position $i$ in the occurrence of $S$ in the body of the recursive rule, and in position $j$ in the occurrence of $S$ in the head. If in position $j$ in the occurrence of $S$ in the head there is a variable which does not appear in the occurrence of $S$ in the body, then draw an arc from "new" to $j$. Every node except "new" has exactly one entering arc and one exiting arc, because there are no repeated variables in an occurrence of the recursive predicate. It is easy to see that if $G$ has a cycle, then $P$ is pivoting, with the nodes of the cycle being the pivots. Since $P$ is not pivoting, $G$ is acyclic, and we conclude that there must be a path from "new" to every other node in $G$. Assume that the shortest path from "new" to some other node, $i$, is of length $m$. We show by induction on $m$, that after $m$ iterations of step 4 of Derive-New-Variables, position $i$ of the atom in Last-Deriv will have a variable which is not in the set $\{x_{10}, ..., x_{n0}\}$. For $m=1$ the claim is obvious; assume that it is true for any $m \leq q$. Let the shortest path, $p$, from "new" to $i$...
be of length $q + 1$. Suppose that $j$ is the predecessor of $i$ in $p$. By inductive assumption, after $q$ iterations of step 4, the atom in $Last-Deriv$ has in argument position $j$ a variable, denoted $Y$, which is not in the set $\{x_1, ..., x_n\}$. At iteration $q + 1$, the variable in position $j$ of the occurrence of $S$ in the body of $Last-Rec-Rule$, is substituted for $Y$. Since $i \rightarrow j$ is an arc of $G$, the variable $Y$ appears in position $i$ of the head of $Last-Rec-Rule$. This head, in turn, becomes the next value of $Last-Deriv$. □

Theorem 4: A linear sirup is decomposable if and only if its recursive rule is pivoting.
Proof: (if) Special case of Theorem 2.
(only if) Immediate from Lemmas 3 and 4. □

6. Simple Chain Programs
A simple chain program is a recursive sirup in which:
(a) All the predicates are binary.
(b) The argument positions in the left hand side of the recursive rule have distinct variables, and these variables appear in the first argument position of the first atom in the body, and in the last argument position of the last atom, respectively.
(c) All the argument positions in the body of the recursive rule have distinct variables, except that the first argument position of the second atom has the same variable as the last argument position of the first atom, the first argument position of the third atom has the same variable as the last argument position of the second atom, etc.

For example, the following is a simple chain program:

$S(x, y) :- A(x, z_1), S(z_1, z_2), S(z_2, z_3), C(z_3, z_4), D(z_4, y)$
$S(x, y) :- B(x, y)$

where the $A, B, C, D$ are base relations. A simple chain program is regular if in its recursive rule there is one occurrence of the predicate $S$ and this occurrence is the first or the last in the body of the recursive rule. Note that a simple chain program is pivoting if and only if it is regular.

Theorem 5: A simple chain program $P$ is decomposable if and only if it is regular.
Proof: (if) Immediate, based on Theorem 2.
(only if) Assume that $P$ is decomposable with respect to restricted copies $P_1, P_2, ..., P_r$ of $P$, for $r > 1$. Denote the recursive rule of $P$ by:

$S(x, y) :- Q^1(x, z_1), ..., Q^t(z_{t-1}, y)$

where some of the $Q^i$'s are $S$'s, and $t > 1$. Using the regular notation, the nonrecursive rule is:

$S(x, y) :- B(x, y)$. 

By nontriviality there are two sequences of constants, $j_1, k_1$ and $j_2, k_2$ with home sites $S_1$ and $S_2$ respectively. Since the recursive rule of $P$ is not regular, there are two cases to analyze:

Case 1: There is a subsequence in the body of the recursive rule, of the following form:

$Q^{i-1}(z_{i-2}, z_{i-1}), S(z_{i-1}, z_i), Q^{i+1}(z_i, z_{i+1})$. 

Technion - Computer Science Department - Technical Report CS0466 - 1987
Let \( I^2 \) consist of the set of ground atoms:

\[
Q^1(c_1c_2), Q^2(c_2c_3), ..., Q^{i-1}(c_{i-1}j_1)B(j_1k_1), Q^{i+1}(k_1c_{i+1}), Q^{i+2}(c_{i+1}c_{i+2}), ..., Q^t(c_{t-1}c_t)
\]

where:

1) Each predicate \( S(m,n) \) in the list is a notation for \( B(m,n) \).
2) Each pair of different \( c \)'s represents different constants.
3) None of the \( c \)'s is in the set \( \{h,k_1,h,k_2\} \).

For the input \( I^2 \) the ground atom \( S(c_1c_t) \) is in the output \( S \) of \( P \). By completeness, for this input, \( S(c_1c_t) \) is in some \( S_j \). We will show that \( S(c_1c_t) \) is in \( S_1 \).

Assume otherwise, i.e. \( S(c_1c_t) \) is in \( S_b \) for \( b \neq 1 \). The atom \( B(c_1c_t) \) is not in \( I^2 \) because \( t > 1 \), therefore \( S(c_1c_t) \) must be added to the database by instantiating a recursive rule of \( P_b \) in step BUE2. Consider the first such instantiation. Note that the sequence \( s \) of ground atoms constituting the body of the instantiated rule cannot have the same atom twice; if it does, then there is at least one ground atom in \( s \), for which both argument positions have the same constant. However, it is easy to realize from the definition of \( I^2 \), that in the database generated by bottom-up-evaluation so far, all atoms must have different constants in the two argument positions. (Note that even though \( j_1 \) may be equal to \( k_1 \), the ground atom \( S(c_1k_1) \) is not in \( S_b \) because \( j_1 \neq k_1 \).)

Therefore, in the first instantiation of a recursive rule, the body of the instantiated rule has \( t \) distinct atoms, each of which is not a \( B \)-ground-atom (because by sirup definition the predicate symbol \( B \) is not in the recursive rule). However, since the home site of \( j_1, k_1 \) is \( S_1 \), in the database generated so far there can be at most \( t-1 \) such atoms. The contradiction results from our assumption that \( S(c_1c_t) \) is in \( S_b \).

Subcase 2.1: Assume that \( j_1 \neq k_1 \). Let input \( I^3 = \{B(j_1k_1), B(k_1j_3), B(j_3k_3)\} \). This input relation can be regarded as a graph consisting of a path, therefore \( S(j_1j_3) \) and \( S(k_1k_3) \) are in \( S \). Assume that the home site of \( j_1, j_3 \) is \( S_1 \) for \( I \neq 1 \). But then it is easy to see that \( S(j_1j_3) \) is not in any \( S_j \); contradicting completeness. If the home site of \( k_1, j_3 \) is \( S_1 \), then it is easy to see that for input \( I^3 \) the atom \( S(k_1k_3) \) is not in in any \( S_j \); again contradicting completeness.

Subcase 2.2: Assume that \( j_1 = k_1 \). In other words, the home site of \( j_1/j_1 \) is \( S_1 \). Let input \( I^6 = \{B(j_1j_3), B(j_3k_3), B(k_3j_1)\} \). This input relation can be regarded as a graph consisting of a cycle, therefore \( S(j_1j_1) \) is in \( S \). Since \( S(j_3k_3) \) is in \( S_d \) only, for the input \( I^6 \), \( S(j_1j_1) \) cannot be
in any $S_i$ other than $S_d$. But then, for the input $I^d \cup \{ B(j_1,j_1) \}$ the ground atom $S(j_1,j_1)$ is in both, $S_1$ and $S_d$; this contradicts lack of duplication. □

7. Future Work.
We shall continue the work on decomposability in several directions. One of them is to extend the characterization of decomposable predicates to other sirups first, e.g. typed (see [K]), and then to general logic programs. Another direction is to determine whether decomposition implies that the work can be evenly divided among the processors, as we have seen that can be done using the mod predicate. For this purpose a notion of fair decomposition should be defined. Another topic which merits attention is minimizing communication when evaluating nondecomposable predicates in a distributed environment. We feel that the work on decomposability should also be helpful in this area. More specifically, observe that the method proposed in this paper to partition the load in evaluating decomposable predicates, can be applied to nondecomposable ones as well; however in that case communication among the processors is necessary. The question is, how does the amount of necessary communication compare in different partitioning schemes. Finally, we shall mention that we intend to study the relationship between the class of decomposable programs and the programs in the complexity class NC.

8. References


