Vol. 38 No. 1 Transactions of Information Processing Society of Japan Jan. 1997

Regular Paper

Decomposable Programs Revised

XIAOYONG DU,† ZHIBIN LIU,† and NAOHIRO ISHII†

Program decomposition is a program optimization technique for multiple linearly recursive programs in deductive databases. It decomposes an original program into a set of subprograms that have small arities and share no recursive predicates. 2D-decomposable programs [7] generalize some previously proposed decomposable programs, including one-sided recursions [2]; separable recursions [3]; right-, left-, and mixed-linear recursions [4]; and generalized separable recursions [5]. This paper revises the concept of 2D-decomposability, and proposes two larger program classes based on detailed analysis and classification of the arguments of the recursive predicate. We prove that the proposed program classes are decomposable to some extent.

1. Introduction

Program decomposition means decomposition of an original program into a collection of subprograms that have small arities and share no recursive predicates. Intuitively, the size of the recursive predicate (relation) is bounded by n^k, where n is the number of distinct constants in the database and k is the arity of the recursive predicate. Reducing the arity of the recursive predicate can thus result in an order-of-magnitude increase in the efficiency of the evaluation algorithm. Wang, et al. [7] proposed the concept of 2D-decomposability for multiple linearly recursive programs, and showed that 2D-decomposable programs systematically generalize some previously proposed decomposable programs, including separable recursions [3]; right-, left-, and mixed-linear recursions [4]; and generalized separable recursions [5]. However, as we will show, 2D-decomposable programs exclude some meaningful recursions that are decomposable to a certain extent.

Example 1.1 Consider the following multiple linearly recursive (mL for short) program which is a modified version of an example from Wang, et al. [7].

Three types of part, called Type A, Type B, and Type C, are used in projects. Relations a(X,Y), b(X,Y), and c(X,Y) denote that X is an immediate subpart of Y for Types A, B, and C, respectively. Relation $d_i(U)\,(i = 1,2,3)$ is a collection of projects satisfying some property. Relation q(X,Y,Z,U) is an initial relation in which parts X, Y, and Z, which come from Types A, B, and C, respectively, are used in the same project U. We now define a relation p(X,Y,Z,U) that computes all triples ⟨X,Y,Z⟩ of parts such that they come from different types and will be used in the same project.

$$r_0 : p(X,Y,Z,U) :- q(X,Y,Z,U)$$
$$r_1 : p(X,Y,Z,U) :- a(X,A), p(A,Y,Z,U), d_1(U)$$
$$r_2 : p(X,Y,Z,U) :- b(Y,B), p(X,B,Z,U), d_2(U)$$
$$r_3 : p(X,Y,Z,U) :- c(Z,C), p(X,Y,C,U), d_3(U).$$

This program is not 2D-decomposable by the definition in Wang, et al. [7], since relation $d_i\,(i = 1,2,3)$ makes all recursive rules connected.

However, it can still be decomposed into three small programs as follows:

$$D_1 : \begin{array}{l} p_1(X,U,-) :- q(X,Y,Z,U,-) \\ p_1(X,U,-) :- a(X,A), \\ \qquad p_1(A,U,-), d_1(U) \end{array}$$

$$D_2 : \begin{array}{l} p_2(Y,U,-) :- q(X,Y,Z,U,-) \\ p_2(Y,U,-) :- b(Y,B), \\ \qquad p_2(B,U,-), d_2(U) \end{array}$$

$$D_3 : \begin{array}{l} p_3(Z,U,-) :- q(X,Y,Z,U,-) \\ p_3(Z,U,-) :- c(Z,C), \\ \qquad p_3(C,U,-), d_3(U), \end{array}$$

where "−" is a unique ID attached to each tuple of $q(X,Y,Z,U)$. It gives a value identifying the initial p tuple from which a derived tuple comes, thus it keeps the original source of a derived tuple [7]. The finial result can be generated by computing a join of $p_1(X,U,-)$, $p_2(Y,U,-)$ and $p_3(Z,U,-)$. □

This example encourages us to revise the con-

† Department of Intelligence and Computer Science, Nagoya Institute of Technology

cept of 2D-decomposability proposed by Wang, et al.[7] so that more programs are included.

The rest of this paper is organized as follows. Section 2 revises the concept of 2D-decomposability after briefly reviewing some basic terms in deductive databases, and defines two program classes that are larger than the 2D-decomposable class. Sections 3 and 4 show that these two classes of programs are decomposable. In the final section, we present a brief comparison with related work and offer our conclusions.

2. Basic Definitions

Here we give some definitions and assumptions required for the rest of the paper. Assume that there is an underlying first-order language without function symbols. A *program* is a finite set of clauses called rules of the form

$$A : -A_1, \cdots, A_m (m \geq 0), \qquad (1)$$

where A, called the *head*, is an atom of an ordinary predicate, and $A_1, \cdots, A_m$, called the *body*, stands for the conjunction $A_1 \wedge \cdots \wedge A_m$; each A_i is called the subgoal and is an atom of either an ordinary predicate or a built-in predicate such as $=$, $\geq$, $\leq$, $\neq$, $>$, $<$. A predicate is called a *base predicate* if it is not a head in the program. Otherwise, it is called a *derived predicate*. A derived predicate is called *recursive* if it is contained in a cycle in the *dependency graph* of a program, which has all predicates as its nodes and has an edge from A to B if A is found in the body and B is found in the head of the same rule. A rule is *linearly recursive* if the head is the sole recursive predicate and appears exactly once in the body. A program is *multiple linearly recursive* if it contains only one recursive predicate defined by more than one recursive rule. Nonrecursive rules are called *exit rules*, and the corresponding predicate E is called the *exit predicate*.

In this paper, we consider only multiple linearly recursive programs with a single recursive predicate, called mL for shorthand. Furthermore, by introducing a special built-in predicate "=", we may assume without loss of generality that rules are *rectified*[6]; that is, the heads of the rules in the program are identical and contain neither constants nor repeated variables.

Definition 2.1 Let r denote a recursive rule in mL program P with recursive predicate p, and let $p[i]$ denote the i th position of predicate p, or i for short if this does not cause any confusion in the context. A position $p[t]$ is *persistent* if the same variable is found in position $p[t]$ of the two p instances and nowhere else in r. A position $p[t]$ is *semi-persistent* if the same variable is found in position $p[t]$ of the two p instances and in at least one nonrecursive predicate. A collection of positions $T = \{p[1], \cdots, p[t]\}$ is a *permutation* if the same set of variables are found in the positions T of the two p instances, and these variables are found nowhere else in r. We denote these three kinds of position as $pers(r)$, $semi(r)$, and $perm(r)$, respectively.

For example, for the second rule r_1 in Example 1.1, we have: $pers(r_1) = \{p[2], p[3]\}$, $semi(r_1) = \{p[4]\}$, and $perm(r_1) = \{\{p[2]\}, \{p[3]\}, \{p[2], p[3]\}\}$.

Example 2.2 Consider another rule r,

$$r : p(X, Y, Z, U, V, W) \; : -a(X, A), b(Z), p(A, Y, Z, V, W, U).$$

We have $pers(r) = \{p[2]\}$, $semi(r) = \{p[3]\}$, and $perm(r) = \{\{p[2]\}, \{p[4], p[5], p[6]\}, \{p[2], p[4], p[5], p[6]\}\}$. □

Definition 2.3 Let T be a set of permutation positions of p. We define a function $h : T \to T$ such that $h(i) = j$ if i and j are in T, and the same variable X is found in the i position of head instance of p and in the j position of body instance of p in the rule r. We also define powers of h as

$$h^1(X) = h(X)$$
$$h^n(X) = h(h^{n-1}(X)).$$

h is called the *permutation function* of the rule r.

From the definition of permutation, $h(X)$ is a bijective function (i.e., a one-to-one function).

Definition 2.4 Let T_1 and T_2 be two subsets of the permutation positions in rule r_1 and r_2, respectively. Let h_1 and h_2 be the permutation functions of r_1 and r_2, respectively. T_1 and T_2 are *consistent* between r_1 and r_2 if $T_1 = T_2$ and for each X in T_1 (T_2), there is

$$h_1(h_2(X)) = h_2(h_1(X)). \qquad (2)$$

T is a *consistent permutation* in P if for each pair of rules r_1 and r_2 in P, Eq. (2) holds.

Example 2.5 Consider the rule in Example 2.2, and its permutation set $T = \{4, 5, 6\}$. Its permutation function on T is $\{h_r(4) = 6, h_r(5) = 4, h_r(6) = 5\}$. If there is another rule s, then

$$s : p(X, Y, Z, U, V, W) \; : -c(Y, B), d(Z), p(X, B, Z, W, U, V).$$

Its permutation on T is $\{h_s(4) = 5, h_s(5) = 6, h_s(6) = 4\}$.

It is easy to certify that $h_r(h_s) = h_s(h_r)$ for every element in T. Hence T is a consistent permutation in the program $P = \{r, s\}$. □

Definition 2.6 Let $e \subseteq P$ be a subset of rules in P. The persistent, semi-persistent, and permutation positions of p w.r.t. e are defined respectively below.

(1) $pers(e) = \cap_{r \in e}(pers(r))$;
(2) $semi(e) = \cap_{r \in e}(semi(r) \cup pers(r)) - pers(e)$;
(3) $perm(e) = \{T | T$ is a consistent permutation in $e\} - pers(e)$.

Definition 2.7 For each recursive rule in P, we define three sets of positions of p as follows:

$vary_1(r) = full(r) - pers(r)$;
$vary_2(r) = full(r) - (pers(r) \cup semi(P))$;
$vary_3(r) = full(r) - (pers(r) \cup perm(P))$,

where $full(r)$ represents the collection of all positions of the recursive predicate p in r.

Definition 2.8 Two nonrecursive predicates Q and R are connected if they share a variable or if there is a nonrecursive predicate S such that Q and S are connected and S and R share a variable. Two variables are connected if they are in the same nonrecursive predicate, or they belong to nonrecursive predicates Q and R, respectively, and Q and R are connected.

Definition 2.9 Two positions $p[t]$ and $p[s]$ are connected if there is at least one rule r in P such that the variables in the $p[t]$ and $p[s]$ position of the head instance of p are connected. We now define the relations among rules.

Definition 2.10 Recursive rules r and s are a c-connection ($c \in \{1, 2, 3\}$) if either $vary_c(r) \cap vary_c(s) \neq \phi$ or there exists some recursive rule t such that $vary_c(r) \cap vary_c(t) \neq \phi$ and t and s are a c-connection. The c-connection partitions the set of recursive rules in P into equivalence classes $e_1, e_2, \cdots, e_{m_c}, m_c \geq 1$, so that r and s are in the same class if and only if they are c-connected. For each class $e_i = \{r_{i_1}, \cdots, r_{i_{m_i}}\}$, c-$dyn_i$ (or c-$dyn(e_i)$) denotes the positions $vary_c(r_{i_1}) \cup \cdots \cup vary_c(r_{i_{m_i}})$, called the c-dynamic positions in e_i, and c-dyn (or c-dyn(P)) denotes c-$dyn_1 \cup \cdots \cup$ c-dyn_{m_c}, called the c-dynamic positions of P.

Example 2.11 Consider program P in Example 1.1.

We have $pers(P) = \phi$, $pers(r_1) = \{p[2], p[3]\}$, $pers(r_2) = \{p[1], p[3]\}$, and $pers(r_3) = \{p[1], p[2]\}$.

(1) Consider the 1-connection. We have $vary_1(r_1) = \{p[1], p[4]\}$, $vary_1(r_2) = \{p[2], p[4]\}$, and $vary_1(r_3) = \{p[3], p[4]\}$. Hence, the partition of recursive rules is $\{e_1 = \{r_1, r_2, r_3\}\}$, and 1-$dyn_1 = \{p[1], p[2], p[3], p[4]\}$. It is a singleton partition.

(2) Consider the 2-connection. We have $semi(r_1) = semi(r_2) = semi(r_3) = semi(P) = \{p[4]\}$. Hence $vary_2(r_1) = \{p[1]\}$, $vary_2(r_2) = \{p[2]\}$, and $vary_2(r_3) = \{p[3]\}$. The partition of recursive rules is $\{e_1 = \{r_1\}, e_2 = \{r_2\}, e_3 = \{r_3\}\}$, and 2-$dyn_1 = \{p[1]\}$, 2-$dyn_2 = \{p[2]\}$, 2-$dyn_3 = \{p[3]\}$. It is a partition that has three equivalent classes. □

Definition 2.12 Let P be an mL program. we define three program classes as follows:

(1) $\Sigma_1 = \{P | nonsingleton(\text{1-dyn})\}$;
(2) $\Sigma_2 = \{P | nonsingleton(\text{2-dyn}) \ \& \ semi(P) \neq \phi\}$;
(3) $\Sigma_3 = \{P | nonsingleton(\text{3-dyn}) \ \& \ perm(P) \neq \phi\}$,

where $nonsingleton(dyn)$ is a predicate that indicates dyn has a nonsingleton partition.

Theorem 2.13 Σ_1 is equivalent to the 2D-decomposable programs defined in Wang, et al.[7], except when $pers(P) \neq \phi$ and 1-dyn is a singleton partition.

Wang, et al.'s definition covers the case in which $pers(P) \neq \phi$ and 1-dyn is a singleton partition. However, the program in this case is not horizontally decomposable, because when the arguments in $pers(P)$ are removed, the number of rules that define a new recursive predicate cannot be reduced. We thus exclude this case from our definition for Σ_1.

Theorem 2.14 Let Σ_i $(i = 1, 2, 3)$ be defined in Definition 2.12. Then

$\Sigma_1 \subset \Sigma_2$

and

$\Sigma_1 \subset \Sigma_3$.

Theorem 2.14 says that Σ_2 and Σ_3 are larger classes than Σ_1. Example 1.1 shows an example in which $P \in \Sigma_2$ but $P \notin \Sigma_1$. In the next two sections, we show that the programs in Σ_2 and Σ_3 classes can also be decomposed into some subprograms that have smaller arities and numbers of rules.

3. Decomposing Programs in the Σ_2 Class

We first prove a syntax property of recursive rules in Σ_2.

Lemma 3.1 Let P be an mL program in the Σ_2 class. Assume that all recursive rules is partitioned into equivalent classes $e_1, \cdots, e_{m_2}$ under 2-connection. Then every recursive rule in e_i has the following form, by reordering the positions of p:

$$p(\vec{X}_i, \vec{Y}, \vec{Z}_i) : -\psi(\vec{X}_i, \vec{W}_i, \vec{Y}), p(\vec{W}_i, \vec{Y}, \vec{Z}_i),$$

where

(1) $\vec{Y} \neq \phi$ is the vector of arguments in the positions $semi(P)$; that is, $\vec{Y}$ is found in the same positions of two instances of p, and possibly in nonrecursive predicates. $\vec{Y}$ is the same for all rules in P;

(2) $\vec{Z}_i$ is the vector of arguments in the positions $pers(e_i)$; that is, $\vec{Z}_i$ is found in the rule exactly twice, once in the head and once in the body in the same position. $\vec{Z}_i$ is the same for all rules in e_i;

(3) $\vec{X}_i$, $\vec{W}_i$ are the vectors of arguments in the positions 2-dyn_i, which are restricted by a conjunction ψ. $\vec{X}_i$ and $\vec{W}_i$ are the same for all rules in e_i.

Proof. Let r be a recursive rule in e_i.

(1) Since $P \in \Sigma_2$, $semi(P) \neq \phi$. Let $\vec{Y} = semi(P)$; then $\vec{Y} \neq \phi$ in r. According to the definition of $semi(P)$, $\vec{Y}$ is found in the same positions of two instances of p, and there is at least one recursive rule in P in which $\vec{Y}$ is in $\psi(\vec{X}_i, \vec{W}_i, \vec{Y})$. That is, $\vec{Y}$ possibly appears in nonrecursive predicates in r.

(2) Now we consider $full(r) - \vec{Y}$, where $full(r)$ is all the arguments in p. $full(r) - \vec{Y}$ can be divided into two parts: those arguments in $pers(r)$ and those in $vary_2(r)$. Let $\vec{Z}_i$ be the vector of arguments in the positions of $\cap_{r \in e_i}(pers(r))$. It is fixed in e_i, and found in the rule exactly twice, once in the head and once in the body, in the same positions.

(3) Let $X_i = full(r) - \vec{Y} - \vec{Z}_i$ be the remainder of the arguments of the p instance in the head, and let W_i be the corresponding arguments of p in the body.

From the definition of the 2-dynamic positions,

$$\begin{aligned} &2\text{-}dyn(e_i)\\ &= \cup_{r \in e_i} vary_2(r)\\ &= \cup_{r \in e_i}(full(r) - (semi(P) \cup pers(r)))\\ &= \cup_{r \in e_i}(full(r) - semi(P) - pers(r))\\ &= full(r) - semi(P) - \cap_{r \in e_i} pers(r)\\ &= full(r) - semi(P) - pers(e_i). \end{aligned}$$

Therefore, X_i and W_i are two vectors of arguments in the 2-dyn_i, and are found in a conjunction formula, say ψ.

From the above three aspects, we have thus proved the lemma. □

The importance of Lemma 3.1 is that only the rules in e_i can derive new values for the positions 2-dyn_i and that positions $pers(e_i)$ in these rules are irrelevant to such derivations. No rules in P can derive any new value for the position $semi(P)$, but all play a role in placing restrictions on the evaluation of the values for the positions 2-dyn_i. Therefore, to compute values for the positions 2-dyn_i and $semi(P)$, we need to consider only the rules in e_i and the positions 2-dyn_i and $semi(P)$ of p. This motivates the following definition.

Definition 3.2 Let $P \in \Sigma_2$. Assume that the set of recursive rules in P is partitioned into equivalent classes $e_1, \cdots, e_{m_2}$ under 2-connection. For each recursive rule r,

$$p(\vec{X}_i, \vec{Y}, \vec{Z}_i) : -\psi(\vec{X}_i, \vec{W}_i, \vec{Y}), p(\vec{W}_i, \vec{Y}, \vec{Z}_i), \tag{3}$$

in e_i, where $\vec{X}_i$, $\vec{Y}$ and $\vec{Z}_i$ are the variables that appear in 2-$dyn(e_i)$, $semi(P)$, and $pers(e_i)$, respectively, let $\Pi_i(r)$ denote the rule

$$q_i(\vec{X}_i, \vec{Y}, -) : -\psi(\vec{X}_i, \vec{W}_i, \vec{Y}), q_i(\vec{W}_i, \vec{Y}, -), \tag{4}$$

and for each exit rule r,

$$p(\vec{X}_i, \vec{Y}, \vec{Z}_i) : -exit(\vec{X}_i, \vec{Y}, \vec{Z}_i) \tag{5}$$

in P, where $\vec{X}_i, \vec{Y}$, and $\vec{Z}_i$ are same vectors of arguments as in formula Eq. (3), let $\Pi_i(r)$ denote the rule

$$q_i(\vec{X}_i, \vec{Y}, -) : -exit(\vec{X}_i, \vec{Y}, \vec{Z}_i, -), \tag{6}$$

where "$-$" is a unique ID for every tuple in the exit relation. The program $D_i = \{\Pi_i(r) | r \in e_i\} \cup \{\Pi_i(r) | r \text{ is an exit rule}\}$ is called the projection of P on e_i. We denote it as $D_i = \Pi_i(P)$.

Theorem 3.3 Let P be an mL program in the Σ_2 class. Assume that the set of recursive rules in P is partitioned into equivalent classes $e_1, \cdots, e_{m_2}$ under 2-connection, and that D_i is the projection of P on e_i. Then

$$P = \bowtie_{i=1}^{m_2} (D_i) \bowtie q$$

where $\bowtie$ is the natural join of relations D_i, and q is the projection of the exit relation on $pers(P) \cup semi(P)$.

Proof. Consider a rule r in e_i:

$$p(\vec{X}_i, \vec{Y}, \vec{Z}_i) : -\psi^{(r)}(\vec{X}_i, \vec{W}_i, \vec{Y}), \quad p(\vec{W}_i, \vec{Y}, \vec{Z}_i).$$

We can represent $\psi^{(r)}(\vec{X}_i, \vec{W}_i, \vec{Y})$ conceptually as two parts, $\psi_1^{(r)}(\vec{X}_i, \vec{W}_i, id_r)$ and $\psi_2^{(r)}(\vec{Y}, id_r)$, where id_r is the tuple identifier of virtual relation $\psi^{(r)}$.

We now show that for any rule $r_i \in e_i$, $r_j \in e_j$ $(i \neq j)$ r_i and r_j commute. In fact, since e_i and e_j are two different equivalent classes, $\vec{X}_i \cap \vec{X}_j = \phi$, and $\vec{W}_i \cap \vec{W}_j = \phi$. We also have $\vec{Y} \cap \vec{T} = \phi$, where $\vec{T} \in \{\vec{X}_i, \vec{X}_j, \vec{W}_i, \vec{W}_j\}$. Hence,

$$\begin{aligned}
&\psi^{(r_i)}(\vec{X}_i, \vec{W}_i, \vec{Y}), \psi^{(r_j)}(\vec{X}_j, \vec{W}_j, \vec{Y}) \\
&= \psi_1^{(r_i)}(\vec{X}_i, \vec{W}_i, id_{r_i}), \psi_2^{(r_i)}(\vec{Y}, id_{r_i}), \\
&\quad \psi_1^{(r_j)}(\vec{X}_j, \vec{W}_j, id_{r_j}), \psi_2^{(r_j)}(\vec{Y}, id_{r_j}) \\
&= \psi_1^{(r_j)}(\vec{X}_j, \vec{W}_j, id_{r_j}), \psi_2^{(r_j)}(\vec{Y}, id_{r_j}), \\
&\quad \psi_1^{(r_i)}(\vec{X}_i, \vec{W}_i, id_{r_i}), \psi_2^{(r_i)}(\vec{Y}, id_{r_i}) \\
&= \psi^{r_j}(\vec{X}_j, \vec{W}_j, \vec{Y}), \psi^{r_i}(\vec{X}_i, \vec{W}_i, \vec{Y})
\end{aligned}$$

Therefore, for any sequence of applications of rules in P, we can commute them in such a way that all rules in e_i are applied before those in e_j $(i < j)$. By Lemma 3.1, all rules in e_i can derive new values only for the positions 2-dyn_i, and the positions $pers(e_i)$ in these rules are irrelevant to such derivations. This means that we can eliminate all variable in $pers(e_i)$ if we consider only rules in e_i. □

Example 3.4 Consider the following program P:

$$\begin{aligned}
r_0 &: p(X, Y, Z, U, V, W) : - \\
&\qquad q(X, Y, Z, U, V, W) \\
r_1 &: p(X, Y, Z, U, V, W) : -a(X, A), \\
&\qquad p(A, Y, Z, U, V, W), d_1(U) \\
r_2 &: p(X, Y, Z, U, V, W) : -b(Y, B), \\
&\qquad p(X, B, Z, U, V, W), d_2(V) \\
r_3 &: p(X, Y, Z, U, V, W) : -c(Z, C), \\
&\qquad p(X, Y, C, U, V, W), d_3(W)
\end{aligned}$$

We have $pers(P) = \phi$, $semi(P) = \{p[4], p[5], p[6]\}$. The partition of recursive rules is $\{e_1 = \{r_1\}, e_2 = \{r_2\}$, and $e_3 = \{r_3\}\}$, and 2-$dyn_1 = \{p[1]\}$, 2-$dyn_2 = \{p[2]\}$, 2-$dyn_3 = \{p[3]\}$. From the Theorem 3.3, the program P can be decomposed into three subprograms, $D_i = \Pi_i(P)\,(i = 1, 2, 3)$, as follows.

$$\begin{aligned}
D_1 &: p_1(X, U, V, W, -) : - \\
&\qquad q(X, Y, Z, U, V, W, -) \\
&\quad p_1(X, U, V, W, -) : -a(X, A), \\
&\qquad p_1(A, U, V, W, -), d_1(U)
\end{aligned}$$

$$\begin{aligned}
D_2 &: p_2(Y, U, V, W, -) : - \\
&\qquad q(X, Y, Z, U, V, W, -) \\
&\quad p_2(Y, U, V, W, -) : -b(Y, B), \\
&\qquad p_2(B, U, V, W, -), d_2(V)
\end{aligned}$$

$$\begin{aligned}
D_3 &: p_3(Z, U, V, W, -) : - \\
&\qquad q(X, Y, Z, U, V, W, -) \\
&\quad p_3(Z, U, V, W, -) : -c(Z, C), \\
&\qquad p_3(C, U, V, W, -), d_3(W).
\end{aligned}$$

□

Obviously, further optimization of the above example is possible. According to the definition of $semi(P)$, it contains two parts for every recursive rule r, one belonging to $semi(r)$ and the other belonging to $pers(r)$. If $\cup_{r \in e_i}(semi(r)) \cap \cup_{r \in e_j}(semi(r)) = \phi$, the variables in the position of $semi(P) - \cup_{r \in e_i}(semi(r))$ can be eliminated from the subprogram D_i, since the rules in e_i cannot derive new values for those positions.

Example 3.5 The subprogram in Example 3.4 can be further optimized as follows:

$$\begin{aligned}
D_1 &: p_1(X, U, -) : - \\
&\qquad q(X, Y, Z, U, V, W, -) \\
&\quad p_1(X, U, -) : -a(X, A), \\
&\qquad p_1(A, U, -), d_1(U)
\end{aligned}$$

$$\begin{aligned}
D_2 &: p_2(Y, V, -) : - \\
&\qquad q(X, Y, Z, U, V, W, -) \\
&\quad p_2(Y, V, -) : -b(Y, B), \\
&\qquad p_2(B, V, -), d_2(V)
\end{aligned}$$

$$\begin{aligned}
D_3 &: p_3(Z, W, -) : - \\
&\qquad q(X, Y, Z, U, V, W, -) \\
&\quad p_3(Z, W, -) : -c(Z, C), \\
&\qquad p_3(C, W, -), d_3(W).
\end{aligned}$$

□

By the definitions in Section 2 and the theorem in Section 3, it is easy to construct a polynomial time algorithm to decompose programs in the Σ_2 class.

4. Transforming Programs in the Σ_3 Class

In this section, we first discuss some properties of the permutation, and then prove the theorem that any program in Σ_3 class can be transformed into a program in Σ_1 or Σ_2 class by introducing indexes. We also show that the computation of the indexes can be simplified, and hence that there is no shortcoming like that of the indexes used in the counting method [1].

Definition 4.1 Let $T = \{1, \cdots, m\}$ be a set of permutation positions of p, and let h be a bijective function defined on T. T is called a *primary permutation set* if no subset of T is a permutation set.

Example 4.2 Consider the recursive definition

$p(X, Y, Z, U, V) : -p(Y, Z, X, V, U).$

Obviously $T = \{1, 2, 3, 4, 5\}$ is a permutation set, whose permutation function is defined as $\{h(1) = 2, h(2) = 3, h(3) = 1, h(4) = 5, h(5) = 4\}$. However, it is not a primary permutation set, since one of its subsets, $T_1 = \{1, 2, 3\}$, is also a permutation set. It is easy to show that T_1 is a primary permutation set.

Lemma 4.3 Let $T = \{1, \cdots, m\}$ be a permutation set of p, and let h be a bijective function defined on T. If $T_1 \subset T$ is a primary permutation set, then $T_2 = T - T_1$ is also a permutation set. □

Hence, a permutation set is either a primary permutation set or can be divided into several primary permutation parts that are disjoint.

Lemma 4.4 Let $T = \{1, \cdots, m\}$ be a permutation set of p, and let h be a bijective function defined on T. Assume that h is not an identity.

(1) if T is a permutation set, then $h^i(x) \neq x$, $h^i(x) \neq h^j(x)$, for all $0 < i \neq j < m$, and $h^m(x) = x$, for any $x \in T$.

(2) If T is not a primary component, and T is assumed to be divided into d primary permutation subsets $\{T_1, \cdots, T_d\}$ with c_i elements in T_i $(i = 1, \cdots, d)$, then for any $x \in T$, $h(x) = h^c(x)$, where $c = lcm(c_1, \cdots, c_d)$ and lcm is the least common multiple of the list.

Proof.

(1) Assume that there is a constant $c < m$ such that $h^c(x) = x$. Consider a set $T_1 = \{a_0, \cdots, a_c\}$, where $a_0 = x$, $a_i = h^i(x)\,(i = 1, \cdots, c)$. Since $h(a_i) = h(h^i(x)) = h^{i+1}(x) = a_{i+1}$ for $i < c$, and $a_c = h^c(x) = x = a_0$, moreover $h(x)$ is a bijective function on T_1, hence $T_1 \subset T$ is a primary permutation set. This contradicts the condition that T is a primary permutation set. Similarly, we have $h^i(x) \neq h^j(x)$.

If $h^m(x) \neq x$, then there are m elements besides x in T that are different. This is a contradiction.

(2) The second step of the proof can be derived from Lemma 4.4 and (1). □

Lemma 4.4 says that all elements in the permutation set T form a cycle; that is, its elements can be represented as $a_0, \cdots, a_{m-1}$, where $a_i = h(a_{i-1})$ and $a_m = a_0$.

Lemma 4.5 Let $T = \{1, \cdots, m\}$ be a permutation set of p, and let h_1, and h_2 be two consistent permutation functions defined on T; then there is a constant $c \leq m$ such that for any $x \in T$, $h_2(x) = h_1^c(x)$.

Proof.

(1) If T is a primary permutation set for h_1, according to Lemma 4.4, all elements in T can be represented as a cycle under the permutation function h_1. Let $T = \{a, a^2, \cdots, a^m | a^{i+1} = h_1^i(a)\}$. Then, for each $x \in T$, $h_2(x) = h_1^{c(x)}(x)$, where $c(x)$ is the distance between x and $h_2(x)$ under h_1. We now prove that $c(x)$ is independent of x, that is, a constant. Since h_1 and h_2 are two consistent functions, $h_1(h_2(x)) = h_2(h_1(x))$ for any $x \in T$. Let $y = h_1(x)$, then $h_1(h_2(x)) = h_1^{c(x)+1}(x)$ and $h_2(h_1(x)) = h_2(y) = h_1^{c(y)}(y) = h_1^{c(y)+1}(x)$.

Thus we have $c(x) = c(y)$. This means that $c(x)$ is a constant for all elements x in T.

(2) If T is not a primary permutation set under h_1, let T have two primary permutation sets under h_1 without loss of generality; that is, $T_1 = \{a, a^2, \cdots, a^{m_1}\}$, and $T_2 = \{b, b^2, \cdots, b^{m_2}\}$. We then have the following three cases for h_2:

- T is a primary permutation set under h_2. This is the case of (1) above.
- T has the same primary permutation sets under h_2. Obviously, we can consider every primary permutation set separately; we then have case (1).
- T has different primary permutation sets under h_2. Then, there exists an element $a \in T_1$ such that $h_2(a) \in T_2$. For every element $x = a^{i+1} \in T_1$, $h_2(h_1(a^{i+1})) = h_2(h_1^{i+1}(a)) = h_1^{i+1}(h_2(a)) \in T_2$.

  However, $h_1(a^{i+1}) \in T_1$ may be any element in T_1; hence, h_2 maps all elements in T_1 to T_2. This result is obviously applicable to T_2.

Therefore, T is partitioned into the same primary permutation sets under h_2 as under h_1. This contradicts the assumption. □

Theorem 4.6 Let $P \in \Sigma_3$. Then, P can be transformed into a new program Q such that $Q \in \Sigma_1$ or $Q \in \Sigma_2$.

Proof. Assume that the permutation position in $perm(P)$ is $p[n-k+1], \cdots, p[n]$, which is placed at the end of p, and that $\{e_1, \cdots, e_{m_3}\}$ is a partition of the rule set of P under 3-connection. Let $e_i = \{r_{i_1}, \cdots, r_{i_{m_i}}\}$. Clearly, for permutation positions, its values can be derived from the trails of application of rules. We define a new program Q with a new recursive predicate q where q is obtained by deleting $perm(p)$ from p and inserting an ID to record the tuple identifier of the exit relation, and Q is obtained by replacing p by q for each rule r in P. Then,

$$p(X_1, \cdots, X_n) : - \quad q(X_1, \cdots, X_{n-k}, ndx, -), \quad h_{ndx}(q_0(X_{n-k+1}, \cdots, X_n, -))$$
$$q_0(X_{n-k+1}, \cdots, X_n, -) : - \quad exit(X_1, \cdots, X_n, -), \qquad (7)$$

where h_{ndx} is the composition of a set of permutation functions $h_1, \cdots, h_l$ that reorder the position of $X_{n-k+1}, \cdots, X_n$ in q_0, and "$-$" is the tuple identifier of the exit relation.

$P \in \Sigma_3$; hence, for any permutation function h_i and h_j, there is $h_i(h_j) = h_j(h_i)$. Hence

$$h_{ndx} = h_{ndx_1} \cdots h_{ndx_{m_3}}$$

where ndx_i is a sequence of numbers in $\{i_1, \cdots, i_{m_i}\}$.

This formula means that the index can be replaced by m_3 sub-indexes, each sub-index ndx_i traces the application of rules in e_i. Therefore, the new predicate q can be replaced by $q(X_1, \cdots, X_{n-k}, ndx_1, \cdots, ndx_{m_3}, -)$, while h_{ndx} can be replaced by $h_{ndx_1} \cdots h_{ndx_{m_3}}$.

For each rule $r_{i_j} \in e_i$, the new rule r'_{i_j} in Q is

$$q(\ T_i, Y, ndx_1, \cdots, ndx'_i, \cdots, ndx_{m_3}, -) : - \quad a_{i_j}(T_i, A_i), \quad q(A_i, Y, ndx_1, \cdots, ndx_i, \cdots, ndx_{m_3}, -), \quad ndx'_i = ndx_i || i_j,$$

where T_i is the vector of arguments in the 3-dyn_i, $Y = pers(r_{i_j})$, and "$||$" means concatenation of the index string.

Obviously, program Q is in Σ_1 if $Y = \phi$, otherwise Q is in Σ_2. The partition is also $\{e_1, \cdots, e_{m_3}\}$, and $e_i = \{r_{i_1}, \cdots, r_{i_{m_i}}\}$, but ndx_i is added to the recursive predicate q_i as its dynamic position. That is, the program Q can be decomposed into m_3 subprograms in which $Q_i = \{r'_{i_1}, \cdots, r'_{i_{m_i}}\}\ (i = 1, \cdots, m_3)$ is defined as follows:

$$r'_{i_j} : \ q_i(T_i, ndx'_i, -) : -a_{i_j}(T_i, A_i), \quad q_i(A_i, ndx_i, -), ndx'_i = ndx_i || i_j. \quad (8)$$

The recursive predicate q in Eq. (7) is

$$q(T_1, \cdots, \ T_{m_3}, ndx, -) : - \quad \bowtie_{i=1}^{m_3} (q_i(T_i, ndx_i, -)), \quad ndx = ndx_1 || \cdots || ndx_{m_3}. \quad (9)$$

Therefore, P is decomposable. □

We now consider how to compute the indexes in the equations. Although the coding method proposed in the counting method [1] is available, it is possible, in this case, to simplify the evaluation of the indexes by using the properties of permutation.

Assume that h_1 is not an identity function, and that $h_i = h_1^{d_i}\ (i = 2, \cdots, m)$. d_i is called the distance between h_i and h_1, or "distance" for short. Consider the subprogram Q_i that includes the rules $\{r'_{i_1}, \cdots, r'_{i_{m_i}}\}$. Assume that the sub-index ndx_i that records the trace of application of rules in Q_i is $j_1, \cdots, j_k$. Then

$$\begin{aligned} h_{ndx_i} &= h_{j_1} \cdots h_{j_k} \\ &= h_{i_1}^{l_1} \cdots h_{i_{m_i}}^{l_{m_i}} \\ &= (h_1^{d_1})^{l_1} \cdots (h_1^{d_{m_i}})^{l_{m_i}} \\ &= h_1^{(\sum_{j=1}^{m_i}(d_j.l_j))} \\ &= h_1^{mod(\sum_{j=1}^{m_i}(d_j.l_j),c)}, \end{aligned}$$

where l_j is the number of rules r_{i_j} that appear in the sub-index ndx_i, and c is a constant defined by Lemma 4.5. Therefore we can simplify the rules in Eq. (8) as follows:

$$r'_{i_j} : q_i(T_i, I'_i, -) : -a_{i_j}(T_i, A_i), \quad q_i(A_i, I_i, -), \quad I'_i = mod(I_i + d_j, c).$$

Similarly,

$$\begin{aligned} h_{ndx} &= h_{ndx_1} \cdots h_{ndx_{m_3}} \\ &= h_1^{mod(\Sigma_{i=1}^{m_3}(count_i),c)} \end{aligned}$$

where $count_i = mod(\sum_{j=1}^{m_i}(d_j.l_j), c)$, $i = 1, \cdots, m_3$.

Equation (9) can be simplified as follows:

$$q(T_1,\cdots,T_{m_3},I,-):-\bowtie_{i=1}^{m_3}(q_i(T_i,I_i,-),\quad I=mod(\Sigma_{i=1}^{m_3}(I_i),c).$$

Example 4.7 Consider the following program P, which contains a permutation set:

$$\begin{aligned}
r_0 &: p(X,Y,Z,U,V,W):-\\ &\qquad q(X,Y,Z,U,V,W)\\
r_1 &: p(X,Y,Z,U,V,W):-a(X,A),\\ &\qquad p(A,Y,Z,U,V,W)\\
r_2 &: p(X,Y,Z,U,V,W):-b(Y,B),\\ &\qquad p(X,B,Z,V,W,U)\\
r_3 &: p(X,Y,Z,U,V,W):-c(Z,C),\\ &\qquad p(X,Y,C,W,U,V).
\end{aligned}$$

$\{e_1=\{r_1\},e_2=\{r_2\},e_3=\{r_3\}\}$ is the partition of P under 3-connectivity. The permutation position set is $\{4,5,6\}$. Let h_i be the permutation function for rule r_i $(i=1,2,3)$; then

$$\begin{aligned}
&h_1(4)=4,\quad h_1(5)=5,\quad h_1(6)=6\\
&h_2(4)=6,\quad h_2(5)=4,\quad h_2(6)=5\\
&h_3(4)=5,\quad h_3(5)=6,\quad h_3(6)=4.
\end{aligned}$$

Therefore, we have $h_1=h_2^3$, and $h_3=h_2^2$; that is, $d_1=3,d_2=1$, and $d_3=2$. Moreover, $c=3$. Hence the decomposed subprograms are

$$\begin{aligned}
D_1 &: q_1(X,0,-):-q(X,Y,Z,U,V,W,-)\\
&\quad q_1(X,I_1,-):-a(X,A),q_1(A,I_1,-)\\
D_2 &: q_2(Y,0,-):-q(X,Y,Z,U,V,W,-)\\
&\quad q_2(Y,I_2',-):-b(Y,B),q_2(B,I_2,-),\\
&\qquad I_2'=mod(I_2+1,3)\\
D_3 &: q_3(Z,0,-):-q(X,Y,Z,U,V,W,-)\\
&\quad q_3(Z,I_3',-):-c(Z,C),q_3(C,I_3,-),\\
&\qquad I_3'=mod(I_3+2,3)
\end{aligned}$$

and

$$\begin{aligned}
q(X,Y,Z,I,-)&:-q_1(X,I_1,-),q_2(Y,I_2,-),\\
&\quad q_3(Z,I_3,-),\\
&\quad I=mod(I_1+I_2+I_3,3).
\end{aligned}$$

The final result is

$$\begin{aligned}
&p(X,Y,Z,U,V,W):-q(X,Y,Z,I,-),\\
&\qquad q_0(I,U,V,W,-)\\
&q_0(0,U,V,W,-):-exit(X,Y,Z,U,V,W,-)\\
&q_0(1,U,V,W,-):-exit(X,Y,Z,V,W,U,-)\\
&q_0(2,U,V,W,-):-exit(X,Y,Z,W,U,V,-).
\end{aligned}$$

5. Related Work and Conclusions

In this paper, we have extended the decomposable program classes proposed by Wang, et al.[7] in two aspects:

(1) Extracting semi-persistent positions from dynamic positions. In Wang, et al.'s definition, persistent positions contain only those semi-persistent positions in which the variables are bounded with a value by the query. The reason for this is that the shared variables in such positions can be replaced by the bound values of Q. Therefore, their method for program decomposition is query-dependent. In contrast, our method is query-independent. It treats semi-persistent positions as an independent class.

(2) Extracting permutation positions from dynamic positions. By introducing a special indexing technique based on the properties of permutation positions, our method can separate permutation variables so as to reduce the arity of recursive predicates. In contrast, Wang, et al.'s method treats all permutation positions in same way as general dynamic positions.

As in Wang, et al.'s method, queries can be decomposed into subqueries on subprograms, and thus magic rewriting, a powerful query optimization technique in recursive program processing, can also be applied to our decomposed programs without any changes being required.

Although Σ_2 and Σ_3 are currently the two largest known decomposable program classes, it is still unknown whether there are any larger decomposable program classes. Further investigation is necessary.

Acknowledgments The work is supported in part by The Hori Information Science Promotion Foundation.

References

1) Beeri, C. and Ramakrishnan, R.: On the Power of Magic, *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symp.Principles of Database Systems (PODS)*, pp.269–283 (1987).

2) Naughton, J.: One-Sided Recursions, *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems (PODS)*, pp.340–348 (1987).

3) Naughton, J.: Compiling Separable Recursions, *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems (PODS)*, pp.312–319 (1988).

4) Naughton, J., Ramakrishnan, R., Sagiv, Y. and Ullman, J.D.: Efficient Evaluation of Right-, Left-, and Mixed-Linear Rules, *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp.235–242 (1989).

5) Naughton, J., Ramakrishnan, R., Sagiv, Y. and Ullman, J.D.: Argument Reduction by Factoring, *Proc. 15th Int. Conf. on Very Large Data*

Bases, pp.173–182 (1989).
6) Ullman, J.D.: *Principles of Database and Knowledge-Base Systems*, Vol.II, Computer Science Press (1989).
7) Wang, K., Zhang, W. and Chou, S.C.: Decomposition of Magic Rewriting, *JACM*, Vol.42, No.2, pp.329–381 (1995).

(Received May 7, 1996)
(Accepted October 1, 1996)

Zhibin Liu Zhibin Liu received the B.E. degree in computer science and technology from Tianjin University, Tianjin, China in 1985, and the M.E. degree in information and computer science from the People's University of China, Beijing, China in 1988. From 1990 to 1992 he was a lecturer of the Department of Computer Science and Technology at Tianjin University. From 1992 to 1995, he was a lecturer in the Department of Computer Science and Technology at Beijing Polytechnic University. He is now pursuing the Doctor of Engineering degree at the Nagoya Institute of Technology, Nagoya, Japan. His current research interests include algorithm design and analysis, deductive databases, database integration and heterogeneous database systems.

Xiaoyong Du received the B.S. degree in computational mathematics from Hangzhou University, Zhejing, China in 1983, and the M.E. degree in information and computer science from the People's University of China, Beijing, in 1988. From 1989 to 1992, he was a lecturer in the Institute of Data and Knowledge Engineering at the People's University of China. He is now a Ph.D. candidate in the Department of Intelligence and Computer Science at the Nagoya Institute of Technology, Nagoya, Japan. His current research interests include databases and artificial intelligence. He is a member of the IPSJ.

Naohiro Ishii received the B.E. and M.E., and Doctor of Engineering degrees in electrical and communication engineering from Tohoku University, Sendai, in 1963, 1965, and 1968, respectively. From 1968 to 1974 he was at the School of Medicine in Tohoku University, where he worked on information processing in the central nervous system. Since 1975 he has been with the Nagoya Institute of Technology, where he is a professor in the Department of Electrical and Computer Engineering. His current research interests include databases, software engineering, algorithm design and analysis, nonlinear analysis of neural network and artificial intelligence. He is a member of the IPSJ, IEICE, ACM, and IEEE.