Minimize Linear Mutual Recursion by Rule Unfolding

Ke Wang

Department of ISCS
National University of Singapore
Singapore 0511

Weining Zhang and Siu-Cheung Chau

Department of Mathematics
University of Lethbridge
Lethbridge, Alberta, Canada, T1K 3M4

Abstract

In this paper, we minimize the number of recursive predicates in a linear mutual recursion by the rule unfolding technique. The minimization is based on a newly proposed syntactic equivalence called the unfolding-equivalence. Algorithms for testing unfolding-equivalence and constructing an unfolding-equivalent recursion with the fewest possible number of recursive predicates are presented.

1 Introduction

Unfolding a rule is simply to replace a derived subgoal in the rule body with the body of an instance of a rule defining that subgoal. The rule unfolding technique has been used in many occasions to optimize evaluation of logic programs, with the most popular use in the top-down evaluation in Prolog. But to our knowledge, a systematic use of rule unfolding for reducing and minimizing the number of recursive predicates and the number of recursive rules has not been investigated. The importance of reducing these numbers is obvious: while more recursive predicates mean more interactions among relations in computation, more recursive rules means more work to be done in each iteration. It is generally believed that, given the same data, a program with fewer recursive predicates and fewer recursive rules can be evaluated more efficiently than a program with more recursive predicates and more recursive rules. In this paper, we study the role of the unfolding technique in optimizing deductive databases through reducing these numbers.

We study the problem of minimizing the number of recursive predicates in a linear mutual recursion, that is, constructing an equivalent recursion that has the fewest number of recursive predicates. This problem in its entirety is undecidable as it subsumes the undecidable boundedness problem which tests whether a recursion is equivalent to a non-recursive program [1, 3, 5, 10]. We therefore modify this problem by restricting to recursions that are obtained from unfolding original rules and are equivalent by having the same set of sequences of rule applications in expanding the query predicates. Formal notions of unfolded programs and unfolding-equivalence are proposed to capture these concepts in Section 3. We reduce the unfolding-equivalence to the equivalence of finite automata and present a construction of a minimized recursion for a given linear mutual recursion. The study shows that the minimization strictly reduces the number of recursive predicates as long as not every recursive predicate has a direct recursive rule.

2 A Motivating Example

The following example shows how rule unfoldings reduce the number of recursive predicates and rules and improves the efficiency of evaluation.

Example 2.1 Assume a recursion $P$ has rules

$r_1 : p(x, y) :- q(x, z), a(x, y)$
$r_2 : q(x, y) :- s(x, y, z), b(x, y)$
$r_3 : s(x, y, z) :- p(x, z), c(x, y)$
$r_4 : p(x, y) :- d(x, y)$

where predicates $p, q, s$ are mutually recursive with one another. Suppose that $p$ is the query predicate. The definition of $p$ is completely determined by all possible sequences $(r_1 r_2 r_3)^k r_4$ ($k \geq 0$) of rules that expand the predicate $p$, where $(r_1 r_2 r_3)^k$ is the $k$ times repetition of the sequence $r_1 r_2 r_3$. By abuse of notation, let $r_1 r_2 r_3$ also denote the rule obtained from unfolding the recursive subgoal in $r_1$ with $r_2$ and unfolding the recursive subgoal in the result rule with $r_3$ in that order; that is,

$r_1 r_2 r_3 : p(x, y) :- p(x, z_0), c(z_0, x), b(z_0, x), a(x, y)$
where all variables are distinct. Clearly, each backward application of rule \( r_1 r_2 r_3 \) is equivalent to the backward applications of three rules \( r_1, r_2, r_3 \) in that order. Thus, \( P \) is equivalent to the recursion \( P' = \{ r_1 r_2 r_3, r_2 \} \) with respect to \( P \). \( P' \) is superior to \( P \) for two reasons. First, in most cases \( P' \) can be evaluated more efficiently than \( P \): Originally in \( P \) three iterations are needed to go through the recursion \( \{ r_1, r_2, r_3 \} \) once, now in \( P' \) only one iteration is needed. In other words, each iteration of \( P' \) is accomplishing the work of three iterations of \( P \) in terms of tuples derived. The amount of work in each iteration of \( P' \) can be reduced by replacing the non-recursive subgoals \( c(x_0, z), b(x_0, z), a(x, y) \) in rule \( r_1 r_2 r_3 \) with a single subgoal \( c(x_0, y) \), where predicate \( c \) is defined as

\[
p(x, y) : -p(x, z), c(x_0, y)
\]

\[
e(x_0, y) : -c(x_0, z), b(x_0, z), a(x, y)
\]

\[
r_p
\]

where the instantiation of the subgoals \( c(x_0, z), b(x_0, z), a(x, y) \) is evaluated only once outside the loop for the recursive rule. The new recursive rule

\[
p(x, y) : -p(x, z), c(x_0, y)
\]

is expected to be evaluated more efficiently than \( r_1 r_2 r_3 \) in most cases.

The second reason for us to believe that \( P' \) is superior to \( P \) is the simplicity of \( P' \): only one recursive predicate and one recursive rule, i.e., a sirup. Such a simple syntax usually allows high performance evaluation algorithms to be applied [6, 7, 4, 8]. If \( q \) and \( s \) are also query predicates, by adding the non-recursive rules \( r_2 r_3 \) and \( r_3 \) to \( P' \), their equivalences can be preserved as well, and the result recursion still has the same nice property as before, i.e., one recursive predicate and one recursive rule.

Motivated by this example, naturally one would like to know how far it can go to reduce the number of recursive predicates based on rule unfoldings. We try to answer this question in the rest of the paper. As the first step, we define the notion of unfolding-equivalence.

3 Unfolding-equivalence of Recursions

Without lose of generality, we consider only linear recursions that are mutually recursive. Let \( r \) and \( r' \) be rules such that \( A \) is the recursive subgoal in \( r \) and the predicate for \( A \) is defined by \( r' \). By renaming of variables, we assume that \( r \) and \( r' \) have no common variables. \( r \) can be unfolded by \( r' \) as follows: Let \( \theta \) be the unifier of the head of \( r' \) and subgoal \( A \), in which variables or constants in \( A \) always replace variables in the head of \( r' \) (by rectifying the head first [9]). Then replace the subgoal \( A \) in \( r \) with the body of \( r' \theta \). \( rr' \) denotes the result rule.

Definition 3.1 Let \( P \) be a linear mutual recursion. An unfolded rule wrt \( P \) is defined inductively as follows. Every rule in \( P \) is an unfolded rule wrt \( P \). Suppose that \( r_1 \ldots r_{k-1} (k > 1) \) is an unfolded rule wrt \( P \), where \( r_i \) are rules in \( P \), and that \( r_1 \ldots r_{k-1} \) has a recursive subgoal whose predicate is defined by a rule \( r_\k \) in \( P \). Then the rule obtained from unfolding \( r_1 \ldots r_{k-1} \) by \( r_\k \), denoted \( r_1 \ldots r_{k-1} r_\k \), is an unfolded rule wrt \( P \). An unfolded rule \( r_1 \ldots r_\k \) is an exit if \( r_\k \) is an exit rule in \( P \). An unfolded recursion wrt \( P \) is a finite set of unfolded rules wrt \( P \).

In other words, an unfolded rule wrt \( P \) is a string of names of rules in \( P \) that specifies a legal unfolding order. Consider the recursion \( P \) in Example 2.1, every rule in \( P \) is an unfolded rule wrt \( P ; r_1 r_2 \) and \( r_1 r_3 \) are unfolded rules wrt \( P ; (r_1 r_2)^2 r_\k \) (\( k > 0 \)) are unfolded rules wrt \( P \) and, in fact, are exits; and so on. However, \( r_1 r_3 \) or \( r_1 r_2 \) are not unfolded rules wrt \( P \). In the following, \( h \) denotes the mapping from unfolded rules to their string representation. In particular, if \( s \) is the name of an unfolded rule wrt \( P \) specified by the string \( r_1 \ldots r_\k \) of rules in \( P \), then \( h(s) = r_1 \ldots r_\k \).

Definition 3.2 Assume \( U \) is an unfolded recursion wrt \( P \), where each rule in \( U \) is named \( s_1 \). Let \( p \) be an IDB predicate in \( U \). A word for \( p \) wrt \( U \) is a string of rules in \( P \) of the form \( h(s_1) \ldots h(s_k) \), where \( s_1 \ldots s_k \) is an unfolded rule wrt \( U \) such that \( s_1 \) defines the predicate \( p \) and \( s_k \) is an exit. word\( (U, p) \) denotes the set of all words for \( p \) wrt \( U \).

Intuitively, a word for \( p \) wrt \( U \) is a sequence of rules in \( P \) that "corresponds" to some sequence of rules in \( U \) for expanding the definition of \( p \). Obviously, if two unfolded recursions wrt \( P \) define exactly the same set of words for a predicate, then they are equivalent wrt that predicate. This motivates the following definition.

Definition 3.3 Let \( U, U' \) be two unfolded programs wrt \( P \) and let \( \Delta \) be a set of IDB predicates in both \( U \) and \( U' \). \( U \) is unfolding-equivalent to \( U' \) wrt \( \Delta \) if word\( (U, p) = \text{word}(U', p) \) for every \( p \in \Delta \).

Example 3.1 Let a recursion \( P \) have recursive rules

\[
r_1 : q : -p, e
\]
\( r_2 : p : -s, c \\
(2) r_3 : q : -s, d \\
(3) r_4 : s : -q, b \\
and exit rules \( r_p, r_q, r_r \) for predicates \( p, q, s \), respectively, where all arguments are omitted for simplicity. The left of Figure 1 shows the dependency graph \( G(P) \). First consider the definition of predicate \( s \). Each word in word\((P, s)\) is given by a string of form \( l_1 \ldots l_k l_{k+1} \), where \( l_1, \ldots, l_k \) are labels along a path starting at \( s \) and \( l_{k+1} \) is the exit rule for the predicate at which the path ends. There are two simple cycles in \( G(P) \), i.e., \( s \rightarrow q \rightarrow p \rightarrow s \) and \( s \rightarrow q \rightarrow s \). Substrings of a word generated by these cycles can be generated by unfolded rules \( r_q r_p r_2 \) and \( r_q r_2 \), which are constructed by unfolding rules along these simple cycles. Thus, word\((P, s)\) = word\((U, s)\), where \( U \) is the unfolded recursion \( \{r_q r_1 r_2, r_q r_3, r_q r_4 r_5, r_q r_1 r_p\} \). So by definition 3.3, \( U \) is unfolding-equivalent to \( P \) wrt \( \{s\} \).

Once \( s \) is defined, predicates \( q \) and \( p \) can be defined easily in terms of \( s \) and EDB predicates by non-recursive rules according to the dependency graph \( G(P) \). For instance, the dependency graph tells that \( q \) can be defined in terms of \( s \) through unfolded rules \( r_3 \) and \( r_2 r_3 \) and defined in terms of EDB predicates through the exits \( r_q \) and \( r_q r_2 \). Therefore, to preserve equivalence of \( q \) we add these unfolded rules to \( U \). Similarly, to preserve equivalence of \( p \) we add unfolded rules \( r_2 \) and \( r_3 \) to \( U \). The result recursion \( U \) now looks like

\[ \text{rules for } s: \quad r_q r_1 r_2, r_q r_3, r_q r_5, \quad r_q r_4 r_5, r_q r_1 r_p \]
\[ \text{rules for } q: \quad r_3, r_1 r_2, r_2 r_1, r_1 r_p \]
\[ \text{rules for } p: \quad r_2, r_3 \]

\( U \) is unfolding-equivalent to \( P \) wrt all IDB predicates \( p, q, s \), but it has fewer recursive predicates and recursive rules than \( P \), i.e., one recursive predicate \( s \) and two recursive rules \( r_q r_1 r_2 \) and \( r_q r_3 \). The dependency graph of \( U \) is given on the right of Figure 1.0

![Figure 1](image)

Now we show the decidability of unfolding-equivalence by reducing it to equivalence of finite automata.

Assume that \( U \) is an unfolded recursion wrt \( P \) and \( p \) is an IDB predicate in \( U \). We show that word\((U, p)\) is a regular set on the alphabet of rules in \( P \) and thus unfolding-equivalence is reduced to equivalence of finite automata. For the given \( U \) and \( p \), we define a finite automata \( M(U, p) = (Q, \Sigma, \Delta, p, q_f) \), such that word\((U, p)\) is the language defined by \( M(U, p) \): \( p \) is the initial state, \( q_f \) is a new symbol denoting the final state, \( \Sigma \) is the set of input symbols consisting of the names of unfolded rules in \( U \), \( Q \) is the set of states consisting of all IDB predicates of \( U \) plus the final state \( q_f \). The transition function \( \delta \) is defined as follows: for an input symbol \( s_i \) (a rule in \( U \)) in \( \Sigma \) with the head predicate \( p \), if some IDB predicate \( q \) of \( U \) is found in the body of \( s_i \) then \( \delta(p, s_i) = q \); otherwise, \( \delta(p, s_i) = q_f \). \( L(U, p) \) denotes the regular set defined by the automata \( M(U, p) \). Assume that, for each input symbol \( s_i \in \Sigma \), \( h(s_i) \) is the string representation for the unfolded rule named \( s_i \), \( h \) can be extended to strings \( s_1 \ldots s_n \) of input symbols and to languages \( L \) as follows. \( h(s_1 \ldots s_n) = h(s_1) \ldots h(s_n) \), and \( h(L) = \cup_{x \in L} h(x) \). We have

Theorem 3.1 Let \( U, p \) be specified as above. Then (1) word\((U, p) = h(L(U, p)) \), (2) word\((U, p) \) is a regular set, (3) unfolding-equivalence is decidable.

4 Minimization of Linear Mutual Recursions

The minimization problem of recursions is defined as follows.

Definition 4.1 Assume that \( P \) is a linear mutual recursion. Let \( U \) be an unfolded recursion wrt \( P \) and let \( \Delta \) be a non-empty set of IDB predicates in \( U \). \( U \) is a minimized recursion of \( P \) wrt \( \Delta \) if \( U \) is unfolding-equivalent to \( P \) wrt \( \Delta \) and \( U \) has the fewest possible number of recursive predicates. The minimization problem is to find a minimized recursion for a given linear mutual recursion.\( \square \)

We assume that at least one predicate in \( \Delta \) reaches a recursive predicate in the dependency graph \( G(P) \), otherwise \( P \) is equivalent to the set of non-recursive rules in \( P \) wrt \( \Delta \). The solution to the minimization problem is closely related to the following concepts. A simple cycle in \( G(P) \) is covered by a node if it contains that node. A set \( V \) of nodes in \( G(P) \) is called a minimum cycle cover of \( G(P) \) if \( V \) contains the fewest possible number of nodes so that every simple cycle in \( G(P) \) is covered by at least one node in \( V \). Let \( X \) be a
minimum cycle cover of \( G(P) \). A simple cycle in \( G(P) \) is \( X \)-dedicated to a node \( v \) in \( X \) if it is covered by \( v \), but not by any other node in \( X \). By the minimality of \( X \), for each node \( v \) in \( X \), at least one simple cycle in \( G(P) \) is \( X \)-dedicated to \( v \).

**Example 4.1** In the graph on the left of Figure 1, there are only two simple cycles \( q \rightarrow p \rightarrow s \rightarrow q \) and \( q \rightarrow s \rightarrow q \). \( X_1 = \{q\} \) and \( X_2 = \{s\} \) are minimum cycle covers, but not \( \{p\} \) because simple cycle \( q \rightarrow s \rightarrow q \) is not covered by \( p \). Both simple cycles are \( X_i \)-dedicated to the node in \( X_i \), \( i = 1, 2 \).

We now sketch out the intuitive idea of finding a minimized recursion of \( P \) wrt \( \Delta \).

First, we find a minimum cycle cover of \( G(P) \). A general algorithm for this can be obtained by reducing this problem to the prime implicant covering problem in switching circuit theory [2]: Given a set \( F \) of prime implicants, find a subset \( M \) of \( F \) so that all minterms covered by \( F \) are covered by \( M \) and \( M \) has the fewest possible number of prime implicants. The reduction goes as follows: each simple cycle in \( G(P) \) is mapped to a distinct minterm and each node \( v \) of \( G(P) \) is mapped to a distinct prime implicant that covers exactly the minterms corresponding to the simple cycles covered by \( v \). Clearly, a node covers a simple cycle if and only if the corresponding prime implicant covers the corresponding minterm. Thus, a minimum cycle cover of \( G(P) \) is given by a solution to the reduced prime implicant covering problem. From now on, we assume that a minimum cycle cover \( X \) of \( G(P) \) is found.

Second, we construct an unfolded recursion that is unfolding-equivalent to \( P \) wrt \( X \) and in which only predicates in \( X \) are recursive. By the definition, we need to construct unfolded rules (wrt \( P \)) that generate exactly words in \( \text{word}(P,p) \) for every \( p \in X \).

We borrow the dependency graph \( G(P) \) for the transition diagram of the finite automata \( M(P,p) \). Each word in \( \text{word}(P,p) \) can be generated by traversing a path in \( G(P) \) from \( p \) to a node that has an exit rule; a word generated by such a path is given by an element of the regular set defined by the regular expression \( l_1 \ldots l_k \) where \( l_1, \ldots, l_k \) are the sequence of labels along the path. Therefore, each word in \( \text{word}(P,p) \) can be partitioned into substrings for cycles and substrings for non-cycle paths in \( G(P) \). To generate substrings for cycles, we construct the following recursive unfolded rules: For each recursion corresponding to a simple cycle \( X \)-dedicated to \( p \), an unfolded rule for \( p \) corresponding to this simple cycle is constructed; and for each pair of predicates in \( X \), unfolded rules corresponding to simple paths between them through only nodes not in \( X \) are constructed. To generate substrings for non-cycle paths, we construct non-recursive unfolded rules for \( p \) corresponding to simple paths from \( P \) to EDB predicates through only nodes not in \( X \). The details of these constructions are given in steps 1, 2, and 3 in Algorithm 1 at the end of the paper.

Third, once the predicates in \( X \) are defined, the query predicates in \( \Delta \) can be defined non-recursively in terms of predicates in \( X \) and EDB predicates, based on the dependency in \( G(P) \). The minimality of the constructed recursion is proved in Theorem 4.1 below.

To present the algorithm, we need some notation. Let \( \lambda : p_1 \rightarrow p_2 \ldots \rightarrow p_k (k > 1) \) be a path in \( G(P) \) with edge labels \( l_1, l_2, \ldots, l_{k-1} \). Recall that each label \( l_i \) is a regular expression of form \( r_1 + \ldots + r_m \), where \( r_j \) are the rules in \( P \) with \( p_i \) in the head and \( p_{i+1} \) in the body. \( \text{un} \text{fold}(\lambda) \) denotes the regular expression \( l_1 \ldots l_{k-1} \), i.e., the concatenation of \( l_1, \ldots, l_{i-1} \), and \( \text{L(un} \text{fold}(\lambda)) \) denotes the regular set given by \( \text{un} \text{fold}(\lambda) \). Intuitively, \( \text{L(un} \text{folded}(\lambda)) \) is the set of unfolded rules (represented as strings of rules in \( P \)) obtained from unfoldings along path \( \lambda \). Let \( \lambda_1, \ldots, \lambda_n \) be all simple paths from \( p_1 \) to \( p_k \) in \( G(P) \) through only nodes not in \( X \). \( \text{exit}_X(p_1,p_k) \) denotes the regular expression

\[
(\text{un} \text{fold}(\lambda_1) + \ldots + \text{un} \text{fold}(\lambda_n))\text{exit}_X(p_2,p_k),
\]

where \( \text{exit}_X(p_2,p_k) \) is the regular expression denoting the set of exit rules for \( p_k \) in \( P \). In other words, \( \text{exit}_X(p_1,p_k) \) is the set of exit unfolded rules for \( p_1 \) obtained from unfoldings along these paths. Notice that if \( \text{exit}_X(p_2,p_k) = \emptyset \) or \( n = 0 \) then \( \text{exit}_X(p_1,p_k) = \emptyset \).

A construction of a minimized recursion is given by Algorithm 1 at the end of the paper.

**Theorem 4.1** Algorithm 1 is correct.

**Example 4.2** Consider a linear mutual recursion \( P \) with four recursive predicates \( p,q,s,t \), six recursive rules \( r_1, \ldots, r_6 \), and four exit rules \( r_7, r_8, r_9, r_{10} \) for \( p,q,s,t \), respectively. The dependency graph \( G(P) \) is
shown on the left of Figure 2. \( X = \{ p, q \} \) is a minimum cycle cover of \( G(P) \). The simple cycle \( p \to s \to p \) is \( X \)-dedicated to \( p \) and the simple cycle \( q \to t \to q \) is \( X \)-dedicated to \( q \). The following recursive unfolded rules are constructed

\[ \text{step 2:} \quad r_1r_2, \quad r_3r_4 \]
\[ \text{step 3:} \quad r_1r_3r_4, \quad r_6 \]

The following exit unfolded rules are constructed:

\[ \text{step 1:} \quad r_9, \quad r_9r_9, \quad r_1r_3r_4, \quad r_9, \quad r_9r_9 \]

Let \( U \) be the set of above unfolded rules. It is easy to see that \( U \) is unfolding-equivalent to \( P \) wrt \( X \). To make \( U \) unfolding-equivalent to \( P \) wrt all recursive predicates \( \{ p, q, s, t \} \), we add to \( U \) the non-recursive unfolded rules:

\[ \text{step 4:} \quad r_5, \quad r_3r_5, \quad r_5 \]
\[ \text{step 5:} \quad r_5, \quad r_3r_5, \quad r_4 \]

\( U \) is a minimized recursion of \( P \) wrt \( \{ p, q, s, t \} \). Notice that \( U \) has only two recursive predicates and four recursive rules. The dependency graph of \( U \) is shown on the right of Figure 2.

References

[2] A.D. Friedman, P.R. Menon, Theory & design of switching circuits, CSP, Rockville, Maryland, 1975

Algorithm 1: Find a minimised recursion of a linear mutual recursion.

Input: A linear mutual recursion \( P \) and a non-empty set \( \Delta \) of IDB predicates, where at least one predicate in \( \Delta \) reaches a recursive predicate in \( G(P) \).

Output: A minimised recursion of \( P \) wrt \( \Delta \).

Method:

1. For \( j = 1 \) to \( k \) do begin
   \( U \leftarrow U \cup L(\text{exit}_X(q_j, q_j)) \);
   for each \( p \notin X \) do \( U \leftarrow U \cup L(\text{exit}_X(q_j, p)) \);
   end;

2. For \( j = 1 \) to \( k \) do begin
   \( U \leftarrow U \cup L(\text{fold}(c)) \);
   for each \( q_j \) to \( q_j \) in \( Y_j \) do \( U \leftarrow U \cup L(\text{fold}(c)) \);
   end;

3. For \( j = 1 \) to \( k \) do begin
   \( U \leftarrow U \cup L(\text{fold}(\lambda)) \);
   for each \( q_j \) to \( q_j \) through only nodes not in \( X \) do \( U \leftarrow U \cup L(\text{fold}(\lambda)) \);
   end;

4. For each \( p \in \Delta - X \) do
   for each IDB predicate \( q \notin X \) do \( U \leftarrow U \cup L(\text{exit}_X(p, q)) \);

5. For each \( p \in \Delta - X \) do
   for \( j = 1 \) to \( k \) do begin
   for each \( p \) to \( q_j \) through only nodes not in \( X \) do \( U \leftarrow U \cup L(\text{fold}(\lambda)) \);
   return \( U \).