Compiling Separable Recursions

Jeffrey F. Naughton
Princeton University

Abstract

In this paper we consider evaluating queries on relations defined by a combination of recursive rules. We first define separable recursions. We then give a specialized algorithm for evaluating selections on separable recursions. Like the Generalized Magic Sets and Generalized Counting algorithms, this algorithm uses selection constants to avoid examining irrelevant portions of the database; however, on some simple recursions this algorithm is $O(n)$, whereas Generalized Magic Sets is $\Omega(n^2)$ and Generalized Counting is $\Omega(2^n)$.

1 Introduction

Evaluation algorithms for queries on recursively defined relations can be divided into two classes: those intended for a specific class of recursion, and those intended for general recursions. This paper continues research into algorithms for special classes of recursions. Developing specialized algorithms in addition to general algorithms will be a "win" if the classes for which we have specialized evaluation algorithms are frequently used, and the specialized evaluation algorithms are better than the general algorithms on these recursions.

In this paper we identify a class of recursions called "separable recursions." While it is impossible to be certain how frequently separable recursions will appear in as-yet unavailable systems, initial studies of logic programs suggest that they will be common. Furthermore, for selections on separable recursions, our evaluation algorithm can outperform popular general evaluation algorithms by a factor proportional to the size of the database.

Separable recursions are a subset of linear recursions, and include non-chain rules and non-binary predicates. Section 2 gives a precise definition of separable recursions; here we present two examples.

Example 1.1 Suppose that we have a relation *perfectFor*(X,Y) of people X and products Y such that Y is perfect for X, and also two relations *friend*(X,Y) and *idol*(X,Y) of people X and their friends and idols Y. Furthermore, suppose that a person will buy a product if it is perfect for them, or if their friend or idol has bought it. Then the following recursion defines a relation *buys*(X,Y) of people X and products Y that they buy.

r_1: *buys*(X,Y) :- *friend*(X,W) & *buys*(W,Y).
r_2: *buys*(X,Y) :- *idol*(X,W) & *buys*(W,Y).
r_3: *buys*(X,Y) :- *perfectFor*(X,Y).

This is a separable recursion. ∎

Example 1.2 Suppose now that someone will no longer buy a product if their idol buys it, but that they will buy a product if it is cheaper than another product they will buy. Assuming that we have a relation *cheaper*(X,Y) of products X and Y such that X is cheaper than Y, we can now define the relation *buys* as follows:

r_1: *buys*(X,Y) :- *friend*(X,W) & *buys*(W,Y).
r_2: *buys*(X,Y) :- *buys*(X,Z) & *cheaper*(Y,Z).
r_3: *buys*(X,Y) :- *perfectFor*(X,Y).

This is also a separable recursion. ∎

Two currently popular evaluation algorithms are Magic Sets [BMSU86,BR87] and Counting [BMSU86, BR87,SZ86]. Section 4 discusses the relative worst case performance of Magic Sets, Counting, and Separable on some general classes of separable recursions. That section also shows that if n is the number of distinct constants in the base relations mentioned in the recursion, on the query *buys(tom,Y)?* on the relation *buys* defined in Example 1.2, the Generalized Magic Sets algorithm generates relations of size is $\Omega(n^2)$. For the same query on the relation *buys* defined in Example 1.1, the Generalized Counting Method generates relations of size $\Omega(2^n)$. The evaluation algorithm presented here is $O(n)$ for both queries.

In related work, Beeri et al. [BKBR87] suggest that an evaluation algorithm uses a selection effectively if it reduces the arity of the recursion, and prove that a selection on a binary chain rule program can be replaced by a monadic recursion if and only if the underlying language for the program is regular. For binary chain programs that are separable, this work provides an algorithm to do so. However, there are separable recursions that are not binary chain programs; for such programs the connection between the two papers is less obvious.

Chang [Cha81] presents an algorithm for evaluating queries on relations defined by a system of regular chain rules. As separable recursions do not include mutually recursive predicates, but do include nonchain rules, the class of programs to which Chang's algorithm applies and the class to which our algorithm applies are incommensurate. Chang's algorithm requires that the base relations be acyclic, and is $\Omega(n^2)$ on the queries in Examples 1.1 and 1.2. Minker and Nicolas [MN82] note that Chang's algorithm can be extended to some nonlinear chain rule recursions.

Henschen and Naqvi [HN84] note that for some special cases of single rule linear recursions (no "induced part" or no "determined part") their algorithm can be simplified. Han and Henschen [HH87] mention a similar algorithm for computing selections on transitive closure queries. The separable recursion evaluation algorithm reduces to this simplification for these special cases, but is more general in that it applies to multiple rule recursions. The general Henschen and Naqvi algorithm [HN84] fails for cyclic data and, like generalized counting, is $\Omega(2^n)$ on the query in Example 1.1.

The one-sided recursion evaluation algorithm in Naughton [Nau87] and the separable algorithm are identical for separable single rule recursions. However, as not all one-sided recursions are separable and not all separable recursions are one-sided, the one-sided evaluation algorithm and the separable evaluation algorithm apply to different classes of programs.

Aho and Ullman [AU79] present a technique of pushing selections into fixpoints that, when combined with semi-naive evaluation, produces an instance of our algorithm if the selection is on a "stable" variable and the recursion is separable. As their algorithm applies to recursions that are not separable but not to all selections on separable recursions, their algorithm also applies to a set of programs and queries incommensurate with that to which our separable recursion evaluation algorithm applies.

As the algorithm presented here applies only to a special class of recursions, it must supplement more general algorithms such as Generalized Magic Sets rather than replace them. However, because of its superior performance on queries on separable recursions, and because it is computationally simple to detect separable recursions, we expect that this evaluation algorithm will be a useful component of a recursive query processor.

2 Separable Recursions

We consider queries on relations defined by function-free pure horn clause programs. We use Prolog syntax, and require that the heads (consequent) of the rules contain no repeated variables and no constants. The predicates are divided into two types: IDB predicates, which appear in the head of some rule, and EDB predicates, which appear in the head of no rule and are defined by their extent.

If the predicate appearing in the rule head appears exactly once in the rule body, then the rule is *linear recursive.* The *definition* of an IDB predicate t is the set of all rules in which t appears in the head. In this paper we consider queries on relations defined by one or more linear recursive rules r_1 through r_n. We assume without loss of generality that each definition also contains a single nonrecursive rule r_e, and that the body of r_e consists of the single predicate instance t_0. Furthermore, we assume that the definitions of the predicates other than t do not depend on t — that is, they are not mutually recursive with t. We will call any predicate other than t a *base* predicate.

A *query* is some predicate instance, possibly containing variables and constants. The evaluation algorithm must return the set of all tuples of values for the variables that make the query true.

The *expansion* of a predicate t is the set of all conjunctions of EDB predicates that can be generated by some sequence of rule applications beginning with applying some rule to t. To "apply" a rule to a conjunction of predicate instances, choose some predicate instance in the conjunction and some rule with a head that unifies with that predicate instance. Then replace the chosen predicate instance with the body

```
1)  Give all variables in rules subscript 0;
2)  S := ∅;
3)  $Fringe$ := {$t$};
4)  while $true$ do
5)      $NewFringe$ := ∅;
6)      for each element $e$ of $Fringe$ do
7)          S := S ∪ {$e$ with $r_e$ applied};
8)          for $i$ := 1 to $n$ do
9)              $NewFringe$ := $NewFringe$∪
10)                 {$e$ with $r_i$ applied};
11)         end;
12)     end;
13)     increment subscripts in all rules;
14)     $Fringe$ := $NewFringe$;
    endwhile;
```

Figure 1: Procedure Expand

of the chosen rule after the most general unifier has been applied. For recursive predicates, the expansion is infinite.

For the recursions considered in this paper, the expansion can be generated by procedure Expand in Figure 1. The input to that procedure is some recursion; the output is the infinite set S, the expansion of that recursion.

The elements of an expansion are conjunctive queries, and will be called *strings.* If a variable V appears in the initial instance of t, then V is a *distinguished* variable; otherwise, it is *nondistinguished.* By our assumption that the heads of rules contain no repeated variables, the unifications in line 7 and 10 of Procedure Expand can always be done by replacing the variables in the heads of r_e or r_i by the corresponding variable in the instance of the recursive predicate in e. As *Fringe* is initialized to the instance of t containing the distinguished variables, this implies that the distinguished variables will always appear without subscripts, while the nondistinguished variables will always appear with subscripts.

If $V_1, \ldots, V_i$ are the distinguished variables, and $W_1, \ldots, W_j$ the nondistinguished variables, then the relation specified by the string $p_1 \ldots p_n$ is

$$\{(V_1, \ldots, V_i) | (\exists W_1, \ldots, W_j)(p_1 \wedge \ldots \wedge p_n)\}$$

The recursively defined relation is the union of the relations for the strings in the expansion.

Example 2.1 The expansion of the definition in Example 1.1 begins

$$p(X, Y),$$

$$f(X, W_0)p(W_0, Y),$$
$$i(X, W_0)p(W_0, Y),$$
$$f(X, W_0)f(W_0, W_1)p(W_1, Y),$$
$$f(X, W_0)i(W_0, W_1)p(W_1, Y),$$
$$i(X, W_0)f(W_0, W_1)p(W_1, Y),$$
$$i(X, W_0)i(W_0, W_1)p(W_1, Y),$$

∎

Definition 2.1 A predicate instance p_1 is *connected* to a predicate instance p_2 if p_1 shares a variable with p_2, or shares a variable with a predicate instance connected to p_2.

Definition 2.2 A subset of predicate instances C is a *maximal connected set* if

1. For every pair of predicate instances p_1 and p_2 in C, p_1 and p_2 are connected, and
2. No predicate instance in C shares a variable with any predicate instance not in C.

Definition 2.3 Let r be a linear recursive rule and let t be the recursive predicate in r. Then r contains *shifting variables* if there is some variable X such that X appears in position p in the instance of t in the head of r and in position q in the instance of t in the body of r, where $p \neq q$.

Definition 2.4 (Separable Recursions) Let t be defined by n recursive rules r_1 through r_n. Furthermore, let t_i^h be the argument positions of t such that in the instance of t at the head of rule r_i, each argument position in t_i^h shares a variable with a nonrecursive predicate in the body of r_i. Similarly, let t_i^b be the argument positions of t such that in the instance of t in the body of rule r_i, each argument position in t_i^b shares a variable with a nonrecursive predicate in the body of r_i. Then the definition of t is a *separable recursion* if

1. For $1 \leq i \leq n$, r_i has no shifting variables, and
2. For $1 \leq i \leq n$, $t_i^h = t_i^b$, and
3. For $1 \leq i \leq n$ and $i < j \leq n$, either $t_i^h = t_j^h$ or t_i^h and t_j^h are disjoint, and
4. For $1 \leq i \leq n$, removing the instance of t from the body of r_i leaves a maximal connected set.

Section 5 discusses what happens when each of these restrictions is removed. Note that Condition 3 of the above definition partitions the recursive rules into equivalence classes, where rules r_i and r_j are in the same equivalence class if $t_i^h = t_j^h$. These equivalence classes can be evaluated essentially independently; this is the motivation for the term "separable recursion." If a separable recursion has n such equivalence classes, we will denote these equivalence classes e_i, for $1 \leq i \leq n$. The columns of t that share variables with nonrecursive predicate instances in equivalence class e_i will be denoted by $t|_{e_i}$.

In general, there may be columns of t that share variables with no equivalence class. These columns are denoted $t|_{pers}$, because the variables in these positions are *persistent*, that is, they always appear in the same position in the instances of t in *Fringe* throughout the expansion.

Example 2.2 In Example 1.1, $n = 2$. The recursive rules contain no shifting variables, so the recursion satisfies Condition 1 of Definition 2.4. Also, $buys_1^h = \{buys^1\}$, $buys_1^b = \{buys^1\}$, while $buys_2^h = \{buys^1\}$, and $buys_2^b = \{buys^1\}$, so that recursion satisfies Condition 2 of Definition 2.4. Condition 3 is satisfied since $\{buys_1^h\} = \{buys_2^h\}$. Finally, removing the instance of *buys* from r_1 and r_2 leaves a maximal connected set of size 1 in each case, so that recursion satisfies Condition 4 of Definition 2.4. This recursion has one equivalence class, e_1, containing rules r_1 and r_2. Here $t|_{e_1}$ is the first column of t, and $t|_{pers}$ is the second.

In Example 1.2, there are again no shifting variables, and we have $buys_1^h = \{buys^1\}$, $buys_1^b = \{buys^1\}$, while $buys_2^h = \{buys^2\}$, and $buys_2^b = \{buys^2\}$, so that recursion satisfies Conditions 1 and 2 of Definition 2.4. Also, $\{buys_1^h\}$ and $\{buys_2^h\}$ are disjoint, so that recursion satisfies Condition 3. Finally, removing the instance of *buys* from r_1 and r_2 again leaves a maximal connected set of size 1 in each case, so that recursion is also separable. This recursion has two equivalence classes, e_1 containing r_1, and e_2 containing r_2. Here $t|_{e_1}$ is the first column of t, while $t|_{e_2}$ is the second and $t|_{pers}$ is empty. ∎

Definition 2.5 A selection on a separable recursion is a *full selection* if either the query predicate contains a constant in $t|_{pers}$, or there is at least one equivalence class e_i such that in the query predicate, all variables in $V_h(t|_{e_i})$ are replaced by constants.

In this paper we consider only full selections. The extension to selections that are not full is straightforward and is discussed in [Nau88b]. (A selection that is not full is one that binds only a subset of the columns of an equivalence class.)

3 Algorithms

3.1 Detection Algorithms

If an algorithm for a class of recursive definitions is to be practical, we must have an efficient way of determining whether or not a given recursion falls into the class. To decide if a recursion is a separable recursion, we have to verify each of the conditions of Definition 2.4. Below we give upper bounds on the time required to do so using straightforward algorithms.

Let r be the number of rules in the recursion, k be the maximum over all predicates occuring in the recursion of the number of arguments of a predicate, and let l be the maximum over all rules of the number of predicates in the rule body.

Condition 1 can be verified in time $O(k^2r)$ by, for each rule, checking each variable in the head to see where it appears (if at all) in the instance of the recursive predicate in the body. Similarly, condition 2 can be verified in time $O(k^2lr)$ by, for each i, first identifying t_i^h ($O(k^2l)$), then identifying t_i^b (also $O(k^2l)$), and finally checking that they are equal ($O(k^2)$.)

Once t_i^h and t_i^b have been identified, for $1 \leq i \leq n$, then condition 3 can be verified in $O(k^2r^2)$ by checking, for each i and for all $j \neq i$, that either $t_i^h = t_j^h$ or t_i^h and t_j^h are disjoint. Finally, condition 4 can be verified in time $O(rk^2l^2)$ by, for each i and each predicate instance p in r_i, checking each argument of p to see if it shares a variable with some other predicate instance p' in r_i.

Clearly, the above can be improved upon. The important thing to notice, however, is that the parameters r, k, and l refer to the rules and predicates in the recursion, not to the database. If we let n be the size of the database, in general we expect n to be much larger than r, l, or k. Hence the time to evaluate the query, which is proportional to n, will dominate the time to verify that the recursion is separable. Put another way, spending time that is a small polynomial in the size of the query to deduce that an algorithm that is $O(n)$ can be used instead of one that is $O(n^2)$ or even $O(2^n)$ will be a "win" in almost all cases.

3.2 The Evaluation Algorithm

We begin with a motivating abstract example. Consider the recursion

$$\begin{array}{l} t(X,Y) \text{ :- } a_1(X,W) \ \&\ t(W,Y). \\ t(X,Y) \text{ :- } a_2(X,W) \ \&\ t(W,Y). \\ t(X,Y) \text{ :- } t(X,W) \ \&\ b_1(W,Y). \\ t(X,Y) \text{ :- } t(X,W) \ \&\ b_2(W,Y). \\ t(X,Y) \text{ :- } t_0(X,Y). \end{array}$$

This is a separable recursion. Ignoring variables, the elements of the expansion can be described by the regular expression $(a_1 + a_2)^* t_0 (b_1 + b_2)^*$.

Suppose that we wish to evaluate the query $t(x_0, Y)$?. Then, after substituting x_0 for X in each string, the strings of the expansion can be evaluated from left to right, using the selection constant to restrict the first lookup and shared variables to restricted subsequent lookups. The answer to the query is the union of the values for Y, which will appear in column 2 of the last predicate of the string.

Every value that could be passed to t_0 in the evaluation of some string of the expansion can be found by the following algorithm:

```
$carry_1 := \{x_0\};$
$seen_1 := carry_1;$
while $carry_1$ not empty do
        $carry_1 := \pi_3(carry_1 \bowtie_{1=1} a_1) \cup$
                        $\pi_3(carry_1 \bowtie_{1=1} a_2);$
        $carry_1 := carry_1 - seen_1;$
        $seen_1 := seen_1 \cup carry_1;$
endwhile;
```

In the above, $carry_1$ and $seen_1$ are unary, relation-valued variables. At the end of the algorithm, $seen_1$ contains all values that will be passed to t_0 in any string of the expansion.

By taking $\pi_3(seen_1 \bowtie_{1=1} t_0)$, we get all values that can be passed from t_0 to an instance of b_1 or b_2 in any string of the expansion. Thus the following algorithm finds all values that appear at the right end of some string (the Y values):

```
$carry_2 := \pi_3(seen_1 \bowtie_{1=1} t_0);$
$seen_2 := carry_2;$
while $carry_2$ not empty do
        $carry_2 := \pi_3(b_1 \bowtie_{2=1} carry_2) \cup$
                        $\pi_3(b_2 \bowtie_{2=1} carry_2);$
        $carry_2 := carry_2 - seen_2;$
        $seen_2 := seen_2 \cup carry_2;$
endwhile;
```

Here $carry_2$ and $seen_2$ are again unary, relation-valued variables. At the end of this algorithm, $seen_2$ will be the answer to the query.

There is no need to record how a tuple got into $seen_1$ or $seen_2$; this independence of the evaluation of the a and b parts of the recursion is crucial to the efficiency of the separable recursion evaluation algorithm. Note also that the above evaluation procedure only looks at tuples along a path from x_0, and it examines each tuple at most once.

```
1)  init $carry_1$;
2)  $seen_1 := carry_1;$
3)  while $carry_1$ not empty do
4)          $carry_1 := f_1(carry_1);$
5)          $carry_1 := carry_1 - seen_1;$
6)          $seen_1 := seen_1 \cup carry_1;$
7)  endwhile;
8)  $carry_2 := g_2(seen_1);$
9)  $seen_2 := carry_2;$
10) while $carry_2$ not empty do
11)         $carry_2 := f_2(carry_2);$
12)         $carry_2 := carry_2 - seen_2;$
13)         $seen_2 := seen_2 \cup carry_2;$
14) endwhile;
15) $ans := seen_2;$
```

Figure 2: A schema for evaluating full selections on separable recursions.

Figure 2 gives a general schema for evaluating a full selection on a separable recursion. The $carry_i$, the $seen_i$, and ans are relation-valued variables. The f_i, g_i, and h are relational operators. In addition to the arguments listed, the f_i, g_i, and h may involve relations and constants appearing in the rules and in the query.

We assume that the rules are "rectified," that is, they have been re-written so that heads of the rules are identical and contain no repeated variables. (The term "rectified" is from Ullman [Ull88].) Furthermore, we assume that the rules are re-written so that if $t_b^i = t_b^j$, then the variables in corresponding positions of t_b^i and t_b^j are identical.

For convenience we will describe how the operators are generated from the rules using Datalog notation rather than relational algebra. For example, instead of writing $p := \pi_{1,3}(p \bowtie_{2=1} q)$ we will write $p(X, Y) := p(X, W)$ & $q(W, Y)$. This is not a Datalog rule; in particular, it is not a fixpoint equation. The right side of the statement is evaluated, and assigned to the left side, once. For convenience we extend the notation to allow unions and differences on the right side. For example, $p(X, Y) := q(X, Y) \cup r(X, Y)$ assigns the union of the relations q and r to p.

Let there be n equivalence classes $e_1, \ldots, e_n$. To describe how the schema in Figure 2 is instantiated, we will use the following notation:

- Let the rules in equivalence class e_k be r_{k1}, r_{k2}, $\ldots$, r_{km_k}. We represent the nonrecursive predicate instances in the body of the rule r_{ki} by a_{ki}.

- For $1 \leq k \leq n$, let $V_h(t|_{e_k})$ be the variables that appear in $t|_{e_k}$ in the head of the rules in e_k, in the order in which they appear in $t|_{e_k}$ in the heads of the rules.
- For $1 \leq k \leq n$, let $V_b(t|_{e_k})$ be the variables that appear in $t|_{e_k}$ in the body of the rules in e_k, in the order in which they appear in $t|_{e_k}$ in the bodies of the rules.
- Let $V(t|_{pers})$ be the variables that appear in $t|_{pers}$.
- Let x_0 be the selection constant, and let X be the variable that appears in the selected-on column in the heads of the rules. (In general x_0 and X will be vectors of constants and variables.)

If the selection constants appear in $t|_{pers}$, then replace lines 1-7 of Figure 2 with the single statement $seen_1(x_0)$ and create a dummy equivalence class e_1, setting $V(t|_{e_1}) = X$, the where X is the vector of variables replaced by the selection constants in the query. Also, delete the positions in which the variables X appeared from $t|_{pers}$, and renumber the other equivalence classes so there is only one e_1.

If the selection constants do not appear in $t|_{pers}$, then without loss of generality assume that the selection constants appear in $t|_{e_1}$. We create a pair of relation-valued variables $carry_1$ and $seen_1$. These relation variables have one column for each variable in $V_h(t|_{e_1})$. The carry initialization statement (line 1 in Figure 2) is the fact

$carry_1(V_h(t|_{e_1}))$;

with the variables in $V_h(t|_{e_1}))$ replaced by the corresponding selection constants. The carry extension operator f_1 (line 4 of Figure 2) is given by

$$carry_1(V_b(t|_{e_1})) := a_{11} \ \& \ carry_1(V_h(t|_{e_1})) \cup \ldots \cup a_{1m_1} \& carry_1(V_h(t|_{e_1}));$$

Next create another pair of relation-valued variables $carry_2$ and $seen_2$. These variables have one column for each variable in the concatenation of $V_h(t|_{e_2}), \ldots, V_h(t|_{e_n}), V(t|_{pers})$. The initialization of $carry_2$ is

$$carry_2(V_h(t|_{e_2}), \ldots, V(t|_{e_n}), V(t|_{pers})) := t_0 \ \& \ seen_1(V_h(t|_{e_1})) \cup t_0;$$

where the final instance of t_0 has the selected-on variables replaced by the corresponding constants. The carry extension operator f_2 (line 11 of Figure 2) is given by

$$carry_2(V_h(t|_{e_2}), \ldots, V_h(t|_{e_n}), V(t|_{pers})) := \bigcup_{i=2}^{n} \bigcup_{j=1}^{m_i} a_{ij} \& carry_2(V_b(t|_{e_2}), \ldots, V_b(t|_{e_n}), V(t|_{pers}));$$

```
$carry_1(tom)$;
$seen_1(W) := carry_1(W)$;
while $carry_1$ not empty do
        $carry_1(W) := carry_1(X) \ \& \ f(X,W) \cup$
                $carry_1(X) \ \& \ i(X,W)$;
        $carry_1(W) := carry_1(W) - seen_1(W)$;
        $seen_1(W) := seen_1(W) \cup carry_1(W)$;
endwhile;
$carry_2(Y) := seen_1(X) \ \& \ p(X,Y) \cup p(tom,Y)$;
$seen_2(Y) := carry_2(Y)$;
$ans(Y) := seen_2(Y)$;
```

Figure 3: The instantiated separable recursion algorithm for Example 1.1.

Finally, the answer operator h is given by

$$ans := seen_2(V_h(t|_{e_2}), \ldots, V_h(t|_{e_n}), V(t|_{pers}));$$

where ans contains all the variables from the head of the rule except $V_h(t|_{e_1}))$.

Example 3.1 On the query $buys(tom, ?)$ on the recursion of Example 1.1, the evaluation algorithm produced by instantiating the schema is shown in Figure 3.

For the same query on the recursion of Example 1.2, instantiating the schema produces the algorithm of Figure 4. ∎

Theorem 3.1 *The separable recursion evaluation algorithm produced by instantiating the schema in Figure 2 terminates and correctly evaluates any selection of the form "column = constant" on any separable recursion.*

Proof: The proof proceeds by demonstrating the equivalence between the relation produced by the instantiated separable recursion algorithm and the relation that is the union of the relations for the strings in the expansion of the recursion. The details can be found in [Nau88b]. ∎

4 Comparison

There are a number of measures one can use to compare algorithms. An important measure is the "focus" of an algorithm, that is, how well it uses selection constants to limit the portion of the database searched. Here we ignore this measure, as the three

```
$carry_1(tom)$;
$seen_1(W) := carry_1(W)$;
while $carry_1$ not empty do
    $carry_1(W) := carry_1(X)$ & $f(X,W)$;
    $carry_1(W) := carry_1(W) - seen_1(W)$;
    $seen_1(W) := seen_1(W) \cup carry_1(W)$;
endwhile;
$carry_2(Y) := seen_1(X)$ & $p(X,Y) \cup p(tom,Y)$;
$seen_2(Z) := carry_2(Z)$;
while $carry_2$ not empty do
    $carry_2(Y) := carry_2(Z)$ & $c(Y,Z)$;
    $carry_2(Y) := carry_2(Y) - seen_2(Y)$;
    $seen_2(Y) := seen_2(Y) \cup carry_2(Y)$;
endwhile;
$ans(Y) := seen_2(Y)$;
```

Figure 4: The instantiated separable recursion algorithm for Example 1.2.

algorithms are equivalent in that respect. Instead, we focus on the size of the relations generated by each algorithm in the course of answering a query. In [Nau88a] we give empirical average case performance figures for the evaluation algorithms on some representative recursions; here we consider the worst-case figures for classes of separable recursions.

Of course, we must be more precise about what we mean by "relations generated by an algorithm." The difficulty is that one can put an arbitrary project-join query into the body of a recursive rule. Clearly, we cannot put a bound on the size of relations constructed in solving an arbitrary project-join query. Because of this, we focus on the relations that the algorithms would produce even if the bodies of the recursive rule consisted only of a single, nonrecursive predicate, and the recursive predicate. For example, Separable generates $carry_1$, $carry_2$, $seen_1$, $seen_2$, and *ans*. Generalized Magic Sets constructs *magic* and t, the original recursive predicate. Generalized Counting produces *count* and a modified version of t.

Definition 4.1 Let S_p^k be the class of all separable recursions with p recursive rules and a recursive predicate of arity k.

Definition 4.2 Let R be a recursion, let Q be a query on the recursively defined relation of R, let n be the number of distinct constants in the base relations of R, and let M be a query evaluation method. Then we say that M is $O(f(n))$ on Q if, in evaluating Q, M constructs only relations of size $O(f(n))$.

Definition 4.3 Let R be a separable recursion, and let e_1 be an equivalence class of R. Then the *width of* e_1, written $w(e_1)$, is the number of columns in $t|_{e_1}$.

Lemma 4.1 *For all $k > 0$ and $p > 0$, for any recursion R in S_p^k, on any query Q on the recursive relation of R, if e_1 is the equivalence class such that the selection constants of Q appear in $t|_{e_1}$, then Separable is $O(n^{\max(w(e_1),k-1)})$ on Q.*

Lemma 4.2 *For all $k > 0$ and $p > 0$, there is a recursion R in S_p^k, and a query Q on R, such that Generalized Magic Sets is $O(n^k)$ on Q.*

Note that as $w(e_1) \leq k$, $\max(w(e_1), k-1) \leq k$, so by this measure, Separable is never worse than Generalized Magic Sets.

Lemma 4.3 *For all $k > 0$ and $p > 0$, there is a recursion R in S_p^k, and a query Q on R, such that Generalized Counting is $O(p^n)$ on Q.*

The proofs for these lemmas appear in [Nau88b]. Here we illustrate the lemmas with queries on the recursions in Examples 1.2 and 1.1.

On the recursion of Example 1.2, and the query $buys(tom, Y)$? the Generalized Magic Sets algorithm [BR87] will generate the rules

$magic(tom)$.
$magic(W)$:- $magic(X)$ & $friend(X, W)$.

$buys(X,Y)$:- $magic(X)$ & $perfectFor(X,Y)$.
$buys(X,Y)$:- $magic(X)$ & $friend(X, W)$ & $buys(W,Y)$.
$buys(X,Y)$:- $magic(X)$ & $buys(X, Z)$ & $cheaper(Z,Y)$.

Let the relation *friend* contain the tuples $(a_1 = tom, a_2), (a_2, a_3), \ldots, (a_{n-1}, a_n)$. Also, let *cheaper* consist of the tuples $(b_n, b_{n-1}), \ldots, (b_2, b_1)$ and *perfectFor* consist of the tuple (a_n, b_n). At the end of the evaluation of these rules, *buys* will contain the n^2 tuples (a_i, b_j), for $1 \leq i,j \leq n$. Thus Magic Sets generates relations that are $\Omega(n^2)$.

On the recursion of Example 1.1 and the query $t(tom, Y)$?, the Generalized Counting Method [BR87, SZ86] constructs (among others) the rules

$count(1, 1, 1, tom)$.
$count(i+1, 2j, 2k, W)$:- $count(i, j, k, X)$ & $friend(X, W)$.
$count(i+1, 2j+1, 2k, W)$:- $count(i, j, k, X)$ & $idol(X, W)$.

Consider the sample database in which both the relation *friend* and the relation *idol* contain the tuples $(a_1, a_2), (a_2, a_3), \ldots, (a_{n-1}, a_n)$. At the end of evaluating these rules the *count* relation contains the tuples $(i, j, 2^{i-1}, a_i)$ for $1 \leq i < n$, $2^{i-1} \leq j < 2^i$. Thus Generalized Counting is $\Omega(2^n)$. This implies that a 30 tuple database can generate a several gigabyte relation.

On both of the preceding queries, the separable recursion algorithm generates only monadic relations. If the *friend*, *idol*, and *cheaper* relations contain n distinct constants, then no more than $O(n)$ tuples can appear in these monadic relations. Hence the separable algorithm is $O(n)$ on both of the queries above.

5 Conclusion

We have demonstrated that for some separable recursions, the separable recursion evaluation algorithm is significantly more efficient than Magic Sets and Generalized Counting. However, as the separable algorithm is limited in applicability, it must be used to supplement these more general algorithms rather than to replace them.

In view of the performance comparison in Section 4, it is interesting to attempt to apply the separable evaluation algorithm to more general recursions. Perhaps the most promising direction is to attempt to generalize the separable evaluation algorithm to non-linear recursions and to recursions that contain mutually recursive predicates. We are currently investigating this possibility.

References

[AU79] Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Proceedings of the Sixth ACM Symposium on Principles of Programming Languages*, pages 110–120, 1979.

[BKBR87] Catriel Beeri, Paris Kanellakis, Francois Bancilhon, and Raghu Ramakrishnan. Bounds on the propagation of selection into logic programs. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 214–226, 1987.

[BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 1–15, 1986.

[BR87] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 269–283, 1987.

[Cha81] C. Chang. On the evaluation of queries containing derived relations in a relational data base. In H. Gallaire, J. Minker, and J. Nicolas, editors, *Advances in Data Base Theory, Volume 1*, Plenum Press, 1981.

[HH87] Jiawei Han and Lawrence J. Henschen. Handling redundancy in the processing of recursive database queries. In *Proceedings of the ACM-SIGMOD Conference on the Management of Data*, pages 73–81, 1987.

[HN84] Lawrence J. Henschen and Shamim A. Naqvi. On compiling queries in recursive first order databases. *JACM*, 31(1):47–85, 1984.

[MN82] Jack Minker and Jean M. Nicolas. On recursive axioms in relational databases. *Information Systems*, 8(1):1–13, 1982.

[Nau87] Jeffrey F. Naughton. One sided recursions. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 340–348, 1987.

[Nau88a] Jeffrey F. Naughton. *Benchmarking Multi-Rule Recursion Evaluation Strategies.* Technical Report CS-TR-141-88, Princeton University, 1988.

[Nau88b] Jeffrey F. Naughton. *Compiling Separable Recursions.* Technical Report CS-TR-140-88, Princeton University, 1988.

[SZ86] Domenico Sacca and Carlo Zaniolo. The generalized counting methods for recursive logic queries. In *Proceedings of the First International Conference on Database Theory*, 1986.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems.* Volume 1, Computer Science Press, 1988.