Linearizing some recursive logic programs*

Irène Guessarian†and Jean-Eric Pin‡

E-mail: guessarian@litp.ibp.fr,
JeanEric.Pin@frcl.bull.fr

Abstract

We give in this paper a sufficient condition under which the least fixpoint of the equation $X = a + f(X)X$ equals the least fixpoint of the equation $X = a + f(a)X$. We then apply that condition to recursive logic programs containing chain rules: we translate it into a sufficient condition under which a recursive logic program containing $n \geq 2$ recursive calls in the bodies of the rules is equivalent to a linear program containing at most one recursive call in the bodies of the rules. We conclude with a discussion comparing our condition with the other approaches to linearization studied in the literature.

1 Introduction.

We apply fixpoint techniques together with language theory tools to derive simple algorithms for answering some queries on recursive logic programs. We give sufficient conditions on the query and the logic program which enable us to find an iterative program, i.e. a program containing only right-linear recursions, computing exactly the relevant facts needed to answer the query. The method consists of first characterizing the semantics of the logic program using fixpoint theory tools, via algebraic or denotational methods. We compute syntactically the least fixpoint of the logic program in a Herbrand model, then interpret this least fixpoint in the actual domains. The syntactic expression of the least fixpoint can be expressed in language theory terms, as a language $L(P)$, depending on the syntax of P. Then, using language theory tools, we give sufficient conditions on P which ensure that $L(P)$ will be a rational (or regular) language. Hence, we can find an equivalent iterative (or right linear) program P' such that $L(P) = L(P')$, which will thus give the same answers to queries as P. This program P' provides us with an efficient and easy algorithm to answer queries on P. The present method applies to a popular class of programs called chain Horn clauses; it can also be extended to programs allowing for the use of aggregate functions provided that they are stratified and that the evaluation algorithms preserve stratification. Linearization of recursive logic programs has been extensively studied in recent deductive database research; we survey in the

*Support from the PRC Mathématiques-Informatique is gratefully acknowledged.

†LITP/IBP, Université Paris VI et CNRS, Tour 55-65, 4 Place Jussieu, 75252 Paris Cedex 05, France

‡BULL, Recherche et Développement, Rue Jean Jaurès, 78340 Les Clayes-sous-Bois, France

last section the papers dealing with similar topics, and compare them with our result.

A Datalog program P is a finite set of function-free Horn clauses, called *rules*, of the form:

$$Q(X_1, \dots, X_n) \longleftarrow Q_1(Y_{1,1}, \dots, Y_{1,n_1}), \dots, Q_p(Y_{p,1}, \dots, Y_{p,n_p})$$

where $X_1, \dots, X_n$ are variables, the $Y_{i,j}$'s are either variables or constants, Q is an intensional predicate, the Q_i's are either intensional database predicates (i.e. relations defined by logical rules) or extensional database predicates (i.e. relations explicitly stored in the database). We will in the sequel use the usual abbreviations IDB (resp. EDB) standing for intensional data base (resp. extensional data base). The set of all extensional database predicates is denoted by D. A *database* d over D maps every extensional database predicate onto a relation. In practice, we use the same notation for the EDB predicate Q and its associated relation (which should be denoted $d(Q)$).

A *chain-rule* is a rule of the form:

$$Q(X_1, X_2) \longleftarrow Q_1(X_1, Y_1), Q_2(Y_1, Y_2), \dots, Q_p(Y_{p-1}, X_2)$$

A *chain program* is a Datalog program containing only chain rules. Two EDB predicates Q_i and Q_j commute (for a given database), if $Q_i \circ Q_j = Q_j \circ Q_i$, i.e.,

$$\begin{aligned} Q_i \circ Q_j &= \{(X, Y) \mid \exists Z\ Q_i(X, Z) \wedge Q_j(Z, Y)\} \\ &= \{(X, Y) \mid \exists Z\ Q_j(X, Z) \wedge Q_i(Z, Y)\} = Q_j \circ Q_i \end{aligned}$$

Example 1.1 Examples of commuting relations are the following:

(1) A database where all EDB predicates are of the form A^n, for a given relation A, since for all i, j, $A^i \circ A^j = A^j \circ A^i$.

(2) A database on the domain $\mathbb{N}$ of the integers, together with

$$\begin{aligned} A_1 &= \{(n, p) \mid \text{there exists } k\ p = kn\} = \{(n, p) \mid\ n \text{ divides } p\} \\ A_2 &= \{(n, p) \mid \text{there exists } k\ p = k2^n\} = \{(n, p) \mid\ 2^n \text{ divides } p\} \end{aligned}$$

Then:

$$A_1 \circ A_2 = A_2 \circ A_1 = A_2$$

A standard way of answering queries concerning Q would be, for instance by considering Q as a recursive procedure, to compute the relation Q and the course of the computation will involve several recursive calls. This process can be improved in the present case; using language theory tools we show that a chain program P on a database on which the EDB predicate commute is equivalent to a Datalog program P' containing only right linear recursions, i.e. a program all of whose rules are of one of the following forms:

$$\begin{aligned} Q(X_1, X_2) &\longleftarrow A_1(Y_1, Y_1'), A_2(Y_2, Y_2'), \dots, A_{p-1}(Y_{p-1}, Y_{p-1}'), Q(Y_{p-1}', X_2) \\ Q(X_1, X_2) &\longleftarrow A_1(Y_1, Y_1'), A_2(Y_2, Y_2'), \dots, A_{p-1}(Y_{p-1}, Y_{p-1}'), A_p(Y_{p-1}', X_2) \end{aligned}$$

where all the A_i's are EDB predicates.

To this end we first give a new sufficient condition for a certain type of context-free language to be *rational* (or *regular*). We then translate it into a sufficient condition under which a recursive logic program containing $n \geq 2$ recursive calls in the bodies of the rules is equivalent to a right linear program. The main difference between our approach and the previously studied cases of linearizations consists in the following facts:

(a) We linearize only chain rule programs.

(b) We can linearize directly programs containing an arbitrary number of recursive calls, versus the usually considered bilinear programs.

(c) Our linearization condition can be easily checked. See also the discussion at the end of the paper.

The paper is organized as follows: in the next section, we prove the language theory result, in Section 3 we apply this result to chain rule programs, and in the last section we conclude with a discussion of related literature.

2 Rational functions and least fixpoints

In this section, we introduce the language theoretic tools needed in this paper. The reader interested only in databases may omit the proofs and concentrate on the examples. Our purpose is to give an abstract model of the following situation. Let U be a set and let E be the set of relations on U. Then E is naturally equipped with the inclusion ordering: given two relations R and S on U, $R \subset S$ if and only if, for every $(x, y) \in U \times U$, $R(x, y)$ implies $S(x, y)$. Now, the least upper bound is simply union and the largest lower bound is intersection. Another natural operation on E is the composition of relations (denoted by $\circ$, or to simplify notations, by just concatenation). Observe that, for all relations R, S, T on U,

$$(R \cup S) \circ T = (R \circ T) \cup (S \circ T) \quad \text{and} \quad R \circ (S \cup T) = (R \circ S) \cup (R \circ T)$$

Denote by R^* the reflexive transitive closure of R.

$$R^* = I \cup R \cup R \circ R \cup R \circ R \circ R \cup \ldots$$

We now give our formal model. Let E be a complete lattice together with a composition operation denoted multiplicatively. The ordering will be denoted $\leq$, the least upper bound $+$, the bottom of the lattice by 0. We assume that E, equipped with this addition and multiplication, is a *complete semiring*, that is, it satisfies the following properties:

(1) $(ab)c = a(bc)$ for every a, b, $c \in E$,

(2) there exists an element $1 \in E$ such that, for every $a \in E$, $a1 = 1a = a$,

(3) $0a = a0 = 0$ for every $a \in E$,

(4) $(\sum_{i \in I} a_i)b = \sum_{i \in I} a_i b$ for any family $(a_i)_{i \in I}$.

It follows in particular that, for every $a, b, c \in E$, $a \leq b$ implies $ac \leq bc$ and $ca \leq cb$. Note that 1 is not, in general, the top of the lattice E.

A *complete subsemiring* of E is a subset F of E containing 0 and 1 and closed under product and arbitrary sums (that is, if $a, b \in F$ then $ab \in F$, and for any family $(a_i)_{i \in I}$ of elements of F, $\sum_{i \in I} a_i \in F$). If $a \in E$, we set $a^0 = 1$, $a^{n+1} = a^n a$ for every $n \geq 0$, $a^+ = \sum_{n>0} a^n$ and $a^* = \sum_{n \geq 0} a^n$. A semiring is said to be commutative if in addition:

(5) $ab = ba$ for every a, $b \in E$.

We recall the following standard result (Arden rule, see [12]):

Lemma 2.1 *Let $a, b \in E$. Then the least solution of the equation $X = a + bX$ is b^*a.*

It is instructive to translate this lemma in the terminology of relations. Let A and B be two relations on U. Then the least relation satisfying $X = A \cup B \circ X$ is $B^* \circ A$.

The next proposition extends Lemma 2.1 to a more general situation. Under certain assumptions, it gives the least fixpoint of an equation of the form $X = a + f(X)X$, where f is a monotone function.

Proposition 2.2 *Let $f : E \to E$ be a monotone function, let $a \in E$, and suppose that*

(i) $af(a) = f(a)a$

(ii) $g(f(a)^*a) = f(a)^*g(a)$, *where* $g(X) = f(X)X$

Then the least fixpoint of the equation $X = a + f(X)X$ is $af(a)^*$.

Proof. Let X_0 be a solution of the equation $X = a + f(X)X$. Then X_0 is also a solution of

$$X = a + f(X_0)X \tag{1}$$

whence $X_0 \geq f(X_0)^*a$ which is the least solution of (1). Moreover, $X_0 = a + f(X_0)X_0$ implies $X_0 \geq a$, and since f is monotone, $f(X_0) \geq f(a)$, whence $f(X_0)^* \geq f(a)^*$, and $X_0 \geq f(X_0)^*a \geq f(a)^*a = af(a)^*$ by (i).

Let us check that $af(a)^*$ is a solution of the equation $X = a + f(X)X$:

$$\begin{aligned} a + g(f(a)^*a) &= a + f(a)^*g(a) \quad \text{(by (ii))} \\ &= a + f(a)^*f(a)a = a + f(a)^+a = f(a)^*a. \quad \square \end{aligned}$$

Proposition 2.2 is powerful, but conditions (i) and (ii) may be difficult to verify in practice. A more usable form is given below (Propositions 2.3 and 2.4), but requires some auxiliary definitions.

The set $R(E)$ of *rational* functions from E into E is the smallest set R of functions such that

(1) R contains the identity function and the constant functions,

(2) if $h_1 \in R$ and $h_2 \in R$ then $h_1 + h_2 \in R$ and $h_1h_2 \in R$,

(3) if $h \in R$, then $h^* \in R$, where $h^*(x) = h(x)^*$

The set of *polynomial* functions is the smallest set of functions R satisfying conditions (1) and (2) above.

Rational functions are usually represented by *rational expressions.* Rational expressions on E are recursively defined as follows:

(1) for every variable x over E, x is a rational expression, and for every $e \in E$, e is a rational expression,

(2) if h_1 and h_2 are rational expressions, then $(h_1 + h_2)$ and h_1h_2 are rational expressions,

(3) if h is a rational expression, then h^* is a rational expression.

For instance, if e_1, e_2, e_3 are elements of E, then $(e_1xe_1e_2x)^*e_2+((e_1xe_3)^*+e_2)^*$ is a rational expression. To pass from rational expressions to rational functions, it suffices to interpret x as the identity function and, for every $e \in E$, e as the constant function that maps every element of E onto e.

For instance, if e_1, e_2, e_3, e_4 are elements of E, then the rational expression $xe_3xe_2e_4 + e_1xe_1e_2xxe_1e_2xe_3$ denotes the function h defined by

$$h(x) = xe_3xe_2e_4 + e_1xe_1e_2xxe_1e_2xe_3$$

which is a polynomial function.

Note that different rational expressions may represent the same function. For instance, if e_1, e_2 and e_3 are elements of E such that $e_3 = e_1e_2$, then e_1e_2 and e_3 represent the same constant function, that maps every element of E onto $e_1e_2 = e_3$.

The elements of E occurring in a rational expression are the *coefficients* of the expression. For instance, the coefficients of the expression $(e_1xe_1e_2x)^*e_2 + ((e_1xe_3)^* + e_2)^*$ are e_1, e_2 and e_3. We are now ready to state our particular instances of Proposition 2.2, the proofs of which are deferred to the appendix.

Proposition 2.3 *Let $f : E \longrightarrow E$ be a monotone function and let $a \in E$. Suppose there exists a commutative complete subsemiring F of E, containing a, such that f induces a rational function from F into F. Then the least fixed point of the equation $X = a + f(X)X$ is $af(a)^*$.*

Remark 2.1 Let E be the set of relations on U. Proposition 2.3 applies in particular if f is a polynomial (or more generally rational) function given by a rational expression all of which coefficients commute and commute with a (but they don't need to commute with *every* element of E). Indeed, one can take for F the complete subsemiring generated by the coefficients and by a. Thus, in this case, the least fixpoint of $X = a + f(X)X$ is $af(a)^*$. The equation $X = a + f(X)X$ is thus equivalent to the right-linear iteration $X = a + f(a)X$.

More generally, we have the following result:

Proposition 2.4 *Let*

$$P: \quad \begin{cases} X_1 & = & g_1(X_1, \ldots, X_n) \\ & & \ldots \\ X_n & = & g_n(X_1, \ldots, X_n) \end{cases}$$

be a system of recursive equations where each equation $X_i = g_i(X_1, \ldots, X_n)$ can be written in the form:

$$X_i = a_i(X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n) + f_i(X_1, \ldots, X_n)X_i,$$

where a_i and f_i are rational functions given by rational expressions all of whose coefficients commute. Then the least fixpoint of P is a rational function which can be computed by a right linear iteration.

3 Applications to recursive logic programs

We now apply the results of the previous section to Datalog programs.

3.1 Preliminary results

We recall first the fundamentals of the semantics of Datalog programs. Let P be a Datalog program,

$$P: \quad \begin{cases} Q_1(X,Y) & \leftarrow \quad Q_1^1(X,X_1),\dots,Q_{n_1}^1(X_{n_1-1},Y) \\ & \dots \\ Q_p(X,Y) & \leftarrow \quad Q_1^p(X,X_1),\dots,Q_{n_p}^p(X_{n_p-1},Y) \end{cases}$$

Formally, a *database* is a map that associates a ground atomic formula (or fact) for each EDB predicate. It is generally identified with a finite set D of ground atomic formulas. For each database D, let $T_P(D)$ be the set of immediate consequences of D using P, i.e., the set of facts which can be deduced from D using one rule in P; let $T_P^0(D)=\emptyset$, and, for $i\geq 1$, let $T_P^i(D)=T_P(T_P^{i-1}(D))$ be the i-th power of $T_P(D)$. Define

$$T_P^\infty(D)=\bigcup_{i\geq 0} T_P^i(D)$$

and let $Q_P(D)$ be the consequences about predicate Q in $T_P(D)$.

Similarly, let $Q_P^\infty(D)$ (resp. $Q_P^i(D)$) be the set of facts about predicate Q which can be deduced from D by (resp. at most i) applications of the rules in P. Recall first [5] that the IDB predicates defined by the Datalog program P can be equivalently defined by an infinite union of conjunctive queries $(C_{QP,j})_{j\geq 0}$, that is, for every database D, $Q_P^\infty(D)=\bigcup_{j\geq 0} C_{QP,j}(D)$. The $C_{QP,j}$'s are called the Q-expansions of P.

Assuming a Herbrand domain containing a denumerable set of constants, the $C_{QP,j}$'s can be computed as the least fixpoint of a variant T' of the operator of immediate consequences defined by

$$I=(Q_1,\dots,Q_p)\longleftarrow I'=T_P((Q_1,\dots,Q_p)\cup(R_1,\dots,R_n))$$

where $(Q_1,\dots,Q_p)$ (resp. $(R_1,\dots,R_n)$) is a finite set of relations defining the IDB predicates (resp. the EDB predicates). Then, letting for every $Q\in IDB$,

$$\begin{aligned} C_{QP}^0(X,Y) &= \emptyset \\ C_{QP}^1(X,Y) &= \{\ C \mid Q(X,Y)\longleftarrow C \text{ is a non recursive rule of } P \text{ and} \\ &\qquad C\subseteq T_P^0(D)\} \\ &\ \ \vdots \\ C_{QP}^n(X,Y) &= \{\ C(Q_i\longleftarrow C_{Q_iP}^{n-1}) \mid Q(X,Y)\longleftarrow C \text{ is a (possibly recursive) rule} \\ &\qquad \text{of } P \text{ and } C\subseteq T_P^{n-1}(D)\} \end{aligned}$$

where $C(Q_i\longleftarrow C_{Q_iP}^{n-1})$ denotes the result of substituting $C_{Q_iP}^{n-1}$ for Q_i, $i=1,\dots,p$ in C, we have

$$\bigcup_{j\geq 0} C_{QP,j}(D)=\bigcup_{n\geq 0} C_{QP}^n=Q_P^\infty(D).$$

Then

$$T_P^\infty(D)=\bigcup_{Q\in IDB}\cup_{i\geq 0}C_{QP,i}(D)=\bigcup_{Q\in IDB}\cup_{n\geq 0}C_{QP}^n$$

For P a Datalog program, we will consider its semantics as being defined by the least fixpoint μP of T'. Thus $\mu P = \bigcup_{Q \in IDB} \cup_{n \geq 0} C^n_{QP}$. It can be shown that μP can be defined also by $\mu P : D \rightarrow \sup\{T^n_P(D) \mid n \in \mathbb{N}\}$ in the Herbrand universe, where T_P is the immediate consequence operator. For an IDB predicate Q of P, the semantics of Q are the set of facts about Q in μP, i.e. $\cup_{n \geq 0} C^n_{QP}$.

Definition 3.1 *Two Datalog programs P and P' are said to be equivalent if $\mu P = \mu P'$, and this is equivalent to saying that for all D, $T^\infty_P(D) = T^\infty_{P'}(D)$.*

Assuming Q is binary, and letting $Q(?,?)$ (resp. $Q(X_0, ?)$) be a query, the answers to the query are all the substitutions $\nu : \{x, y\} \longrightarrow D$ (resp. $\nu : \{y\} \longrightarrow D$) such that $Q(\nu(x), \nu(y))$ (resp. $Q(X_0, \nu(y))$ is in μP.

For more details about the semantics of logic programs, and query answering methods, the reader can refer to [2, 11].

A Datalog rule:

$$Q(X_1, \ldots, X_n) \longleftarrow Q_1(Y_{1,1}, \ldots, Y_{1,n_1}), \ldots, Q_p(Y_{p,1}, \ldots, Y_{p,n_p}) \qquad (2)$$

is said to be *linear*[1] (resp. *right linear, left linear*) if and only if at most one of the Q_i's (resp. at most Q_p, Q_1) is an IDB predicate, all other Q_j's in the body of (2) being EDB predicates.

A Datalog program is said to be *linearizable* (resp. *right linearizable, left linearizable*) if it is equivalent to a Datalog program consisting only of linear (resp. right linear, left linear) rules.

Recall finally that the trace language of P on a database D is obtained by erasing variables from the predicate conjunctions of μP, and considering only the predicate symbols. More precisely, the *trace language of P relative to Q* is generated by the context free grammar L_P deduced from P by

(a) considering as terminal (resp. non terminal) symbols the EDB (resp. IDB) predicate symbols of P, together with Q as axiom (or start symbol),
(b) erasing variables,
(c) reversing arrows.

Example 3.1 For instance if P is given by

$$P: \quad \begin{cases} Q(X,Y) & \leftarrow & A(X,Y) \\ Q(X,Y) & \leftarrow & B(X,Z), Q(Z,Z'), C(Z',Y) \end{cases}$$

then L_P is given by

$$Q \longrightarrow A + BQC$$

and the trace language of P relative to Q is $\{B^n A C^n \mid n \geq 0\}$.

If moreover A, B and C commute, then the trace on D can be represented by the regular language $A(BC)^*$.

[1] Our definition is slightly more restrictive than if we would have defined linearity with respect to recursive predicates only. A rule with multiple IDB predicates can still be considered as linear if there is at most one of the Q_i's which is an IDB predicate mutually recursive with the head of the rule Q, the other Q_i's being either non recursive IDB's or EDB's. However, since non recursive predicates can be easily deleted from the program, we can, without loss of generality, restrict ourselves to our definition which simplifies the formulation.

Lemma 3.1 *Let P and P' be chain programs involving the same IDB predicates. Then P and P' are equivalent if and only if their traces relative to all IDB predicates coincide on all databases.*

Proof. This is due to the fact that any Q-expansion of $Q(X,Y)$ in a chain program is a chain, and such a chain is characterized by the sequence of predicates symbols occurring in it. In other words, variables can be omitted without ambiguity when writing such a chain [19], which is fully represented by the associated term in the trace. □

Lemma 3.1 is false if the program is not a chain program. For instance the Datalog programs $P: \ Q(X,Y) \leftarrow A(X), B(Y)$ and $P': \ Q(X,Y) \leftarrow A(Y), B(X)$ have the same traces but are not equivalent.

We now consider chain programs of the form

$$P: \quad \begin{cases} Q_1(X,Y) & \leftarrow & Q_1^1(X,X_1),\ldots,Q_{n_1}^1(X_{n_1-1},Y) \\ & \ldots & \\ Q_p(X,Y) & \leftarrow & Q_1^p(X,X_1),\ldots,Q_{n_p}^p(X_{n_p-1},Y) \end{cases}$$

in which all EDB predicates commute on a database D. We first show that this condition also implies that IDB predicates commute.

Lemma 3.2 *If all EDB predicates commute on a database D, then all the IDB predicates commute, and they also commute with the EDB predicates.*

Proof. By induction on the fixpoint computation of μP, we can see that each IDB predicate can be expressed as a union of conjunctive queries containing only EDB predicates. Since all EDB predicates commute, the IDB predicates will also commute with the EDB predicates and among themselves. □

We can now state

Proposition 3.3 *Let*

$$P: \quad \begin{cases} Q_1(X,Y) & \leftarrow & Q_1^1(X,X_1),\ldots,Q_{n_1}^1(X_{n_1-1},Y) \\ & \ldots & \\ Q_p(X,Y) & \leftarrow & Q_1^p(X,X_1),\ldots,Q_{n_p}^p(X_{n_p-1},Y) \end{cases}$$

be a chain program in which all EDB predicates commute on a Database D. Then:

(1) *P is right linearizable,*

(2) *the traces of P on D are regular languages.*

Proof. By Lemma 3.1, chain programs are equivalent if and only if their traces relative to all EDB coincide and thus (2) is a consequence of (1).

To prove (1), notice first that by Lemma 3.2, the IDB predicates also commute (with both EDB and other IDB predicates). Each rule of P can then be written in one of the two forms: $Q_i \leftarrow Q_1^i \circ \ldots \circ Q_{n_i}^i$ (resp. $Q_i \leftarrow Q_1^i \circ \ldots \circ Q_{n_i}^i \circ Q_i$) where the Q_k^i's are EDB and IDB predicates different from Q_i (resp. arbitrary EDB and IDB predicates).

Grouping together all clauses with head Q_i, we obtain a single equation defining Q_i and having the following form:

$$Q_i \leftarrow a_i(Q_1, \ldots, Q_{i-1}, Q_{i+1}, \ldots, Q_p) + f_i(Q_1, \ldots, Q_p) \circ Q_i$$

where $+$ stands for the set theoretic union or the disjunction $\vee$), the a_i's (resp. f_i's) are polynomials of the EDB predicates and the Q_j's ($j \neq i$) (resp. of the EDB predicates and the Q_j's), and all predicates of a_i (resp. f_i) commute.[2]

Then, by Proposition 2.4, P is equivalent to a program P' of the form:

$$P' : Q_i = a'_i + f'_i \circ Q_i \tag{3}$$

where a'_i and f'_i are polynomials of the EDB predicates only. □

The conditions on P which we consider for the ease of its application, are stronger than really needed and are not necessary (see Example 3.3). It would suffice to demand that the conditions (i) and (ii) of Proposition 2.2 be satisfied. But these conditions would be more difficult to check than commutativity. Commutativity then trivially implies that the conditions (i) and (ii) are satisfied. Commutativity can be checked in each case like in Example 1.1. On the other hand, one can give *sufficient* conditions for the commutativity of EDB predicates that are very easy to check. For instance, predicates modelling functions which modify disjoint sets of attributes will commute (see Example 3.5). See also Examples 3.2 and 3.3.

Example 3.2 Let P be the recursive logic program

$$P : \quad \begin{cases} Anc(x,y) & \leftarrow & Par(x,y) \\ Anc(x,y) & \leftarrow & Anc(x,z), Anc(z,y). \end{cases} \tag{4}$$

P is the archetype of recursive logic programs and it corresponds to the computation of the transitive closure of a relation. P is associated with the single equation:

$$Anc = Par + Anc \circ Anc,$$

i.e. $X = Par + X \circ X$. In the present case, there is a single EDB predicate Par, and all conditions are trivially verified. Commutativity is true since Par commutes with itself, and conditions (i) and (ii) of Proposition 2.2 are also trivial. With the notations of Proposition 2.2, E is the set of relations on a set U, the product is the composition of relations, $a = Par$, and $f(X) = X$, hence conditions (i) and (ii) reduce to (i) $Par \circ Par = Par \circ Par$, and (ii) $(Par^* \circ Par) \circ Par = Par^* \circ (Par \circ Par)$.

By Propositions 2.2 and 3.3, the leastfixpoint μP of (4) is

$$Par \circ f(Par)^* = Par \circ Par^*$$

whence $Anc = Par \circ Par^* = Par^+ = Par + Par^2 + Par^3 + \ldots$. Thus Anc coincides with the least fixpoint of

$$P' : X = Par + Par \circ X$$

[2]Note that Q_i may occur several times in $f_i(Q_1, \ldots, Q_p)$. The above equation is thus multilinear, as opposed to the usual techniques, which linearize only bilinear equations, where Q_i may occur only twice in the body of the equation, and hence only once in $f_i(Q_1, \ldots, Q_p)$.

which is a right linear iterative program for answering queries of the form $Anc(?,?)$.

Example 3.3 Consider the program:

$$P: \begin{cases} Q(X,Y) & \longleftarrow & A_1(X,Y_1), Q(Y_1,Y_2), A_2(Y_2,Y_3), Q(Y_3,Y) \\ Q(X,Y) & \longleftarrow & B(X,Y) \end{cases} \tag{5}$$

that we rewrite, according to our notations, into:

$$Q = A_1 \circ Q \circ A_2 \circ Q \;+\; B$$

In the present case, there are three EDB predicates B, A_1 and A_2, commutativity of B, A_1 and A_2 could still be reasonably easily satisfied (even though it is *not* necessary, cf. Example 4.1). Conditions (i) and (ii) of Proposition 2.2, however, are no longer trivial. We have now $a = B$ and $f(X) = A_1 \circ X \circ A_2$, hence $f(a) = A_1 \circ B \circ A_2$, $g(a) = A_1 \circ B \circ A_2 \circ B$, $f(a)^* a = (A_1 \circ B \circ A_2)^* \circ B$ and conditions (i) and (ii) thus translate into:

(i) $B \circ A_1 \circ B \circ A_2 = A_1 \circ B \circ A_2 \circ B$,

(ii) $A_1 \circ (A_1 \circ B \circ A_2)^* \circ B \circ A_2 \circ (A_1 \circ B \circ A_2)^* \circ B = (A_1 \circ B \circ A_2)^* \circ A_1 \circ B \circ A_2 \circ B$.

These two conditions are certainly weaker than commutativity, but a lot more tedious to check; they are of course implied by commutativity, which is much easier to check. [14] and [19] also gave necessary and sufficient conditions (see Example 4.1) for linearization which may be even more tedious to check. This was the reason for restricting ourselves to commutativity.

When B, A_1 and A_2 commute, the linear program for answering queries of the form $Q(?,?)$ is given by:

$$P': \quad Q = B \;+\; A_1 \circ B \circ A_2 \circ Q$$

The above Proposition 3.3 states that for a program P satisfying its hypotheses, recursion can be replaced by iteration in P. This result has some interesting consequences which we will state in Corollary 3.4. We first give the idea of the method we use: in order to evaluate queries on IDB predicates efficiently, we use the technique of "pushing selections into relations" and to this end we substitute set-valued functions for relations; we thus transform a logic program into a "functional" program (with relational functions returning sets, or even multisets when we will want to deal with aggregation). To every binary predicate or relation $Q(X,Y)$ we will associate two functions, q_1 and q_2: q_1 will correspond to queries on the first column of Q, i.e. queries of the form $Q(?,Y)$ or *select X from Q*, and q_2 will correspond to queries on the second column of Q, i.e. queries of the form $Q(X,?)$ or *select Y from Q*. The relations will be denoted by upper case letters and the associated relational functions by the corresponding lower case letters together with indices.

Definition 3.2 *Let $Q(X,Y)$ be an EDB or IDB predicate or relation, with X ranging over D_1 and Y ranging over D_2. Let $\mathcal{P}(D_i)$ be the powerset of D_i for $i = 1,2$. Define*

$$q_1 : \mathcal{P}(D_2) \longrightarrow \mathcal{P}(D_1) \text{ by } q_1(Y) = Q(?,Y) = \{x \mid Q(x,y) \text{ for some } y \in Y\}$$
$$q_2 : \mathcal{P}(D_1) \longrightarrow \mathcal{P}(D_2) \text{ by } q_2(X) = Q(X,?) = \{y \mid Q(x,y) \text{ for some } x \in X\}$$

The operation $+$, $\circ$ and $*$ that are defined for relations can also be defined for relational functions as follows:

(1) the union of two sets resulting from a function evaluation is denoted by $+$; hence $f(X + X') = f(X) + f(X')$,

(2) the sum of two functions having the same domain is defined by $(f + g)(X) = f(X) + g(X)$,

(3) the composition of f and g is possible provided that the image domain of g is contained in the definition domain of f, and it is defined by $(f \circ g)(X) = f(g(X))$,

(4) the star iterate of f is defined as soon as the image domain of f is equal to the definition domain of f, and it is defined by $f^*(X) = X + f(X) + \cdots + f^n(X) + \cdots$, where $f^n = f \circ f \circ \cdots \circ f$ is f composed with itself n times.

Example 3.4 For instance if P is given, as in Example 3.1 by

$$P: \quad \begin{cases} Q(X,Y) & \leftarrow \quad A(X,Y) \\ Q(X,Y) & \leftarrow \quad B(X,Z), Q(Z,Z'), C(Z',Y) \end{cases}$$

then P translates into the following relational functions

(i) for queries of the form $Q(?,Y)$ on the first column, $q_1 = a_1 + b_1 \circ q_1 \circ c_1$,

(ii) for queries of the form $Q(X,?)$ on the second column, $q_2 = a_2 + c_2 \circ q_2 \circ b_2$,

due to the commutativity hypotheses (cf. Lemma 3.2), these equations can be rewritten as

$$q_1 = a_1 + b_1 \circ c_1 \circ q_1$$
$$q_2 = a_2 + b_2 \circ c_2 \circ q_2$$

Note that q_1 and q_2 could even be defined to operate on multisets instead of sets; this fact will be used when dealing with aggregation.

3.2 Applications

We now apply Proposition 3.3 to queries.

Corollary 3.4 *Answers to queries of the form $Q(?,?)$, or $Q(X_0,?)$, or $Q(?,Y_0)$ in a recursive program P satisfying the hypotheses of Proposition 3.3 can be computed by right linear iterations.*

Proof. Since the semantics of P is equal to the semantics of P', it can be described by regular languages. Thus the semantics of an IDB predicate Q of P is directly given by the semantics of Q in P'. The answers to a query of the form $Q(?,?)$ are thus given by the facts about Q generated by the right linear iteration P'.

For queries of the form $Q_i(?,Y_0)$, their answers are given by $q_{i,1}(Y_0) = Q_i(?,Y_0) = \{x \mid Q_i(x,y), \text{ for some } y \in Y_0\}$. Then if P' is given by the equation

$$P': \quad Q_i = a'_i(R_i^1, \ldots, R_i^{n_i}) + f'_i(R_i^1, \ldots, R_i^{n_i}) \circ Q_i$$

the translation of P' in terms of relational functions is the same right linear iteration where the upper case relation symbols R_i^j's have been replaced by the corresponding lower case function symbols $r_{i,1}^j$'s, i.e.,

$$q_{i,1} = a_i'(r_{i,1}^1, \ldots, r_{i,1}^{n_i}) + f_i'(r_{i,1}^1, \ldots, r_{i,1}^{n_i}) \circ (q_{i,1})$$

and hence the answers $q_{i,1}(Y_0)$ to the query $Q_i(?, Y_0)$ are given by:

$$q_{i,1}(Y_0) = a_i'(r_{i,1}^1, \ldots, r_{i,1}^{n_i})(Y_0) + f_i'(r_{i,1}^1, \ldots, r_{i,1}^{n_i})(q_{i,1}(Y_0))$$

which can be viewed as a right linear iteration in $q_{i,1}(Y_0)$.

For queries of the form $Q_i(X_0, ?)$, similarly, their answers are given by $q_{i,2}(X_0) = Q_i(X_0, ?) = \{y \mid Q_i(x, y) \text{ for some } x \in X_0\}$, which would naturally correspond to a left linear iteration (cf. Example 3.4). However, by the commutativity hypothesis, the rules of the program P' can be written in the form:

$$Q_i = a_i'(R_i^1, \ldots, R_i^{n_i}) + Q_i \circ f_i'(R_i^1, \ldots, R_i^{n_i}).$$

We thus obtain that $q_{i,2}$ is defined by the right linear iteration:

$$q_{i,2} = a_i'(r_{i,2}^1, \ldots, r_{i,2}^{n_i}) + f_i'(r_{i2}^1, \ldots, r_{i,2}^{n_i}) \circ q_{i,2}$$

Note that we use the commutativity hypothesis a second time, because otherwise we would obtain the EDB predicates in the monomials of f_i' in the reverse order (cf. Example 3.4). Finally

$$q_{i,2}(X_0) = a_i'(r_{i,2}^1, \ldots, r_{i,2}^{n_i})(X_0) + f_i'(r_{i2}^1, \ldots, r_{i,2}^{n_i})(q_{i,2}(X_0)) \qquad \square$$

Corollary 3.5 *Let P be a recursive logic program satisfying the hypotheses of Proposition 3.3, then boundedness is decidable for P.*

Proof. P is bounded if and only if all its trace languages are finite [4, 6]. Here the trace languages are regular, and finiteness is decidable for regular languages. $\square$

Example 3.5 Consider a database D representing moves on a map, and having as EDB predicates E, W, S, N, where $E(x, y)$ holds if and only if y is one distance unit *East* from x, $W(x, y)$ holds if and only if y is one distance unit *West* from x, etc. Clearly E, W, S, N all commute pairwise. Let P be the recursive logic program representing some moves on that database and defined by:

$$P \quad \begin{cases} M(x,y) & \leftarrow & S(x,z), W(z,y) \\ M(x,y) & \leftarrow & W(x,z), M(z,u), S(u,y) \\ M(x,y) & \leftarrow & W(x,z), M(z,u), M(u,v), M(v,w), S(w,y). \end{cases}$$

P is associated with the single equation:

$$M = S \circ W + W \circ S \circ M + W \circ S \circ M \circ M \circ M,$$

i.e. $X = S \circ W + (W \circ S + W \circ S \circ X \circ X) \circ X$.

By Propositions 2.2 and 3.3, its least fixpoint or least Herbrand model μP is

$$SW.f(WS)^* = SW.\big(WS + (WS)^3\big)^* = SW.(SW)^*$$

whence $M = SW.(SW)^* = (SW)^+ = SW + (SW)^2 + (SW)^3 + \ldots$ Thus M coincides with the least model of

$$P' : X = SW + (SW)X.$$

P' provides a right linear iterative program for answering queries of the form $M(?,?)$. □

This result can be generalized to chain rule programs containing not only Horn clause rules, but also aggregate functions, provided that the program be properly stratified, so that there is no "aggregation through recursion". More precisely, the clauses of P should be decomposable into a disjoint union $P = P_1 \cup \cdots \cup P_n$ such that the following conditions are satisfied:

(1) for every clause $Q_i \leftarrow A_1 \circ \cdots \circ A_{n_i}$ of P_j, the IDB predicates $Q_1, \ldots, Q_p$ which occur in the body of the rules defining Q_i are defined in the P_k's , $k \leq j$. (i.e. these IDB predicates shall not occur in the head of a clause in a P_k, $k > j$).

(2) all clauses involving an aggregate function h are of the form:

$$Q_1(x,e) \leftarrow A_1(x,x_1), A_2(x_1,x_2), \ldots, A_{n_i}(x_{n_i-1}, x_{n_i}) h(\langle x_{n_i} \rangle, e)$$

where $h : \mathcal{P}(D) \to D'$ is a functional relation taking as argument the set (or multiset) $X_{n_i} = \{x_{n_i} \mid A_1(x,x_1) \wedge A_2(x_1,x_2) \wedge \cdots \wedge A_{n_i}(x_{n_i-1}, x_{n_i})\}$ and returning an element e. We will write $\langle x_{n_i} \rangle$ to denote aggregation on the variable x_{n_i}.
For instance $h = count$ will be defined as follows: $count(\langle y \rangle, e)$ will hold if and only if the multiset $\langle y \rangle$ has e elements, $ave(\langle y \rangle, m)$ will hold if and only if m is the average of the elements of the multiset $\langle y \rangle$.

(3) for all clauses of the stratum P_i of the form

$$Q_i(x,e) \leftarrow A_1(x,x_1), \ldots, A_{n_i}(x_{n_i-1}, x_{n_i}) h(x_{n_i}, e)$$

i.e. containing aggregate functions, all the predicates $A_1, \ldots, A_{n_i}$ have been previously defined in the strata P_k, $k < i$.

(4) in each stratum P_i,
   - either all the EDB predicates present in the strata P_k, $k \leq i$ and IDB predicates which have been defined in the strata P_k, $k < i$, commute
   - or there is only linear recursion in stratum P_i.

If P is a recursive logic program satisfying the conditions (1) to (4) above, then a minimal Herbrand model mP of P can be obtained by a computation respecting the strata. P_1 contains neither negation nor aggregates, hence $mP_1 = \mu P_1$ is the ordinary least Herbrand model. Assuming mP_i is a minimal Herbrand model of $P_1 \cup \ldots \cup P_i$, consider all the IDB predicates computed in mP_i as extensional, then they can be aggregated in P_{i+1}. Define then $mP_{i+1} = \mu P_{i+1}$ (cf. [3] for the case with negations). An example of such a situation is studied below.

Example 3.6 [10] The average number of children of the descendants of X_0 is given by the query $avc(X_0, ?)$, on the program $P = P_1 \cup P_2 \cup P_3$ defined by:

$$\begin{array}{lrcll} P_1 & : \begin{cases} Anc(x,y) \\ \quad ch(x,e) \end{cases} & \begin{array}{c} = \\ = \end{array} & \begin{array}{c} Par(x,y) + Anc(x,z), Anc(z,y) \\ Par(x,y), count(\langle y \rangle, e) \end{array} & \begin{array}{r} (7) \\ (8) \end{array} \\ P_2 & : chd(x,e) & = & Anc(x,y), ch(y,e) & (9) \\ P_3 & : avc(x,m) & = & chd(x,e), ave(\langle e \rangle, m) & (10) \end{array}$$

where in (8) $\langle y \rangle$ is the aggregate of the multiset of y's such that $Par(x, y)$,

$$count(\langle y \rangle, e) \Longleftrightarrow e = \mathrm{card}\,\{y \mid Par(x,y)\}$$

so $ch(x, e)$ holds if and only if e is the number of children of x. Similarly, in (10), $\langle e \rangle$ is the aggregate of the multiset of e's such that $chd(x, e)$, where $chd(x, e)$ holds if and only if e is the number of children of some descendant of x, i.e.

$$ave(\langle e \rangle, m) \Longleftrightarrow m = average(\{e \mid chd(x,e)\}) = average(chd(x,?)),$$

and $average(\langle e \rangle)$ is the average of the multiset of numbers $\langle e \rangle$.

We here are using an extension of Datalog which allows for sets and multisets as arguments to relations, and where grouping is denoted via brackets $\langle\rangle$. Then the relational functions corresponding to such relations can be defined as in the Definition 3.2, namely, for a multiset X and a relation Q on $D_1 \times D_2$, $q_2 \colon \mathcal{P}(D_1) \longrightarrow \mathcal{P}(D_2)$ and $q_2(X)$ is the multiset consisting of all y's such that $Q(x, y)$ for each $x \in X$. We will need however a slight extension in order to deal with aggregation; let $\langle q_2 \rangle \colon \mathcal{P}(D_1) \longrightarrow \mathcal{P}\big(\mathcal{P}(D_2)\big)$ be defined by:

- $\langle q_2 \rangle(x) = Q(x, ?) = \{y \mid Q(x,y) \text{ for some } x \in \{x\}\} = q_2(x)$, for individual variables x,
- $\langle q_2 \rangle(x_1, \ldots, x_n) = \big(q_2(x_1), \ldots, q_2(x_n)\big)$, for multisets $\{x_1, \ldots, x_n\}$ of variables.

For aggregate relations, the definitions are similar, for instance $ave_1(m)$ is the set of multisets whose average is m, $ave_2(\langle e \rangle)$ is $average(\langle e \rangle)$, and $ave_2(\{\langle e \rangle, \langle e \rangle, \langle e' \rangle\})$ is $\{average(\langle e \rangle), average(\langle e \rangle), average(\langle e' \rangle)\}$.

Then, the answers to the query $avc(X_0, ?)$ can be obtained by the right linear program $P_1' \cup P_2' \cup P_3'$:

$$\begin{array}{lrcl} P_1' & : \begin{cases} Anc_2 \\ \quad ch_2 \end{cases} & \begin{array}{c} = \\ = \end{array} & \begin{array}{c} Par_2 + Par_2 \circ Anc_2 \\ count_2 \circ \langle Par_2 \rangle \end{array} \\ P_2' & : chd_2 & = & ch_2 \circ Anc_2 \\ P_3' & : avc_2 & = & ave_2 \circ \langle chd_2 \rangle \end{array}$$

so that, for instance,

$$ch_2(x_1, \ldots, x_p) = \big(count_2 \circ Par_2(x_1), \ldots, count_2 \circ Par_2(x_p)\big) = (n_1, \ldots, n_p),$$

or

$$\begin{aligned} avc_2(x_1, \ldots, x_p) &= ave_2 \circ \langle chd_2 \rangle(x_1, \ldots, x_p) \\ &= \big(ave_2 \circ chd_2(x_1), \ldots, ave_2 \circ chd_2(x_p)\big). \end{aligned}$$

Note that, if we were interested in the average number of children of the descendants of a given subpopulation, this could easily be obtained by the formula:

$$avc_2^G(x_1, \ldots, x_p) = ave_2 \circ chd_2(x_1, \ldots, x_p) = ave_2\big(chd_2(\{x_1, \ldots, x_p\})\big).$$

The iteratively computed answers to the query $avc(X_0, ?)$ will be given by the regular expression:

$$avc(X_0, ?) = ave_2 \circ count_2 \circ \langle Par_2^+ \rangle$$

or in expanded form

$$avc(X_0, ?) = ave(count(\langle \sum_{n=1}^{\infty} Par^n(X_0, ?)\rangle, ?), ?)$$

Assuming we are given an EDB with information going up to the third generation, with the following pairs in the relation Par

Parent	**Child**
Peter	Charles
Peter	Mary
Peter	Ann
Charles	Alex
Charles	Beth
Charles	Sally
Mary	John
Ann	Henry

Parent	**Child**
Alex	Alice
Henry	Emily
John	Sam
John	Molly
Sally	Elisa
Sally	Anthony
Sally	Jack
Sally	Louis

Then we find, for instance,

$$\begin{aligned} chd_2(\text{Charles}) &= (1, 0, 4) \\ chd_2(\text{Charles, Mary}) &= \big((1, 0, 4), (2)\big) \\ chd_2(\text{Peter}) &= (3, 1, 1, 1, 0, 4, 2, 1) \\ avc_2(\text{Charles, Mary, Ann}) &= (5/3, 2, 1) \\ avc_2(\text{Peter}) &= 13/8 \end{aligned}$$

while

$$\begin{aligned} avc_2^G(\text{Charles, Mary, Ann}) &= ave_2\big(chd_2(\text{Charles, Mary, Ann})\big) \\ &= ave_2(1, 0, 4, 2, 1) = 8/5 = 1.6 \end{aligned}$$

which is different from

$$average\big(avc_2(\text{Charles, Mary, Ann})\big) = average(5/3, 2, 1) = 14/9 = 1.555\cdots$$

Another consequence of Proposition 3.3 is to yield a sufficient condition of applicability of the result of [4] stating that a chain-rule program, together with a query $Q(X_0, ?)$ is equivalent to a monadic program (i.e. a program P' all of whose IDB predicates are unary), if and only if its trace language is regular.

Corollary 3.6 *For a recursive logic program P satisfying the hypotheses of Proposition 3.3, any query of the form $Q(X_0, ?), Q(?, X_0)$ or $Q(X_0, X_0)$ can be computed by an equivalent monadic program.*

Proof. By the part 2) of Proposition 3.3, the trace language is regular. Note that in addition, for queries $Q(X_0, ?)$ or $Q(?, X_0)$ the proof of Proposition 3.3 gives the method to construct the monadic program. $\square$

4 Discussion and conclusions

Recently, a number of papers [9, 14, 15, 18, 20] have studied linearizations of Horn clause programs. All of the methods that are effective do not address the general problem of whether a program P is linearizable, but the more specific question of whether the program P is linearizable *and equivalent to a given linear program P'* [9, 15, 18, 20]. One reason for this is the fact that linearization is in general undecidable [14, 15].

The various effective approaches can be classified according to two criteria:

(i) the method used for the linearization,

(ii) the class of programs which they are able to handle.

We sketch in the sequel a brief overview according to these two criteria:

(1) For the first criterion, namely the method used for the linearization, most studies [14, 15, 18, 17, 20] use proof trees and their transformations to linearize programs. The basic idea is

  (a) to expand the original program into a set of proof trees representing conjunctive queries with their derivations.

  (b) take the "bad" proof trees, corresponding to non linear conjunctive queries, and check whether they can be embedded into "good" ones, corresponding to linear conjunctive queries

  (c) when the method is effective, show that the depth of the containing proof trees can be bounded.

  The exceptions are:

  – Ioannidis and Wong, who use relational algebra methods, but note that "...essentially, a proof tree is another representation of a product in a nonassociative closed semiring ...",

  – Saraiya, who uses proof trees for linearization, but uses a language theory encoding to show the undecidability of the general linearization problem (for programs with only one IDB predicate, one bilinear rule, possibly some linear rules and five initialization non-recursive rules).

  Our method on the other hand makes use of language theory for the linearization.

(2) Consider now the second criterion, namely the class of programs which are linearized; at most bilinear sirups, i.e. bilinear recursions with a single recursive rule, are shown to be linearizable in all papers: in a bilinear sirup, each recursive rule body may contain at most two occurrences of the IDB predicates, together with various additional restrictions, whereas we deal with multiple rules and multiple IDB predicates. Bilinear recursions are also called "doubly recursive" programs in [15, 17, 20].

– the earliest work on linearization is [17], where a necessary and sufficient condition is stated for the linearization of programs of the form:

$$\left\{\begin{array}{lcl} Q(X_1,\ldots,X_n) & \longleftarrow & Q(Y_{1,1},\ldots,Y_{1,n_1}), Q(Y_{2,1},\ldots,Y_{2,n_2}), \\ & & \hfill A(Y_{3,1},\ldots,Y_{3,n_3}) \\ Q(X_1,\ldots,X_n) & \longleftarrow & B(X_1,\ldots,X_n) \end{array}\right.$$

consisting of one IDB predicate Q defined by a single bilinear recursive rule having at most one EDB predicate (A), together with one initialization rule,

– Saraiya extends that result by allowing for arbitrary many pairwise different EDB predicates in the bilinear recursive rule, and several initialization rules; he also studies the complexity of the linearization (Ptime), which becomes NP-hard (resp. undecidable) if one allows the *same* EDB predicate, with $n \leq 4$ (resp. $n \geq 5$) in the bilinear recursive rule.

Both of the above papers use proof tree containments to linearize and show that they can bound the depth of the containing proof trees to be considered. It is shown in [14] that if repetitions of the same EDB predicate are allowed in the non recursive subgoals of the bilinear recursive rule, then, the number or the depth of the proof tree containments to be checked can no longer be bounded, whence the undecidability of the linearization.

The most general condition for linearization appears in [14], and, in an equivalent form, albeit obtained by different tools, in [9], where necessary and sufficient conditions are given for the right (resp. left) linearization of bilinear programs. By the above remarks, these conditions must be undecidable. For the sake of clarity we will discuss only the right linearization condition on an example. The left linearization condition is symmetric.

Example 4.1 (cf. Example 3.3) Consider the program:

$$\left\{\begin{array}{lcl} Q(X,Y) & \longleftarrow & A_1(X,Y_1), Q(Y_1,Y_2), A_2(Y_2,Y_3), Q(Y_3,Y) \\ Q(X,Y) & \longleftarrow & B(X,Y) \end{array}\right.$$

that we rewrite, according to our notations, into:

$$Q = A_1 \circ Q \circ A_2 \circ Q \ + \ B$$

Then, on the present example, the condition of right linearization, called reverse power-subassociativity in [14], or power-right-subalternativity in [9], boils down to the following: for all binary relations R, for all $n, m \geq 1$, there exists $k \geq 1$, such that:

$$A_1 \circ R^n \circ A_2 \circ R^m \subseteq f^k(R)$$

where $f^k(R)$ is defined as follows:

$$\begin{aligned} f^0(R) &= R \\ f^1(R) &= A_1 \circ R \circ A_2 \circ R \\ &\vdots \\ f^{n+1}(R) &= A_1 \circ f^n(R) \circ A_2 \circ R \end{aligned}$$

This condition is necessary and sufficient. However, as noted in [9]: "Unfortunately, all properties that are equivalent to [right]-linearizability ... require testing for containment of recursive programs, which is in general undecidable [16]", see also [1].

So, Ioannidis and Wong introduce a much simpler *sufficient* condition, right-subalternativity, which implies that the program is right linearizable, and which would be expressed in the case of the present example by, for all binary relations R, S:

$$A_1 \circ (A_1 \circ R \circ A_2 \circ S) \circ A_2 \circ S = A_1 \circ R \circ A_2 \circ (A_1 \circ S \circ A_2 \circ S)$$

Now, neither our condition, nor the right-subalternativity are necessary conditions, and moreover, these conditions are incomparable, as shown below.

Let Q be defined by the equations (5), and let the domain of the database be $D = \{1, 2, 3, 4, 5, 6\}$.

(1) If A_1 and A_2 do not commute, our condition will not apply, whereas the Ioannidis and Wong condition may apply.

(2) Take

$$\begin{aligned} A_1 &= \{(1,2)\} \\ A_2 &= \{(2,1)\} \\ B &= \{(5,6)\} \end{aligned}$$

Then $A_1 \circ A_2 \neq A_2 \circ A_1$ and our condition does not apply; the Ioannidis and Wong condition does not apply either, since taking $R = S = \{(x, y) \mid (x, y) \in D^2\}$, we have:

$$\begin{aligned} &A_1 \circ (A_1 \circ R \circ A_2 \circ S) \circ A_2 \circ S = \emptyset \\ &\qquad \neq A_1 \circ R \circ A_2 \circ (A_1 \circ S \circ A_2 \circ S) = \{(1, x) \mid x \in D\} \end{aligned}$$

However Q is linearizable and equivalent to $Q = B$, which shows that both conditions are non necessary.

(3) Take

$$\begin{aligned} A_1 &= \{(1,2)\,,\,(2,1)\} \\ A_2 &= \{(3,4)\,,\,(4,3)\} \\ B &= \{(5,6)\} \end{aligned}$$

Then, our commutativity condition trivially applies since $A_1 \circ A_2 = A_2 \circ A_1 = A_i \circ B = B \circ A_i = \emptyset$, hence Q is equivalent to the right linear program: $Q = A_1 \circ B \circ A_2 \circ Q \ + \ B = B$. However, the right-subalternativity of Ioannidis and Wong is not verified since, letting

$$\begin{aligned} R &= \{(x, y) \mid (x, y) \in D^2\} \\ S &= \{(3, x) \mid x \in D\} \end{aligned}$$

we have

$$\begin{aligned} &A_1 \circ (A_1 \circ R \circ A_2 \circ S) \circ A_2 \circ S = \{(1, x)\,,\,(2, x) \mid x \in D\} \\ &\qquad \neq A_1 \circ R \circ A_2 \circ (A_1 \circ S \circ A_2 \circ S) = \emptyset \end{aligned}$$

However, one can check easily that, in the present example, as defined by the equations (5), provided that A_1 and A_2 are regular enough, that is, right or left cancellable, then the Ioannidis and Wong condition implies that A_1 and A_2 commute.

Recall that a relation is said to be *right* (resp. *left*) *cancellable* if and only if, for all relations X and Y, $X \circ A = Y \circ A \Rightarrow X = Y$ (resp. $A \circ X = A \circ Y \Rightarrow X = Y$). Assuming for instance that A_1 is left cancellable and A_2 is right cancellable, and applying the Ioannidis and Wong condition with $R = S = id$, we obtain $A_1 \circ A_2 = A_2 \circ A_1$.

All the papers discussed so far linearize only bilinear recursions with a single recursive rule, with the exception of [9], where the algebraic framework of vectors and matrices allows for the treatment of bilinear recursions with several mutually recursive rules at no extra cost. It is also shown in [9] that multilinear recursions can be encoded into bilinear recursions, but at the price of encoding a k-linear recursion by a vector of vectors ... of vectors of bilinear recursions (i.e. $k-1$ nested levels of vectors), which complicates the treatment.

Our approch, on the other hand, although it may sound somehow restrictive because of the assumption of commutative chain rules, has nevertheless several assets:

- It is extremely simple to handle.
- Commutativity is a non unusual restriction which has been introduced quite often in databases, mostly to simplify the evaluation of queries, for example:
  - In a similar content, commutativity of EDB predicates has also been considered in [7, 9] to optimize the evaluation of transitive closures: it is shown that, if A_1 and A_2 commute, then $(A_1 + A_2)^* = A_1^* \circ A_2^*$, and of course the latter expression can be evaluated a lot more efficiently.
  - In a different context, commutativity of linear rules is assumed in [8, 18] to optimize query evaluations: two rules r_1 and r_2 defining the same IDB predicate Q are said to commute if $r_1 r_2 = r_2 r_1$, namely whenever Q is evaluated by calling first r_1 then r_2, the same result can be obtained by calling r_2 first and then r_1. We can show that (see Example 4.2 below), when the EDB predicates commute, the rules will also commute, and whatever optimizations were obtained will then apply.
- It allows to take into account aggregate functions, whereas it is not clear whether this will be possible with the other methods.

Example 4.2 For instance, the rules r_1 and r_2 commute

$$r_1 : Q(X,Y) \longleftarrow A(X,Z), Q(Z,Y)$$
$$r_2 : Q(X,Y) \longleftarrow Q(Z,Y), B(Z,Y)$$

since both $r_1 r_2$ and $r_2 r_1$ lead to the same query $A(X,W), Q(W,Z), B(Z,Y)$, modulo renaming of variables.

The rules r'_1 and r'_2 do not commute [18]

$$r'_1 : Q(X_1, X_2, X_3, X_4) \longleftarrow Q(X_2, X_3, X_4, X_1)$$
$$r'_2 : Q(X_1, X_2, X_3, X_4) \longleftarrow Q(X_4, X_3, X_2, X_1)$$

since $r'_1 r'_2$ leads to $Q(X_1, X_4, X_3, X_2)$ and $r'_2 r'_1$ leads to $Q(X_3, X_2, X_1, X_4)$.

Commutativity of rules can be extended to nonlinear rules and we can show that

Proposition 4.1 *Let P satisfy the hypotheses of Proposition 3.3, then all the rules of P with Q occurring in their head and their body commute.*

Proof. Since all IDB and EDB predicates commute by Lemma 3.2, it is enough to know the predicates occurring in a rule body, and we can omit the variables, so that, if,

$$r_1 : Q \longleftarrow Q^1_1 \cdots Q^1_{i-1} Q Q^1_{i+1} \cdots Q^1_{n_1}$$
$$r_2 : Q \longleftarrow Q^2_1 \cdots Q^2_{j-1} Q Q^2_{j+1} \cdots Q^2_{n_2},$$

then $r_1 r_2$ results in

$$Q^1_1 \cdots Q^1_{i-1} Q^2_1 \cdots Q^2_{j-1} Q Q^2_{j+1} \cdots Q^2_{n_2} Q^1_{i+1} \cdots Q^1_{n_1}$$

and $r_2 r_1$ results in

$$Q^2_1 \cdots Q^2_{j-1} Q^1_1 \cdots Q^1_{i-1} Q Q^1_{i+1} \cdots Q^1_{n_1} Q^2_{j+1} \cdots Q^2_{n_2}$$

which are equivalent queries due to the commutativity of all the Q^j_i's. Note that the Q^j_i's may be EDB predicates. IDB predicates, or may be equal to Q. □

As for the fact that we consider only chain programs, we note that:

- Programs usually found in the literature are most often chain programs.
- Among non linear programs, chain programs correspond to really nested recursions, where the recursive subgoals cannot be easily evaluated in parallel, and thus really need linearization. for instance, in a chain program such as

$$\left\{ \begin{array}{lcl} Q(X,Y) & \longleftarrow & Q(X,Z), Q(Z,Y) \\ Q(X,Y) & \longleftarrow & B(X,Y) \end{array} \right.$$

most evaluation methods for queries of the form $Q(X, ?)$ will evaluate the first rule by pushing selections strategies, or sideways information passing strategies, finding Z for the first subgoal and passing the result to the second subgoal. Evaluating $Q(X,Z)$ and $Q(Z,Y)$ in parallel would be most inefficient, since that would lead to computing many useless Z's, which would then be eliminated via a costly join. The linearized program

$$\left\{ \begin{array}{lcl} Q(X,Y) & \longleftarrow & Q(X,Z), Q(Z,Y) \\ Q(X,Y) & \longleftarrow & B(X,Y) \end{array} \right.$$

is much more efficient.

On the other hand, a non chained recursion such as

$$\begin{cases} Q(X,Y) & \longleftarrow & Q(X,Y_1), Q(Y,Y_2) \\ Q(X,Y) & \longleftarrow & B(X,Y) \end{cases}$$

should not be linearized, because evaluating $Q(X,Y_1)$ and $Q(Y,Y_2)$ in parallel is more efficient (minimally twice as fast) that evaluating them sequentially via the linearized program: this stems from the fact that there is no nesting of recursion nor information passing between the two recursive subgoals $Q(X,Y_1)$ and $Q(Y,Y_2)$ which can thus be evaluated independently with a better efficiency; hence such recursions should not be linearized!

5 Appendix

This appendix is devoted to the proofs of Propositions 2.3 and 2.4. The first step is rather standard: we show that rational functions are continuous and apply Tarski's fixpoint theorem. Recall that a function $f : E \to E$ is *monotone* if, for each $x, y \in E$, $x \le y$ implies $f(x) \le f(y)$. It is *continuous* if, for every increasing chain $x_0 \le x_1 \le x_2 \le \cdots$, $\sup_{n\ge 0} f(x_n) = f(\sup_{n\ge 0}(x_n))$. Observe that any continuous function is increasing.

Proposition 5.1 *Every rational function from E into E is continuous.*

Proof. Let $\mathcal{C}$ be the set of all continuous functions from E into E. Clearly $\mathcal{C}$ contains the identity and the constants. We show that if $(f_i)_{i\in I}$ is a family of functions of $\mathcal{C}$, then $\Sigma_{i\in I} f_i \in \mathcal{C}$. Indeed, since for an increasing chain $(x_n)_{n\ge 0}$, $\sup_{n\ge 0} x_n = \Sigma_{n\ge 0} x_n$, we have

$$\sup_{n\ge 0} \sum_{i\in I} f_i(x_n) = \sum_{n\ge 0}\sum_{i\in I} f_i(x_n) = \sum_{i\in I}\sum_{n\ge 0} f_i(x_n) = \sum_{i\in I} \sup_{n\ge 0} f_i(x_n) = \sum_{i\in I} f_i(\sup_{n\ge 0} x_n)$$

Next, we claim that if $f, g \in \mathcal{C}$, then $fg \in \mathcal{C}$. Indeed, we have

$$\begin{aligned} fg\Big(\sup_{n\ge 0} x_n\Big) &= f\Big(\sup_{n\ge 0} x_n\Big) g\Big(\sup_{n\ge 0} x_n\Big) = f\Big(\sum_{n\ge 0} x_n\Big) g\Big(\sum_{n\ge 0} x_n\Big) \\ &= \Big(\sum_{n\ge 0} f(x_n)\Big)\Big(\sum_{n\ge 0} g(x_n)\Big) = \sum_{n,m\ge 0} f(x_n) g(x_m) \end{aligned}$$

But now, for every $k \ge 0$,

$$f(x_k)g(x_k) \le \sum_{0\le n,m\le k} f(x_n)g(x_m) \le \sum_{0\le n,m\le k} f(x_k)g(x_k)$$

and thus

$$\sum_{0\le n,m\le k} f(x_n)g(x_m) = f(x_k)g(x_k)$$

Therefore

$$fg\Big(\sup_{n\geq 0} x_n\Big) = \sum_{n\geq 0} f(x_n)g(x_n) = \sum_{n\geq 0} fg(x_n) = \sup_{n\geq 0} fg(x_n)$$

Finally, if $f \in \mathcal{C}$, we also have $f^* \in \mathcal{C}$ since $f^* = \sum_{n\geq 0} f^n$. Therefore $\mathcal{C}$ contains all rational functions. $\square$

Let now $g_1, \ldots, g_n$ be a sequence of rational functions from E into E and consider the system

$$\begin{cases} X_1 = g_1(X_1, \ldots, X_n) \\ \quad\quad \cdots \\ X_n = g_n(X_1, \ldots, X_n) \end{cases} \tag{6}$$

This system can be written as $X = g(X)$ where $g = (g_1, \ldots, g_n)$ is a map from E^n into itself. If E^n is ordered by the product order, g is continuous. Therefore, by a direct application of Tarski's fixpoint theorem, we obtain

Corollary 5.2 *The system (6) has a least solution, given by* $X = \sup_{n\geq 0} g^n(0)$.

We also need a slightly more technical notion. Let F be a commutative complete semiring of E. We define the set $R_0(F)$ of 0-*rational* functions from F into F as the smallest set R of functions such that

(1′) R contains the identity function and the null function $h(x) = 0$,

(2′) if $e \in F$ and $h \in R$, then $eh \in R$,

(3) if $h_1 \in R$ and $h_2 \in R$, then $h_1 + h_2 \in r$ and $h_1h_2 \in R$,

(4′) if $h \in R$, then $h^* \in R$, where $h^*(x) = h(x)^*$.

The next two propositions give the relation between rational and 0-rational functions, which is somewhat reminiscent of the connection between affine and linear functions in geometry.

Proposition 5.3 *Let* $h : F \longrightarrow F$ *be a function. Then* h *is rational if and only if there exists a* 0*-rational function* $h_0 : F \longrightarrow F$ *such that* $h = h(0) + h_0$.

Proof. If $h = h(0) + h_0$ for some 0-rational function h_0, then h is rational. Conversely, let H be the set of all functions h such that $h = h(0) + h_0$ for some $h_0 \in R_0(F)$. We show that H contains all rational functions. Clearly, the identity function and the constant functions are in H. Next, if h, $h' \in H$, then $h = h(0) + h_0$ and $h' = h'(0) + h'_0$ for some h_0, $h'_0 \in R_0(F)$. It follows that $h + h' = h(0) + h'(0) + h_0 + h'_0 = (h + h')(0) + (h_0 + h'_0)$ and $hh' = hh'(0) + (h(0)h'_0 + h'(0)h_0 + h_0h'_0)$. Now $h_0 + h'_0 \in R_0(F)$ by condition (3) above, and $h(0)h'_0 + h'(0)h_0 + h_0h'_0 \in R_0(F)$ by conditions (2') and (3). Thus $h + h' \in H$ and $hh' \in H$. Finally, if $h = h(0) + h_0$ then

$$h^* = (h(0) + h_0)^* = h(0)^*h_0^* = h(0)^* + h(0)^*h_0^+ = h^*(0) + h(0)^*h_0^+.$$

Now $h_0^+ \in R_0(F)$ by condition (4') above and $h(0)^*h_0^+ \in R_0(F)$ by condition (2'). Thus $h^* \in H$. It follows that every rational function belongs to H. $\square$

Proposition 5.4 *Let $h : F \longrightarrow F$ be a function. Then the following conditions are equivalent:*

(i) *h is rational and $h(0) = 0$,*

(ii) *there exists a rational function $f : F \longrightarrow F$ such that $h(x) = f(x)x$,*

(iii) *h is 0-rational.*

Proof. (ii) implies (i) is trivial and the equivalence of (i) and (iii) follows from Proposition 5.3. We show that (iii) implies (ii). Let H be the set of all functions of the form $h(x) = f(x)x$ for some $f \in R(F)$. It is easy to verify that H satisfies conditions (1'), (2') and (3). But H also satisfies (4'). Indeed, if $h(x) = f(x)x$, then $h^+(x) = (f(x)x)^+ = (f(x)x)^* f(x)x$ and the function $(f(x)x)^* f(x)$ is rational. It follows that H contains $R_0(F)$, since $R_0(F)$ is the smallest set of functions satisfying (1'), (2'), (3) and (4'). Every 0-rational function thus satisfies (ii). □

A function $h : F \longrightarrow F$ is said to be $*$-*linear* if, for every $x, y \in F$, $h(x^*y) = x^*h(y)$. Note that the composition of two $*$-linear functions is $*$-linear. Continuing the analogy with linear functions in geometry, we show that 0-rational functions are $*$-linear.

Proposition 5.5 *Every 0-rational function is $*$-linear.*

Proof. We observe that:

(a) the identity function and the null function are $*$-linear,

(b) if $e \in F$ and if h is $*$-linear, then eh is $*$-linear,

(c) if $(h_i)_{i \in I}$ is a family of $*$-linear functions, then $\sum_{i \in I} h_i$ is $*$-linear,

(d) if h_1 and h_2 are $*$-linear, then so is h_1h_2. Indeed

$$\begin{aligned} h_1h_2(x^*y) &= h_1(x^*y)h_2(x^*y) \\ &= x^*h_1(y)x^*h_2(y) \quad \text{(by $*$-linearity)} \\ &= x^*x^*h_1(y)h_2(y) \quad \text{(by commutativity)} \\ &= x^*h_1(y)h_2(y) \quad (x^*x^* = x^*) \\ &= x^*h_1h_2(y) \end{aligned}$$

(e) if h is $*$-linear, h^+ is also $*$-linear, since

$$\begin{aligned} h^+(x^*y) &= [h(x^*y)]^+ \\ &= (x^*h(y))^+ \\ &= x^*h(y)^+ \quad \text{(commutativity)} \\ &= x^*h^+(y) \end{aligned}$$

This shows that every 0-rational function is $*$-linear. □

We are now ready to give the proof of Proposition 2.3. For the convenience of the reader, we first restate this proposition:

Let $f : E \longrightarrow E$ be a monotone function and let $a \in E$. Suppose there exists a commutative complete subsemiring F of E, containing a, such that f induces *a rational function from F into F. Then the least fixed point of the equation $X = a + f(X)X$ is $af(a)^*$.*

Proof. Since f is monotone, it suffices to verify that f satisfies conditions (i) and (ii) of Proposition 2.2. Now $a \in F$ and $f(a) \in F$ since F is stable under f. Condition (i) follows, since F is commutative. It remains only to verify condition (ii). By Proposition 5.4, $g(x) = f(x)x$ is 0-rational, and hence $*$-linear by Proposition 5.5. Now since F is complete, $f(a)^* \in F$ and $f(a)^*a \in F$. Therefore

$$g(f(a)^*a) = f(a)^*g(a)$$

and thus condition (ii) is satisfied. □

We now come to Proposition 2.4:

Let

$$P: \begin{cases} X_1 &= g_1(X_1, \ldots, X_n) \\ & \ldots \\ X_n &= g_n(X_1, \ldots, X_n) \end{cases}$$

be a system of recursive equations where each equation $X_i = g_i(X_1, \ldots, X_n)$ can be written in the form:

$$X_i = a_i(X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n) + f_i(X_1, \ldots, X_n)X_i,$$

where a_i and f_i are rational functions given by rational expressions all of whose coefficients commute. Then the least fixpoint of P is a rational function which can be computed by a right linear iteration.

Proof. Let F be the complete semiring generated by the coefficients of the a_i's and of the g_i's. By assumption, F is a commutative semiring. By Corollary 5.2, the least solution of the system exists and is given by $X = \sup_{n \geq 0} g^n(0)$. It follows in particular that the least solution belongs to F^n. Therefore, it suffices to solve the system in the commutative semiring F. We now prove the theorem by induction on n. If $n = 1$, then the system reduces to $X_1 = a + f_1(X_1)$ and since F is commutative, it suffices to apply Proposition 2.2. For $n > 1$, it suffices to solve the last equation with respect to X_n in the commutative semiring $F[[X_1, \ldots, X_{n-1}]]$ of series in commutative variables $X_1, \ldots, X_{n-1}$ with coefficients in F. For instance, the last equation can be written as

$$X_n = a_n(X_1, \ldots, X_{n-1}) + f_n(X_1, \ldots, X_n)X_n$$

and its solution is the rational expression

$$X_n = a_n(X_1, \ldots, X_{n-1})f_n(X_1, \ldots, X_{n-1}, a_n(X_1, \ldots, X_{n-1}))^*,$$

which can then be substituted for X_n in the first $(n-1)$ equations. Now, by induction the solution of the new system with $(n-1)$ equations is a rational function which can be computed by a right linear iteration. □

Acknowledgements

The authors would like to thank the anonymous referees for their insightful reading and their numerous suggestions to improve the quality of this paper.

References

[1] S. Abiteboul, Boundedness is undecidable for Datalog programs with a single recursive rule, IPL **32** (1989), 281–287.

[2] K.R. Apt, Introduction to Logic Programming, Rep. CWI-CS-R8826, Amsterdam (1988).

[3] K. Apt, H. Blair, A. Walker, Towards a theory of declarative knowledge, in Foundations of Deductive DataBases and Logic Programming, J. Minker Ed., Morgan-Kaufman, Los Altos (1988), 89–148.

[4] F. Bancilhon, C. Beeri, P. Kanellakis, R. Ramakrishnan, Bounds on the Propagation of Selection into Logic Programs, Proc. 6th. A.C.M. - P.O.D.S., San Diego (1987), 214–226.

[5] S. Cosmadakis, H. Gaifman, P. Kanellakis, M. Vardi, Decidable Optimizations for Data Base Logic Programs, Proc. A.C.M. STOC Conf. (1988).

[6] I. Guessarian, Deciding Boundedness for Uniformly Connected Datalog Programs, Proc. ICDT'90, Lect. Notes in Comput. Sci. **470** (1990), 395–405.

[7] J. Han, W. Lu, Asynchronous chain recursions, IEEE Trans. Knowl. and Data Eng., 1 (1989), 185–195.

[8] Y. Ioannidis, Commutativity and its role in the processing of linear recursion, to appear in *Jour. Logic Prog.*.

[9] Y. Ioannidis, E. Wong, Towards an algebraic theory of recursion, *Jour. Assoc. Comput. Mac.*, 38 (1991), 329–381.

[10] J.M. Kerisit, La méthode d'Alexandre: une technique de déduction, Ph. D. Thesis, Paris (1988).

[11] J.W. Lloyd, Foundations of Logic Programming, Springer-Verlag, Berlin (1987).

[12] Z. Manna, Mathematical theory of computation, Mac Graw Hill, New York (1974).

[13] S. Naqvi and S. Tsur, A logical language for data and knowledge bases, Computer Science Press, New York (1989).

[14] R. Ramakrishnan, Y. Sagiv, J. Ullman, M. Vardi, Proof tree transformation theorems and their applications, Proc. 8th. A.C.M. - P.O.D.S., Philadelphia (1989), 172–181.

[15] Y. Saraiya, linearizing nonlinear recursion in polynomial time, in Proc. 8th ACM-PODS (1989), 182–189, see also Ph. D. Thesis, Stanford Univ. (1991).

[16] O. Shmueli, Decidability and expressiveness aspects of logic queries, Proc. 6th. ACM-PODS, San Diego (1987), 237–249.

[17] D. Troy, C. Yu, W. Zhang, Linearization of nonlinear recursive rules, *IEEE Trans. Soft. Eng.* **15** (1989), 1109–1119.

[18] J.D. Ullman, *Data base and knowledge-base systems*, 3rd ed. New York: Computer Science Press, (1989).

[19] J.D. Ullman and A. Van Gelder, Parallel complexity of loical query programs, *Algorithmica* **3** (1988), 5–42.

[20] W. Zhang, C.T. Yu, D. Troy, Necessary and sufficient conditions to linearize doubly recursive programs in logic databases, *ACM-ToDS* **15** (1990), 459–482.