Linearisability on Datalog Programs ⋆

Foto Afrati [a], Manolis Gergatsoulis [b] and Francesca Toni [c]

[a] *Dept. of Electrical and Computing Engineering,*
National Technical University of Athens, 157 73 Zographou, Athens, Greece,
e_mail: afrati@softlab.ece.ntua.gr

[b] *Institute of Informatics and Telecommunications, N.C.S.R. 'Demokritos'*
153 10 A. Paraskevi Attikis, Greece
e_mail: manolis@iit.demokritos.gr

[c] *Department of Computing, Imperial College*
180 Queen's Gate, London SW7 2BZ, UK
e_mail: ft@doc.ic.ac.uk

Abstract

Linear Datalog programs are programs whose clauses have at most one intensional atom in their bodies. We explore syntactic classes of Datalog programs (syntactically non-linear) which turn out to express no more than the queries expressed by linear Datalog programs. In particular, we investigate linearisability of (database queries corresponding to) *piecewise linear* Datalog programs and *chain queries*:

a) We prove that piecewise linear Datalog programs can always be transformed into linear Datalog programs, by virtue of a procedure which performs the transformation automatically. The procedure relies upon conventional logic program transformation techniques.

b) We identify a new class of linearisable chain queries, referred to as *pseudo-regular*, and prove their linearisability constructively, by generating, for any given pseudo-regular chain query, the Datalog program corresponding to it.

Keywords: Datalog programs, program transformation, program optimisation, linearisability, deductive databases, database queries.

Preprint submitted to Elsevier Science *1 April 2002*

1 Introduction

First-order (algebraic) query languages lack recursion and, as a consequence, have limited expressive power. Datalog, the language of Horn logic without function symbols, embeds recursion and therefore allows to express a far wider class of queries. However, queries expressed in Datalog are harder to evaluate than classical first-order queries (from the point of view of *parallel* complexity): whereas first-order queries can be solved in deterministic log-space (namely by a deterministic Turing machine using a workspace whose size is log n, where n is the dimension of the problem, i.e. the number of the relations tuples in the underlying database), Datalog programs are log-space complete for $\mathcal{P}$ in general (namely, Datalog queries cannot be solved in deterministic log-space, unless $\mathcal{P}$ = log-space which is believed to be highly unlikely [19]). Indeed, the prototypical $\mathcal{P}$-log-space complete *path system accessibility* problem [9] can be encoded by the Datalog program

$$access(X) \leftarrow source(X).$$

$$access(X) \leftarrow access(Y_1), access(Y_2), triple(Y_1, Y_2, X).$$

The predicates *source* and *triple* represent, respectively, source nodes and accessibility conditions: in particular, the predicate $triple(Y_1, Y_2, X)$ represents that if Y_1, Y_2 are accessible from the source nodes, then so is X.

As a consequence many efforts have been devoted to detect special classes of Datalog programs for which efficient evaluation methods and optimisation techniques exist [29,6]. These classes are defined by imposing syntactic restrictions on the Datalog programs belonging to them, following two main approaches:

- restricting the *width* (number of arguments) of the predicates defined by the Datalog programs (as, e.g., in [7,10,32]);
- imposing the *linearity* condition on the clauses of Datalog programs that

⋆ This research has been partly supported by the EEC HCM Project no CHRX-CT93-00414 "Logic Program Synthesis and Transformation".

at most one non-database predicate is allowed in the body of each clause defined by the Datalog program (as, e.g., in [17,18,24,10]).

Linear programs have been widely studied (e.g. see [1,17,3]) both as concerns their computational complexity and the efficiency of algorithms for computing their consequences. In particular, it has been shown that all Datalog programs currently known to be $\mathcal{P}$-complete require non-linear clauses, because in each case there is a first-order reduction from path system accessibility to such Datalog programs (e.g. see [11,30,3]). Finally, it is known ([1], see section 5 for more details) that there are Datalog programs in $\mathcal{NC}^2$ which are not equivalent to any linear program: these are Datalog programs corresponding to a special class of (recursive) queries, referred to in the literature as *chain queries* [30,3].

In this context, an interesting question is whether syntactic restrictions on classes of Datalog programs necessarily restrict their expressive power. In particular, linearisability of Datalog programs/recursive queries (i.e., the question whether queries expressed by certain programs can be still expressed within the class of linear programs) has been widely studied in the (deductive) database community, e.g. by [23,33,14].

As an example, consider the following Datalog program which checks if there is a path joining two nodes of a graph:

$$path(X,Y) \leftarrow arc(X,Y).$$

$$path(X,Y) \leftarrow path(X,Z), path(Z,Y).$$

Here *arc* is a database predicate. This program is not *linear*, whereas the equivalent Datalog program:

$$path(X,Y) \leftarrow arc(X,Y).$$

$$path(X,Y) \leftarrow arc(X,Z), path(Z,Y).$$

is. Thus, in this example, imposing the linearity condition did not prevent the "*path* query" to be expressible. That is, the "*path* query" can also be expressed within the class of linear programs. Thus, a natural question is: Are there (syntactic) classes of Datalog programs that have this property, that for

any query expressed by a program in the class, there is a linear program which also expresses the query? In this paper, we answer this question affirmatively for two *special classes* of Datalog programs/recursive queries. More in detail, we investigate linearisability of *piecewise linear programs* and *chain queries*. We prove that piecewise linear programs are always linearisable. Moreover, whereas it is known that chain queries are not linearisable in general [1], we prove that *regular* and *pseudo-regular* chain queries always are. To the best of our knowledge, the class of pseudo-regular chain queries has not been studied elsewhere in the literature.

We prove all linearisability results constructively, by showing how to translate any given programs/queries into corresponding linear Datalog programs. In particular, we transform piecewise linear programs into linear programs via a procedure which relies heavily upon conventional logic program transformation techniques [21,22], such as fold, unfold and Eureka definition introduction operations. The correctness of this procedure is a direct consequence of the fact that these transformation techniques are equivalence preserving.

The rest of the paper is organised as follows. Section 2 gives some preliminary notions. Section 3 and 4 present, respectively, linearisability results for piecewise linear programs and *some* classes of chain queries, namely *regular* chain queries and the newly introduced *pseudo-regular* chain queries. Section 5 reviews non-linearisability results for some other classes of chain queries. Section 6 concludes and discusses future work.

Some of the material in this paper is a revised and extended version of material from [2,5].

2 Preliminaries

Suppose that we have four disjoint, countably infinite sets of symbols namely *constants*, *variables*, *function symbols* of all *arities* and *predicates* of all *arities*. A *term* is either a constant or a variable or an expression of the form $f(u)$ where f is a function symbol of arity n and u is a n-vector of terms. An atom

is an expression of the form $p(u)$, where p is a predicate symbol of arity n and u is a n-vector of terms. A *ground atom* is an atom without variables. Let $A_0, A_1, \ldots, A_k$, with $k \geq 0$, be atoms. Then $A_0 \leftarrow A_1, \ldots, A_n$ is a *Horn clause* or a *rule* (in the following we will call it simply a *clause*). A_0 is referred to as the *head* and $A_1, \ldots, A_n$ as the *body* of the clause. A clause with an empty body ($n = 0$) is referred to as a *unit clause.* A clause with a non-empty body ($n > 0$) is referred to as a *non-unit clause.* A *definite logic program* is a set of Horn clauses. If p is the predicate in the head of a clause then the clause *defines* or is a *definition for p.*

Let P be a definite logic program. Then the *Herbrand Universe* U_P of P is the set of all ground terms that can be formed using the constant and function symbols that appear in P. The *Herbrand base* HB_P of P is the set of ground atoms whose predicate symbols appear in P and whose arguments are terms in U_P. A *Herbrand interpretation* for P is a subset of HB_P. A *Herbrand model* for P is a Herbrand interpretation which satisfies all clauses in P. The *meaning* $M(P)$ of a definite logic program P is defined as $M(P)$ given by the least Herbrand model of P. Two definite logic programs are *equivalent* if they have the same meaning. If S is a set of predicate symbols, then the *meaning* $M_S(P)$ of a definite logic program P *restricted to the predicates in* S is defined as $M_S(P) = M(P) \cap \{A|A$ is a ground atom whose predicate is in $S\}$.

2.1 Database Queries and Datalog Programs

A *relational* (or *extensional*) *database* of arity $(a_1, \ldots a_n)$, where each a_i, with $1 \leq i \leq n$, is a non-negative integer, is a tuple $(D, r_1, \ldots, r_n)$ with D a finite set (called the *domain*) and r_i, $i = 1, \ldots, n$, a relation of arity a_i over D. A *relation* r of arity a over the domain D is a finite subset of D^a i.e. its elements are a-tuples of elements of D. In logic programming terms, an extensional database is represented by a (finite) set of unit clauses without function symbols. Under this formulation, the (implicit) domain D is the Herbrand Universe of the set of unit clauses. The predicates occurring in an extensional database are referred to as *extensional* (or *EDB*) *predicates.* Atoms whose predicate is extensional

are referred to as *extensional* (or *EDB*) *atoms.*

A *Datalog program* (or *intensional database*) is a set of Horn clauses without function symbols. The predicates that appear in the head of clauses in a Datalog program are referred to as *intensional* (or *IDB*) *predicates.* Atoms whose predicate is intensional are referred to as *intensional* (or *IDB*) *atoms.* We assume that extensional predicates cannot appear in the head of clauses, and that predicates appearing only in the bodies of the clauses of a Datalog program are extensional. As a result, the sets of intensional and extensional predicates/atoms are necessarily disjoint.

A *query* of arity $(a_1, \ldots, a_n)$ to (a) is a function from extensional databases of arity $(a_1, \ldots, a_n)$ to extensional databases of arity (a). A query is expressed by a program written in some database query language. In this paper we use Datalog programs to express queries over extensional databases. Given an extensional database EDB and a Datalog program IDB, the *query corresponding to* an intensional predicate p in IDB is:

$Q_p^{IDB}(EDB) = \{d | p(d) \text{ belongs to the least Herbrand model of } EDB \cup IDB\}$ [1].

Example 1 *Let* $EDB = \{arc(a,b), arc(b,c), arc(c,d), arc(a,e)\}$ *(with implicit domain* $\{a,b,c,d,e\}$*) and let* IDB *be*

$path(X,Y) \leftarrow arc(X,Y).$

$path(X,Y) \leftarrow arc(X,Z), path(Z,Y).$

Then, the query corresponding to the (intensional) predicate path is:

$Q_{path}^{IDB}(EDB) = \{(a,b),(b,c),(c,d),(a,e),(a,c),(a,d),(b,d)\}.$

In this paper we will study the transformation of some, syntactically defined classes of Datalog programs and queries into special "linear" Datalog programs, defined as follows:

Definition 2 *A* linear program *is a Datalog program such that every clause*

[1] In general, d is a tuple of elements of the domain D.

in the program has at most one intensional atom in its body.

The program in example 1 is linear.

Notice that the notion of linear Datalog programs presented above has been previously used in the literature [24,10].

Definition 3 *A* linearisable program IDB *is a Datalog program such that there exists a linear program* IDB' *satisfying the property that, for every extensional database* EDB*, the meaning of* $EDB \cup IDB$ *coincides with the meaning of* $EDB \cup IDB'$ *restricted to the predicates occurring in* $EDB \cup IDB$*.*
A linearisable query *is a query corresponding to a linearisable program.*

Notice that, since the database EDB in the earlier definition is not fixed, the Herbrand universe of EDB is not fixed either.

In this paper we study linearisability of "piecewise linear programs" and "chain queries", defined below.

Definition 4 *A* piecewise linear program *is a Datalog program* IDB *such that for every clause in* IDB *there is at most one atom in the body whose predicate is "mutually recursive" with the predicate in the head, where*

- *two predicates* p *and* q *are said to be* mutually recursive *iff* p *"depends on"* q *and* q *"depends on"* p*, where "depends on" is the least relation such that: a predicate* p depends on *a predicate* q *iff there is a clause with* p *in its head and either* q *or a predicate* r *which depends on* q *in its body.*

It is easy to see that every linear program is piecewise linear. However, piecewise linear programs are not guaranteed to be linear in general.

Example 5 *The following Datalog programs are piecewise linear but not linear:*

Program 1:

$path(X,Y) \leftarrow arc(X,Y).$

$path(X,Y) \leftarrow arc(X,Z), path(Z,Y).$

$double_path(X,Y) \leftarrow path(X,Y), path(Y,X).$

Program 2:

$ancestor(X,Y) \leftarrow parent(X,Y).$

$ancestor(X,Y) \leftarrow parent(X,Z), ancestor(Z,Y).$

$parent(X,Y) \leftarrow mother(X,Y).$

$parent(X,Y) \leftarrow father(X,Y).$

where $arc, father$ *and* $mother$ *are extensional predicates.*

Note that the path system accessibility program given in the Introduction is neither linear nor piecewise linear.

In this paper we will refer to extensional databases with binary relations only as *graphs*. Note that the extensional database EDB in example 1 is a graph. Moreover, any EDB for the extensional predicates arc, $father$ and $mother$ in example 5 would be a graph too.

Definition 6 *Let* EDB *be a graph and* Σ *be the (finite) alphabet containing a letter* R_i *for each relation* r_i *in* EDB*. Then, the* chain query *for a language* $L \subseteq \Sigma^*$ *is:*

$CQ_L(EDB) = \{(u,v) |$ *there exists a "path in* EDB *from* u *to* v *spelling a word" in* $L\}$,

where a path from u to v spelling a word $R_{i_1} \dots R_{i_l} \in \Sigma^+$ *is a sequence* $u = u_1, \dots, u_{l+1} = v$ *of elements of the domain of* EDB *such that* $r_{i_j}(u_j, u_{j+1}) \in EDB$*, for each* $j = 1, \dots, l$*, and a* path spelling the empty word*,* ϵ*, is the sequence* u, u*, for every element* u *of the domain of* EDB*.*

Example 7 *Let* EDB *be as in example 1. Then* $\Sigma = \{Arc\}$*.*

Let $L = \{Arc^i | i \geq 1\}$. *Then*

$CQ_L(EDB) = \{(a,b),(b,c),(c,d),(a,e),(a,c),(a,d),(b,d)\}$

(= Q_{path}^{IDB} *in example 1). The spelled words are* Arc, Arc, Arc, Arc, Arc^2, Arc^3, Arc^2, *respectively.*

Example 8 *Let* $EDB = \{r(a,b), r(b,c), s(c,d)\}$ *(with implicit domain* $\{a, b, c, d\}$*). Then,* $\Sigma = \{R, S\}$. *Let* $L = \{\ R^i S^j | i,j \geq 0, i+j > 0\ \}$. *Then,*

$CQ_L(EDB) = \{(a,b),(a,c),(a,d),(b,c),(b,d),(c,d)\}$.

The spelled words are R, RR, RRS, R, RS, S, *respectively.*

Finally, we will also use the following notion:

Definition 9 *The* transitive closure of a predicate p w.r.t. a program P *is the set of clauses* S_p, *where* $S_p \subseteq P$, *defined as follows:*

(i) *If the predicate of the head of* C *is* p, *then* C *belongs to* S_p.
(ii) *Let* C *be a clause in* S_P *and* p' *be the predicate of an atom in the body of* C. *Then the clauses in the transitive closure* S_p' *of* p' *w.r.t.* P *are also in* S_p.
(iii) *All clauses in* S_p *are generated by applying the above rules.*

2.2 Logic Program Transformation

In the transformation system of Tamaki & Sato [27] [2], a sequence $P_0, \ldots, P_n$ of definite logic programs is generated, starting from the *initial program* P_0, by applying the unfold/fold transformation rules [27,20,22,12,13], defined below, and by introducing clauses defining new predicates (called Eureka definitions) [20]. The unfold/fold transformation rules preserve the meaning of definite logic programs. The clause introduction rule preserves the meaning of the definite logic program it is applied to, restricted to the predicates occurring

[2] In the following we adopt it in the formulation which appears in [15]

in the program before the rule is applied.

Definition 10 *An* initial program P_0 *is a program satisfying the following conditions:*

(i) P_0 *is divided into two disjoint sets of clauses,* P_{new} *and* P_{old}. *The predicates defined by* P_{new} *are called* new predicates, *while those defined by* P_{old} *are called* old predicates.

(ii) The new predicates never appear in P_{old} *nor in the bodies of the clauses in* P_{new}.

Note that, although clauses defining new predicates (*Eureka definitions*) can be introduced in any program of the transformation sequence, $P_0, \ldots, P_n$, we will assume that all these definitions are in P_{new} in P_0 to start with.

Definition 11 *Let* C *be the clause* $A \leftarrow B, K$ *in* P_l, *with* $l \geq 0$, *where* B *is an atom and* K *a conjunction of atoms, and* $C_1, ..., C_m$ *be all clauses in* P_l [3] *whose heads are unifiable with* B *by most general unifiers* $\theta_1, ..., \theta_m$, *respectively.*
The result of unfolding C at B *is the set of clauses* $\{C'_1, ..., C'_m\}$ *such that if* C_j, *with* $1 \leq j \leq m$, *is* $B_j \leftarrow Q_j$, *where* Q_j *is a (possibly empty) conjunction of atoms, then* C'_j *is* $(A \leftarrow Q_j, K)\theta_j$.
Then, $P_{l+1} = (P_l - \{C\}) \cup \{C'_1, ..., C'_m\}$.
C *is called the* unfolded clause *and* $C_1, ..., C_m$ *the* unfolding clauses. B *is called the* unfolded atom.

Definition 12 *Let* C *be the clause* $H \leftarrow K, L$ *in* P_l *and* F *the clause* $A \leftarrow K'$ *in* P_{new}, *where* K, K', *and* L *are conjunctions of atoms.*
Then, the clause $C' : \; H \leftarrow A\theta, L$ *is the* result of folding C using F *if there exists a substitution* θ *satisfying the following conditions:*

(i) $K'\theta = K$.

(ii) All variables in the body of F *which do not appear in the head of* F *are mapped through* θ *into distinct variables which do not occur in* C'.

[3] It has been shown [20] that in general one can choose any $P_j, j \leq l$, rather than just P_l. We omit this possibility here as this plays no role in the methodology we propose later, in section 3.

(iii) *F is the only clause in P_{new} whose head is unifiable with $A\theta$.*

(iv) *Either the head predicate of C is an old predicate, or C has been unfolded at least once in the sequence $P_0, P_1, ..., P_{l-1}$.*

Then, $P_{l+1} = (P_l - \{C\}) \cup \{C'\}$.
C is called the folded clause, *and F is called the* folding clause.

Note that this definition prevents self-folding (see part (iv)), namely folding where the same clause serves as both folded and folding clause, which does not preserve the meaning of definite logic programs.

Note that more powerful unfold/fold transformation systems than the Tamaki & Sato's we use in this paper have been proposed in the literature [28,8,13]. In particular, in [28], recursive clauses are allowed to be used as folding clauses. In the system proposed in [13,12] simultaneous folding of more than one clauses is allowed, while in [12] simultaneous folding using recursive Eureka definitions is allowed. A lot of research work has also been done (see for example [25,26]) towards the definition of unfold/fold transformation systems that preserve various semantics of logic programs which allow negative atoms in the clause bodies. Although in this paper we do not need such additional features, it would be interesting to investigate the usefulness of such systems in optimizing transformations of database logic programs.

In the remainder of the paper we will rely upon the program transformation methodology proposed in [22].

Definition 13 *An* unfolding selection rule *(*U-rule *for short) is a (partial) function from clauses to atoms. The value of the function for a clause is a body atom called the* selected atom.

Definition 14 *Let P be a program, C a clause and S a U-rule. An* unfolding tree *(or* U-tree *for short) T for $< P, C >$ via S is a tree labeled with clauses, constructed as follows:*

- *C is the root label of T, and*

– *if M is a node labeled by a clause C and B is the atom selected by S in C, then, for each clause C' in the result of unfolding C at B, there is a child node N of M labeled by C'.*

Definition 15 *A nonempty tree T' is called an* upper portion *of a tree T iff the following hold:*

– *The root node N of T' is also the root node of T.*
– *For every node N of T', N is also a node of T and either N is a leaf node of T' or all child nodes of N in T are also child nodes of N in T'.*

An upper portion of T consisting of a single node is called a trivial upper portion.

It can be shown [22] that for any program P and clause C, if L is the set of leaves of an upper portion of a U-tree for $< P, C >$ via an U-rule S, then $M(P \cup \{C\}) = M(P \cup L)$.

3 Transforming Piecewise Linear Programs into Linear Programs

In this section we show that every piecewise linear Datalog program can be transformed into an equivalent linear program. We show this constructively by presenting a procedure which performs the transformation. The procedure uses unfold/fold transformations and introduction of Eureka definitions. The procedure repeatedly applies a procedure which replaces by linear programs non-linear clauses of a special kind, referred to as "minimally non-linear clauses" (see definition 17 below), which are always guaranteed to exist in piecewise linear programs containing non-linear clauses (see lemma 18 below).

The following example illustrates the overall behavior of the procedure.

Example 16 *Let $P = \{C_1, C_2, C_3, C_4, C_5, C_6\}$ be the piecewise linear Datalog program with:*

$C_1: \quad a(X,Y) \leftarrow edb1(X,Y).$

$C_2: \quad a(X,Y) \leftarrow b(X,Z), a(Z,Y).$

$C_3: \quad b(X,Y) \leftarrow edb2(X,Y).$

$C_4: \quad b(X,Y) \leftarrow edb3(X,Z), c(Z,W), b(W,Y).$

$C_5: \quad c(X,Y) \leftarrow edb4(X,Y).$

$C_6: \quad c(X,Y) \leftarrow edb5(X,Z), c(Z,Y).$

P is not linear due to the non-linear clauses C_2 and C_4. We show how C_4 can be replaced by a set of linear clauses, by applying logic program transformation rules.

First, we introduce the Eureka definition:

$D_1: \quad new1(X,Y) \leftarrow c(X,Z), b(Z,Y).$

Then, we fold C_4 using D_1, thus obtaining the linear clause:

$C_7: \quad b(X,Y) \leftarrow edb3(X,Z), new1(Z,Y).$

The Eureka definition D_1 is a non-linear clause. In order to replace it by a set of linear clauses, we unfold D_1 at '$c(X,Z)$' using the clauses C_5 and C_6, thus obtaining:

$C_8: \quad new1(X,Y) \leftarrow edb4(X,Z), b(Z,Y).$

$C_9: \quad new1(X,Y) \leftarrow edb5(X,W), c(W,Z), b(Z,Y).$

Finally, by folding C_9 using D_1 we obtain:

$C_{10}: \quad new1(X,Y) \leftarrow edb5(X,W), new1(W,Y).$

$\{C_8, C_{10}\}$ is a linear program for 'new1'. Let $P' = P - \{C_4\} \cup \{C_7, C_8, C_{10}\}$. P' is equivalent to $P \cup \{D_1\}$. P' is still not linear due to the non-linear clause C_2. Starting from P', we can replace C_2 by an equivalent set of linear clauses, by applying similar techniques to the ones above. We first introduce the Eureka definition:

$D_2: \quad new2(X,Y) \leftarrow b(X,Z), a(Z,Y).$

Then, we fold C_2 *using* D_2*, thus obtaining the linear clause:*

$$C_{11}: \quad a(X,Y) \leftarrow new2(X,Y).$$

We now unfold D_2 *at '*$b(X,Z)$*' using the clauses* C_3 *and* C_7*, thus obtaining:*

$$C_{12}: \quad new2(X,Y) \leftarrow edb2(X,Z), a(Z,Y).$$

$$C_{13}: \quad new2(X,Y) \leftarrow edb3(X,W), new1(W,Z), a(Z,Y).$$

Then, we introduce the Eureka definition:

$$D_3: \quad new3(X,Y) \leftarrow new1(X,Z), a(Z,Y).$$

Further, we fold C_{13} *using* D_3*, thus obtaining the linear clause:*

$$C_{14}: \quad new2(X,Y) \leftarrow edb3(X,W), new3(W,Y).$$

Again, in order to replace D_3 *by a set of linear clauses, we unfold* D_3 *at '*$new1(X,Z)$*' using* C_8 *and* C_{10}*, and then we fold the clauses obtained, using* D_2 *and* D_3*. In this way we obtain the linear clauses:*

$$C_{15}: \quad new3(X,Y) \leftarrow edb4(X,W), new2(W,Y).$$

$$C_{16}: \quad new3(X,Y) \leftarrow edb5(X,W), new3(W,Y).$$

The final program obtained by the above procedure is $P_{final} = \{C_1, C_3, C_5, C_6, C_7, C_8, C_{10}, C_{11}, C_{12}, C_{14}, C_{15}, C_{16}\}$. P_{final} *is a linear program. Note that, if we are interested only in the predicate 'a', we can just consider the transitive closure of 'a' w.r.t.* P_{final}*, consisting of the clauses* $C_1, C_{11}, C_{12}, C_{14}, C_{15}, C_{16}$.

Definition 17 *Let* P *be a piecewise linear program and* C *a non-linear clause in* P*. Then* C *is said to be* minimally non-linear *iff for every IDB atom in the body of* C *whose predicate* p *is not mutually recursive with the predicate of the head of* C*, the transitive closure of* p *w.r.t.* P *is a linear program.*

Note that, in example 16, C_4 is a minimally non-linear clause in P, whereas C_2 is not; however, C_2 is a minimally non-linear clause in P'.

Every piecewise linear program which is not linear is guaranteed to contain at least one minimally non-linear clause:

Lemma 18 *Let P be a piecewise linear Datalog program and N the set of non-linear clauses in P. Then, either N is empty or there is (at least) one minimally non-linear clause in N.*

Proof. We define an ordering relation $>$, over the set N consisting of all nonlinear clauses of P, as follows: $C_1 > C_2$ if C_2 is in the transitive closure wrt P of some intensional atom in the body of C_1 other than the atom which is mutually recursive with the head of C_1. It is easy to see that $>$ is asymmetric since otherwise P would not be piecewise linear. The minimally non linear clauses of P are the minimal elements of N. □

Basically, the procedure, formally given in section 3.2, selects in turn minimally non-linear clauses and replaces them by a set of linear clauses as given by the procedure, formally given in section 3.1. The procedure applies unfolding, clause introduction (giving a new Eureka definition), and folding. The unfolding steps are determined by an unfolding selection rule, uniquely determined by the set of intensional atoms in the bodies of clauses as follows:

Definition 19 *An unfolding selection rule S is a* linear unfolding selection rule *(*linear U-rule *in short) iff, for any clause C in a program P, S selects an intensional atom $p(t)$ in the body of C such that the transitive closure of p w.r.t. P is a linear program, and S is undefined for C if there is no such predicate p.*

Note that the U-rule (implicitly) adopted in example 16 is linear.

Note also that, in general, the atom selected by a linear U-rule, if any, is not uniquely defined. Therefore, there might be multiple U-trees via a linear U-rule for any Datalog program and clause. Finally:

Lemma 20 *A linear U-rule is always defined for minimally non-linear clauses in piecewise linear programs.*

Proof. Trivial, since if a clause C in a piecewise linear program P is minimally

non-linear, then there is always an atom in the body of C whose predicate's transitive closure w.r.t. P is a linear program. $\Box$

In the remainder of this section we will define formally the procedure.

3.1 Minimally Non-Linear Clause Linearisation Procedure

The following lemma implies that when we unfold a minimally non-linear clause C in a piecewise linear program P via a linear U-rule S, then S is also defined for all non-linear clauses (if any) resulting from this unfolding, as these clauses are minimally non-linear.

Lemma 21 *Let C be a minimally non-linear clause in P, S a linear U-rule and T a U-tree for $< P, C >$ via S. Then, every non-linear clause in the set of leaves L of any finite upper portion of T is minimally non-linear in $(P - \{C\}) \cup L$.*

Proof (By contradiction). Suppose that a clause $D \in L$ is not minimally non-linear. Then, there is an atom in the body of D whose predicate p is not mutually recursive with the predicate of the head of D and whose transitive closure is a non-linear program. However, the clauses in the transitive closure of p are also in the transitive closure of the predicate q of the atom selected by S in the body of C. Therefore, the transitive closure of q is not linear: contradiction. $\Box$

The following definition introduces two kinds of upper portions of U-trees that will be constructed by the procedure for deciding when to stop unfolding, which Eureka definitions to introduce and when to start performing folding.

Definition 22 *Let P be a Datalog program, C be a clause in P, S a U-rule, T a U-tree for $< P, C >$ via S. A finite upper portion U of T is said to be*

– F-linearisable wrt a set of Eureka definitions ED *iff each leaf of U*

  - *either can be folded using as folding clause a definition in ED and giving as a result a linear clause,*
  - *or is a linear clause,*
  - *or is a "failing" clause in P, where a clause is* failing in a program *iff there is an atom in the body of the clause which does not unify with the head of any clause in the program.*
- E-linearisable *iff each leaf of U is*
  - *either a linear clause,*
  - *or a failing clause in P,*
  - *or a "Eurekable" clause, where a clause D in a node of a U-tree T for $< P, C >$ via S is* Eurekable *iff there is an ancestor F of D in T and a tuple I of intensional atoms such that the tuples of all intensional atoms in the bodies of both D and F are instances of I. F is called a* folding ancestor *of D.*

U is a minimal F-linearisable (E-linearisable) upper portion *of T iff there exists no F-linearisable (E-linearisable, resp.) upper portion U' of T, with $U' \neq U$, which is also an F-linearisable (E-linearisable, resp.) upper portion of U.*

As we will see in the procedure 24, the detection of a Eurekable clause in an E-linearisable upper portion tells us that we have to stop unfolding in the corresponding branch of the U-tree and introduce a new Eureka definition. The body of the new definition consists of the tuple I. A failing clause in an F-linearisable upper portion can be removed from any program without affecting the meaning of the program.

Note that, for any Datalog program, clause in the program, U-rule and U-tree, if there exists an E-linearisable upper portion (F-linearisable upper portion wrt some given set of clauses) of the U-tree, then there exists a *unique* minimal E-linearisable (F-linearisable, resp.) upper portion. Moreover:

Lemma 23 *Let P be a Datalog program, C be a minimally non-linear clause in P, S a linear U-rule, T a U-tree for $< P, C >$ via S. Then there exists at least an E-linearisable upper portion of T.*

Proof. Since S is a linear U-rule, it always selects an atom whose transitive closure is a linear program. Thus, the number of the intensional atoms in the body of each clause resulted by unfolding a clause C (i.e. descendant of C in T) is less than or equal to the number of the intensional atoms in the body of C. Since the number of intensional predicates in P is finite, it is obvious that for every branch of T we can find in a finite depth from the root of T either a linear clause or a clause D for which there is a tuple I of intensional atoms and an ancestor clause F such that the tuple of the intensional atoms of both clauses F and D are instances of I. Therefore, there exists a finite upper portion of T with the required properties. □

Instead, even if the chosen clause is minimally non-linear and the U-rule is linear, an F-linearisable upper portion of T is only guaranteed to exist wrt some special set of clauses, for example the set ED chosen below.

Procedure 24 (Clause Linearisation procedure (CLP))

Input : *a piecewise linear program P, a minimally non-linear clause C in P and a linear U-rule S.*

Output : *a set LC of linear clauses and a set ED of clauses defining predicates not occurring in P (Eureka definitions).*

(i) Construct the minimal E-linearisable upper portion of a U-tree T for $< P, C >$ via S.

(ii) For every leaf D in U which is Eurekable via ancestor F introduce a fresh predicate symbol new and construct a clause
$E : new(X_1, \ldots, X_k) \leftarrow I$
with
(a) I a conjunction of intensional atoms such that both the conjunction ID of all intensional atoms in the body of D and the conjunction IF of all intensional atoms in the body of F are instances of I [4] and

[4] The best choice is to use as I the *most specific generalization*[16] of ID and IF. An algorithm to compute the most specific generalization of a set of expressions is given in [16].

(b) $\{X_1, \ldots, X_k\}$ *the minimal subset of the set of all variables in* I *such that both* D *and* F *can be folded using* E*.*

Let ED *be the set consisting of all* E*s constructed as above after having eliminated "copies", differing from other clauses in the set only in the names of the predicate they define and in the order of the variables in the heads.*

(iii) *Select the minimal F-linearisable upper portion* U' *of* U *wrt* ED*.*

(iv) *For each clause* E *in* ED *construct the minimal,* non-trivial *F-linearisable upper portion* U_E *wrt* ED *of a U-tree for* $< P, E >$ *via* S*.*

(v) *Let* LC *be the set of all linear clauses in the leaves of* U' *and* U_E *together with the set of all clauses resulting from the folding of the non-linear and non-failing clauses in the leaves of* U' *and* U_E *using the clauses in* ED*.*

Note that U' in step (iii) can be a trivial upper portion, whereas U_E in step (iv) is necessarily non-trivial, by definition of folding. Indeed, if U_E were trivial, then a step of self-folding would take place in step (v). But this is prohibited by definition 12, part (iv).

All clauses in ED are non-linear clauses by construction (see step (ii)). However:

Lemma 25 *Let* P *be a piecewise linear program,* C *a minimally non-linear clause in* P*,* S *a linear U-rule for* P*, and* (LC, ED) *be the output of the CLP applied to input* (P, C, S)*. Then:*

1) every clause in ED *is a minimally non-linear clause in* $P \cup ED$*;*

2) every clause in LC *is linear.*

Proof.

1) Directly from lemma 21, since C is a minimally non-linear clause in P, and by construction of the Eureka definitions (step (ii)).

2) Trivially, by construction (step (v)) and by definition of F-linearisable upper portion. □

Part 1) of this lemma implies that the linear selection rule S (used in step (i)) is always defined for the clauses in ED and the clauses in U_E, for all $E \in ED$, constructed at step (iv).

Theorem 26 (Correctness of CLP) *Let P be a piecewise linear program, C a minimally non-linear clause in P and S a linear U-rule for P. Then*

† *CLP applied to (P, C, S) terminates.*

‡ *Let LC be the set of linear clauses returned by CLP applied to (P, C, S), and $pred(P)$ be the set of predicates defined in P.*
Then, $M(P) = M_{pred(P)}((P - \{C\}) \cup LC)$.

Proof.

† *Termination*: It is sufficient to prove that it is always possible to construct 1) a minimal E-linearisable upper portion U of a U-tree for $< P, C >$ via S in step (i) of the procedure, 2) a minimal F-linearisable (wrt ED) upper portion U' of U in step (iii), and 3) for every clause E_i in ED, a minimal (non-trivial) F-linearisable (wrt ED) upper portion U_{E_i} of a U-tree for $< P, E_i >$ via S in step (iv) of the procedure.

1) Directly by lemma 23.
2) Directly by construction of the Eureka definitions (step (ii)).
3) Assume that, for the construction of U_{E_i}, we use the same U-rule S as in step (i). Since the selection performed by S is uniquely determined by the set of the intensional atoms in the body of a clause, U_{E_i} will be constructed in a similar way as the U-tree for the clause which led to the introduction of E_i. In fact, as the body of E_i has the same intensional atoms with a clause G in a leaf of U' for which E_i has been introduced, the clauses in the nodes of U_{E_i} can be put into one-to-one correspondence with the clauses in the subtree of U whose root is G. The clause in a node of U_{E_i} has the same intensional atoms with the corresponding clause in a node of U. The two clauses differ in that the EDB atoms in the body of a clause in U_{E_i} is subset of the EDB atoms of the corresponding clause in U. Thus U_{E_i} will be constructed in a finite number of unfolding steps.

‡ *Equivalence*: It is easy to see that the application of the unfold/fold transformations in the procedure 24 complies with the conditions in the definitions 10, 11 and 12. Thus, by the correctness of the transformation system we conclude that $M_{preds(P)}(P \cup ED) = M_{preds(P)}((P - \{C\}) \cup LC)$. Since $P_{new} = ED$ it is easy to by the definition 10 that $M_{preds(P)}(P \cup ED) = M_{preds(P)}(P)$. Therefore $M_{preds(P)}(P) = M_{preds(P)}((P - \{C\}) \cup LC)$. □

Example 27 *Figures 1 and 2 show the application of the CLP procedure 24 to clause C_4 of the program of example 16. In this case, the procedure returns $ED = \{D_1\}$ and $LC = \{C_7, C_8, C_{10}\}$. The underlined atoms in non-leaf nodes of the trees are the atoms selected by the U-rule.*

Figure 1 corresponds to step (i) of the procedure. The underlined tuple of atoms in the leaf is instance of the chosen tuple I (that leaf is a Eurekable clause). The detected Eurekable clause allows to introduce the Eureka definition D_1 (step (ii)).

The minimal F-linearisable upper portion of the U-tree in Figure 1 consists of a single node labeled by the clause C_4. C_4 is folded using D_1. The result of this folding (step (iii) of the CLP procedure) is the clause C_7.

Figure 2 corresponds to the construction of a linear definition for the predicate 'new1' in step (iv).

Finally, figure 3 corresponds to the step (i) of the CLP procedure applied to $< P', C_2 >$. The detected Eurekable clauses allows to introduce the definitions D_2 and D_3.

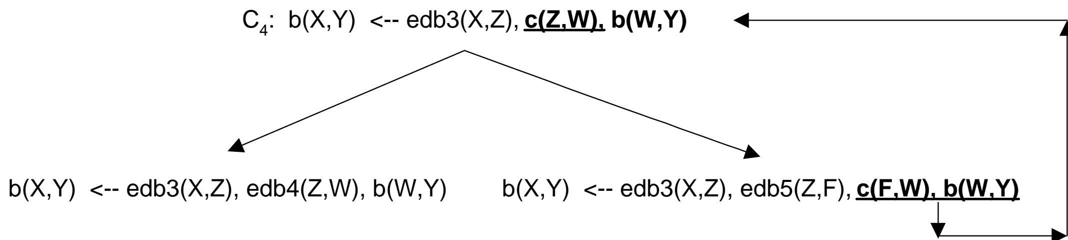

Fig. 1. A minimal E-linearisable upper portion of a U-tree for $< P, C_4 >$.

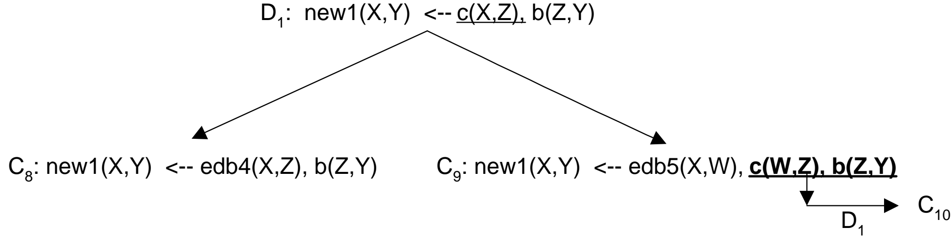

Fig. 2. A minimal (non-trivial) F-linearisable upper portion of the U-tree for $< P, D_1 >$. The non-linear leaf clause C_9 is foldable using D_1.

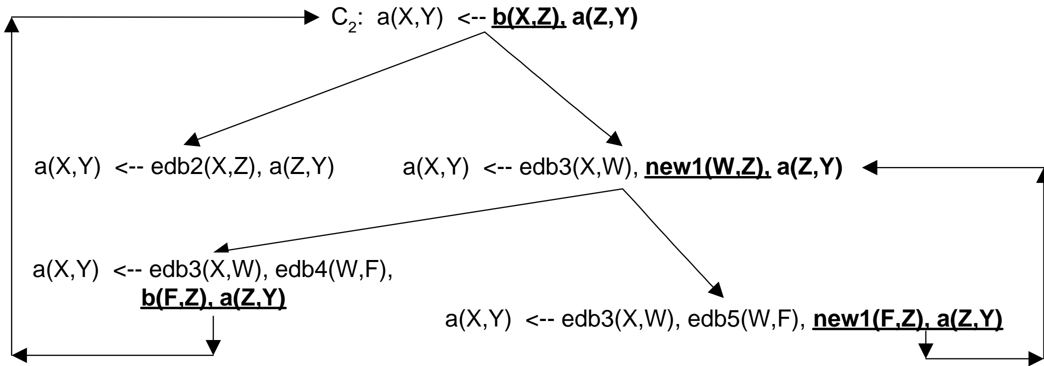

Fig. 3. A minimal E-linearisable upper portion of a U-tree for $< P', C_2 >$.

3.2 Program Linearisation Procedure

The procedure repeatedly applies the CLP, replacing the chosen minimally non-linear clause by the set of linear clauses generated by CLP for that clause.

Procedure 28 (Program Linearisation Procedure (PLP))

Input : *a piecewise linear program P and a linear U-rule S.*

Output : *a set LC of linear clauses and a set of Eureka definitions ED.*

Let $i = 0$ and $P_i = P$.

Let NL be the set of all non-linear clauses in P.

while *NL is non-empty* **do**

- *Select a minimally non-linear clause C from NL.*

- *Apply CLP to (P_i, C, S) giving LC_i and ED_i.*

- *Let $P_{i+1} = (P_i - \{C\}) \cup LC_i$.*

- *Let $NL = NL - \{C\}$, and $i = i + 1$.*

Let $ED = \bigcup_i ED_i$ for all i, and $LC = \bigcup_i LC_i$ for all i.

Theorem 29 (Correctness of PLP) *Let P be a piecewise linear program, S a linear U-rule for P. Then*

† *PLA applied to (P, S) terminates.*

‡ *Let LC be the set of linear clauses returned by PLP applied to (P, S), pred(P) be the set of predicates defined in P, and NL be the set of all non-linear clauses in P.*
Then, $M(P) = M_{pred(P)}((P - NL) \cup LC)$.

Proof.

† *Termination*: The procedure always terminates since: 1) there is a finite number of clauses in NL, 2) in each iteration of PLP exactly one clause in NL is replaced by a set of linear clauses, and 3) each iteration has a finite number of steps.

‡ *Equivalence*: Directly from the correctness of the CLP. □

It is interesting to notice that the CLP procedure does not preserve finite failure in the top-down evaluation of intensional atoms, namely, such evaluation might finitely fail w.r.t. the original (non-linear) Datalog program but might infinitely fail in the program returned by the CLP procedure. Indeed, in order to preserve finite failure, we should impose stronger conditions on the folding rule (see [25]). Besides, our procedure could be easily modified so as to preserve finite failure. In any case, the loss of finite failure does not constitute

a problem when Datalog programs are evaluated bottom-up, which is usually the case.

We have considered a class of Datalog programs, that are called *piecewise linear*, and we showed that it coincides with the class of linear Datalog programs. Up to our knowledge, it was not known until now that these two classes of programs have the same expressive power. To prove this result, we have presented a transformation from non-linear to linear programs. Questions may arise concerning the size and the efficiency of the programs obtained by the transformation. Although answering these questions is outside the scope of the paper, it is worth making the following observations. Firstly, it is important to notice that the number of new predicates introduced by the transformation depends solely on the number of IDB predicates in the original programs. In particular, this number is completely *independent* from the specific EDB database, hence the transformation can be carried out without any reference to any specific EDB database. The only relation of the proposed transformation with any possible EDB is that they share the same EDB predicate names. Moreover, the transformation of a program can be done off-line, prior to using the transformed program in conjunction with any EDB, and thus the complexity of performing the transformation is not of particular importance, especially if such complexity is weighted against multiple repeated uses of the trnasformed program with many different EDBs.

However, note that refinements of the proposed transformation might allow to reduce the number of clauses in the transformed program (e.g. by choosing appropriate unfolding selection rules, and/or by discarding redundant clauses produced by the transformation). This is however outside the scope of this paper.

4 Linearisable Chain Queries

In this section we consider chain queries for some classes of languages (regular and "pseudo-regular", defined below) and prove their linearisability. Linearis-

ability of "pseudo-regular" chain queries is proven with the help of the results in the previous section 3.

Further, we study linearisability of generic chain queries, defined as "combinations" of "simpler" chain queries.

4.1 Regular Chain Queries

Regular languages are generated by grammars with production rules of the form:

$I \to R$,

$I \to RJ$, or

$I \to JR$

where I, J are non-terminal symbols and R is a terminal symbol. The terminal symbols are elements of Σ, the (finite) alphabet for the language $L(G)$ generated by the grammar G.

It is known that chain queries for regular languages are linearisable [30]. We re-prove this result *constructively*, by generating, for each given regular chain query, the corresponding (equivalent) linear Datalog program.

Definition 30 *The* Datalog program $IDB(G)$ corresponding to a regular grammar G *is constructed as follows:*

- *each (terminal or non-terminal) symbol is mapped onto a binary (extensional or intensional, respectively) predicate;*
- *let (non-terminal) symbols I, J be mapped onto the intensional predicates i, j and (terminal) symbol R be mapped onto the extensional predicate r; then each production rule $I \to RJ$ is mapped onto a clause*

  $i(X, Y) \leftarrow r(X, Z), j(Z, Y)$

  each production rule $I \to JR$ is mapped onto a clause

$i(X,Y) \leftarrow j(X,Z), r(Z,Y)$

and each production rule $I \rightarrow R$ is mapped onto a clause

$i(X,Y) \leftarrow r(X,Y)$.

Then, the query for a regular language $L(G)$ coincides with the query corresponding to the predicate i in $IDB(G)$ on which the initial symbol I in G is mapped:

Theorem 31 *Given a regular grammar G with initial symbol I, let I be mapped onto the intensional predicate i in $IDB(G)$. For every extensional database EDB for the extensional predicates in $IDB(G)$:*

$CQ_{L(G)}(EDB) = Q_i^{IDB(G)}(EDB)$.

Proof. By definition, $Q_i^{IDB(G)}(EDB) = \{(u,v) | i(u,v)$ belongs to the least Herbrand model of $EDB \cup IDB(G)\}$. Since SLD resolution is complete with respect to the least Herbrand model semantics [31], $Q_i^{IDB(G)}(EDB) = \{(u,v)|$ there is an SLD refutation for $i(u,v)$ in $EDB \cup IDB(G)\}$.

It is easy to prove, by induction, that there is a one-to-one correspondence between derivation trees for words $R_{i_1} \dots R_{i_l}$ of $L(G)$ and SLD derivations, in $IDB(G)$ (and therefore in $EDB \cup IDB(G)$) from goals $\leftarrow i(X,Y)$ to goals $\leftarrow r_{i_1}(X,Z_1), \dots, r_{i_l}(Z_{l-1},Y)$, for some distinct variables $Z_1, \dots, Z_{l-1}$. Indeed, the application of a production rule $I \rightarrow JR$ to a non-terminal symbol I corresponds to a step of resolution between the goal $\leftarrow \dots, i(X,Y), \dots$ and the clause $i(X,Y) \leftarrow j(X,Z), r(Z,Y)$ (similarly for the other kinds of production rules).

Then, for any such word $R_{i_1} \dots R_{i_l}$ of $L(G)$, assume $r_{i_j}(u_j, u_{j+1}) \in EDB$, for $j = 1, \dots, l$, i.e. $(u_1, u_{l+1}) \in CQ_{L(G)}(EDB)$. Then, trivially there is a refutation for $\leftarrow r_{i_1}(X,Z_1), \dots, r_{i_l}(Z_{l-1},Y)$ in EDB (and therefore in $EDB \cup IDB(G)$) returning the substitution $\{X/u_1, Z_1/u_2, \dots, Z_{l-1}/u_l, Y/u_{l+1}\}$, i.e. $(u_1, u_{l+1}) \in Q_i^{IDB(G)}(EDB)$.

Conversely, for any derivation from $\leftarrow i(X,Y)$ to $\leftarrow r_{i_1}(X,Z_1), \dots, r_{i_l}(Z_{l-1},Y)$,

assume there is a refutation for $\leftarrow r_{i_1}(X, Z_1), \ldots, r_{i_l}(Z_{l-1}, Y)$ in EDB (and therefore in $EDB \cup IDB(G)$) returning the substitution $\{X/u_1, Z_1/u_2, \ldots Z_{l-1}/u_l, Y/u_{l+1}\}$, i.e. $(u_1, u_{l+1}) \in Q_i^{IDB(G)}(EDB)$.
Then, trivially $r_{i_j}(u_j, u_{j+1}) \in EDB$, for $j = 1, \ldots, l$, namely $(u_1, u_{l+1}) \in CQ_{L(G)}(EDB)$. $\square$

Trivially, for every regular grammar G, the corresponding $IDB(G)$ is linear and therefore linearisable. As a consequence, the chain queries for regular languages are linearisable.

4.2 Pseudo-Regular Chain Queries

We identify a class of languages, containing all regular languages, such that all chain queries for languages in such class are linearisable. This is the class of all *pseudo-regular languages*, of the form:

$\{\alpha_1^{k_1} \ldots \alpha_n^{k_n} |$ for each $j = 1, \ldots, n$, either k_j is an index and $k_j \geq 0$
or k_j is a positive natural number $\}$

with $\alpha_j \in \Sigma^+$, $j = 1, \ldots, n$, and $n > 0$.

We will refer to the chain queries for such languages as *pseudo-regular (chain) queries.*

Note that every regular language is trivially pseudo-regular.

We prove that pseudo-regular queries are linearisable by constructing the corresponding Datalog programs, show that they are piecewise linear and therefore linearisable, by the results in section 3. We first illustrate the construction by means of examples.

Example 32 *Let* $L = \{R_1^{i_1} R_2^{i_1} R_3^{i_1} | i_1 \geq 0\}$ *(with* $\Sigma = \{R_1, R_2, R_3\}$*). The Datalog program corresponding to the query for L is:*

$$i_0(X,Y) \leftarrow i_1(X,Z,Z,W,W,Y).$$

$$i_1(X,Z,W,U,V,Y) \leftarrow$$

$$i_{j,1}(X,X_1), i_{j,2}(W,W_1), i_{j,3}(V,V_1), i_1(X_1,Z,W_1,U,V_1,Y).$$

$$i_1(X,X,W,W,Y,Y).$$

$$i_{j,1}(X,X_1) \leftarrow r_1(X,X_1).$$

$$i_{j,2}(W,W_1) \leftarrow r_2(W,W_1).$$

$$i_{j,3}(V,V_1) \leftarrow r_3(V,V_1).$$

This program is not linear but is piecewise linear and therefore linearisable (see section 3). Note that in this example linearisation can be achieved simply by unfolding the predicates $i_{j,e}$, for $e = 1,2,3$:

$$i_0(X,Y) \leftarrow i_1(X,Z,W,U,V,Y).$$

$$i_1(X,Z,W,U,V,Y) \leftarrow r_1(X,X_1), r_2(W,W_1), r_3(V,V_1),$$

$$i_1(X_1,Z,W_1,U,V_1,Y).$$

$$i_1(X,X,W,W,Y,Y).$$

Example 33 *Let $L = \{R_1^{i_1} R_2^{i_2} R_3^{i_1} R_4^2 | i_1, i_2 \geq 0\}$ (with $\Sigma = \{R_1, R_2, R_3, R_4\}$). The Datalog program corresponding to the query for L is:*

$$i_0(X,Y) \leftarrow i_1(X,Z,W,U), i_2(Z,W), i_3(U,V), i_4(V,Y).$$

$$i_1(X,Z,W,Y) \leftarrow i_{1,1}(X,X_1), i_{1,2}(W,W_1), i_1(X_1,Z,W_1,Y).$$

$$i_1(X,X,Y,Y).$$

$$i_{1,1}(X,X_1) \leftarrow r_1(X,X_1).$$

$$i_{1,2}(W,W_1) \leftarrow r_3(W,W_1).$$

$$i_2(Z,W) \leftarrow i_{2,1}(Z,Z_1), i_2(Z_1,W).$$

$$i_2(Z,Z).$$

$$i_{2,1}(Z,Z_1) \leftarrow r_2(Z,Z_1).$$

$$i_3(U,V) \leftarrow i_{3,1}(U,V).$$

$$i_4(V,Y) \leftarrow i_{4,1}(V,Y).$$

$$i_{3,1}(U,V) \leftarrow r_4(U,V).$$

$$i_{4,1}(V,Y) \leftarrow r_4(V,Y).$$

This program is not linear but is piecewise linear and therefore linearisable (see section 3). Note that in this example it is not sufficient just unfolding the predicates $i_{j,e}$, for $j = 1,2,3$, in order to achieve linearisation. Indeed, the result of such unfolding is:

$$i_0(X,Y) \leftarrow i_1(X,Z,W,U), i_2(Z,W), i_3(U,V), i_4(V,Y).$$

$$i_1(X,Z,W,Y) \leftarrow r_1(X,X_1), r_3(W,W_1), i_1(X_1,Z,W_1,Y).$$

$$i_1(X,X,Y,Y).$$

$$i_2(Z,W) \leftarrow r_2(Z,Z_1), i_2(Z_1,W).$$

$$i_2(Z,Z).$$

$$i_3(U,V) \leftarrow r_4(U,V).$$

$$i_4(V,Y) \leftarrow r_4(V,Y).$$

Example 34 *Let $L = \{(R_1R_2R_3)^{i_1} R_4^{i_2} R_5^{i_1} | i_1, i_2 \geq 0\}$ (with $\Sigma = \{R_1, R_2, R_3, R_4, R_5\}$). The Datalog program corresponding to the query for L is:*

$$i_0(X,Y) \leftarrow i_1(X,Z,W,Y), i_2(Z,W).$$

$$i_1(X,Z,W,Y) \leftarrow i_{1,1}(X,X_1), i_{1,2}(W,W_1), i_1(X_1,Z,W_1,Y).$$

$$i_1(X,X,Y,Y).$$

$$i_{1,1}(X, X_1) \leftarrow r_1(X, X_2), r_2(X_2, X_3), r_3(X_3, X_1).$$

$$i_{1,2}(W, W_1) \leftarrow r_5(W, W_1).$$

$$i_2(Z, W) \leftarrow i_{2,1}(Z, Z_1), i_2(Z_1, W).$$

$$i_2(Z, Z).$$

$$i_{2,1}(Z, Z_1) \leftarrow r_4(Z, Z_1).$$

This program is not linear but is piecewise linear and therefore linearisable (see section 3).

Before we define the general technique for mapping pseudo-regular chain queries onto Datalog programs, note that each pseudo-regular language can be equivalently rewritten in such a way that every integer exponent is 1. For example, the language in example 33 can be rewritten as $\{R_1^{i_1} R_2^{i_2} R_3^{i_1} R_4 R_4 | i_1, i_2 \geq 0\}$. In the sequel, we will assume such rewriting of pseudo-regular languages.

Definition 35 *The* Datalog program $IDB(L)$ corresponding to a pseudo-regular query *for a language*

$$\{\alpha_1^{k_1} \ldots \alpha_n^{k_n} |\ \textit{for each } j = 1, \ldots, n, \textit{ either } k_j \textit{ is an index and } k_j \geq 0 \quad \text{OR } k_j \text{ IS } 1\ \}$$

for some $n > 0$ and $\alpha_j \in \Sigma^+$, $j = 1, \ldots, n$, is constructed as follows. Let:

- *m be the number of integer and distinct indexes amongst $k_1, \ldots, k_n$ (trivially, $m \leq n$), and, after renaming them (for ease of reference) as $i_1, \ldots, i_m$, according to the order in which they appear, let*
- *a_j be the number of factors with (the integer or index) i_j as exponent, $j = 1, \ldots, m$.* [5]

For ease of reference, let us

[5] 'Note that, if i_j is 1 then $a_j = 1$.

– *rename the bases of the* a_j *factors of exponent* i_j *(*$j = 1, \ldots, m$*)* *as* $\alpha_{j,1}, \ldots, \alpha_{j,a_j}$, *and*
– *assume each* $\alpha_{j,e}$ *(*$j = 1, \ldots, m$, $e = 1, \ldots, a_j$*)* [6] *be* $R_{j,e,1}, \ldots, R_{j,e,i_{j,e}}$, *for some* $i_{j,e} \geq 1$ *(given in the definition of* L*).*

Then, $IDB(L)$ *has* $m+1$ *intensional predicates,* $i_0, i_1, \ldots, i_m$*, with arity 2,* $2a_1, \ldots, 2a_m$*, respectively, and, for each* $j = 1, \ldots, m$, a_j *intensional predicates,* $i_{j,1}, \ldots, i_{j,a_j}$*, each with arity 2, defined by*

$$i_0(var_1(\alpha_1), var_2(\alpha_n)) \leftarrow \\ \quad i_1(vars_1), \ldots, i_m(vars_m)$$

by clauses (for $j = 1, \ldots, m$*)*

$$i_j(var_1(\alpha_{j,1}), var_2(\alpha_{j,1}), \ldots, var_1(\alpha_{j,a_j}), var_2(\alpha_{j,a_j})) \leftarrow \\ \quad i_{j,1}(var_1(\alpha_{j,1}), X^{j,1}), \ldots, i_{j,a_j}(var_1(\alpha_{j,a_j}), X^{j,a_j}), \\ \quad i_j(X^{j,1}, var_2(\alpha_{j,1}), \ldots, X^{j,a_j}, var_2(\alpha_{j,a_j})) \\ i_j(var_1(\alpha_{j,1}), var_2(\alpha_{j,1}), \ldots, var_1(\alpha_{j,a_j}), var_2(\alpha_{j,a_j})) \leftarrow \\ \quad var_1(\alpha_{j,1}) = var_2(\alpha_{j,1}), \ldots, var_1(\alpha_{j,a_j}) = var_2(\alpha_{j,a_j})$$

and by clauses (for $j = 1, \ldots, m$, $e = 1, \ldots, a_j$*)*

$$i_{j,e}(var_1(\alpha_{j,e}), X^{j,e}) \leftarrow \\ \quad r_{j,e,1}(var_1(\alpha_{j,e}), X_1), r_{j,e,2}(X_1, X_2), \ldots, r_{j,e,i_{j,1}}(X_{i_{j,e}-1}, X^{j,e})$$

where X_i *are fresh, distinct variables, each* $r_{j,e,i}$ *is an (extensional) predicate symbol corresponding to the letter* $R_{j,e,i}$*, and*

– $var_1(\alpha_i)$, $var_2(\alpha_i)$ *be (distinct) variables associated to the factor with base* α_i*, for* $i = 1, \ldots, n$ *(we use a functional representation of variables for ease of reference), such that, for* $i = 1, \ldots, n-1$, $var_2(\alpha_i) = var_1(\alpha_{i+1})$*, and*
– $vars_j = var_1(\alpha_{j,1}), var_2(\alpha_{j,1}), \ldots, var_1(\alpha_{j,a_j}), var_2(\alpha_{j,a_j})$*, for* $j = 1, \ldots, m$*.* [7]

---

[6] Note that, if i_j is 1 then $e = 1$.

[7] Note that, if i_j is 1, then $vars_j = var_1(\alpha_{j,1}), var_2(\alpha_{j,1})$.

In example 32, there is one distinct index, i_1, and thus two intensional predicates, i_0 and i_1. The predicate i_1 has arity 6, since $a_1 = 3$, as there are three factors ($R_1^{i_1}$, $R_2^{i_1}$ and $R_3^{i_1}$) with the index i_1 as exponent. Since $a_1 = 3$, there are three additional (binary) intensional predicates $i_{1,1}, i_{1,2}, i_{1,3}$.

In example 33, there are two distinct indexes, i_1 and i_2, and an integer exponent, 2 (expressible via two integer exponents, 1), and thus five intensional predicates, i_0, i_1, i_2, i_3 and i_4. The predicate i_1 has arity 4, since $a_1 = 2$, as there are two factors ($R_1^{i_1}$ and $R_3^{i_1}$) with the index i_1 as exponent. The predicate i_2 has arity 2, since $a_2 = 1$, as there is only one factor ($R_2^{i_2}$) with the index i_2 as exponent. The predicates i_3 and i_4, corresponding to the integer exponents, have arity 2, since $a_3 = a_4 = 1$. [8] Since $a_1 = 2$, there are two additional (binary) intensional predicates $i_{1,1}, i_{1,2}$. Since $a_2 = 1$, there is one additional (binary) intensional predicate $i_{2,1}$. Since $a_3 = 1$, there is one additional (binary) intensional predicate $i_{3,1}$. Since $a_4 = 1$, there is one additional (binary) intensional predicate $i_{4,1}$.

Similarly in example 34.

Moreover, in example 32, $\alpha_{1,1} = \alpha_1 = R_1$, $\alpha_{1,2} = \alpha_2 = R_2$ and $\alpha_{1,3} = \alpha_3 = R_3$ (all corresponding to exponent i_1).

In example 33, $\alpha_{1,1} = \alpha_1 = R_1$ and $\alpha_{1,2} = \alpha_3 = R_3$ (both corresponding to exponent i_1), $\alpha_{2,1} = \alpha_2 = R_2$ (corresponding to exponent i_2), $\alpha_{3,1} = \alpha_4 = R_4$ (corresponding to integer exponent 1), and $\alpha_{4,1} = \alpha_5 = R_4$ (corresponding to integer exponent 1).

In example 34, $\alpha_{1,1} = \alpha_1 = R_1 R_2 R_3$ and $\alpha_{1,2} = \alpha_3 = R_5$ (both corresponding to exponent i_1), and $\alpha_{2,1} = \alpha_2 = R_4$ (corresponding to exponent i_2). Moreover, $R_{1,1,1} = R_1$, $R_{1,1,2} = R_2$, $R_{1,1,3} = R_3$ (for $\alpha_{1,1}$), $R_{1,2,1} = R_5$ (for $\alpha_{1,2}$), and $R_{2,1,1} = R_4$ (for $\alpha_{2,1}$).

Finally, in example 34, $var_1(\alpha_1) = X$, $var_2(\alpha_1) = Z = var_1(\alpha_2)$, $var_2(\alpha_2) = W = var_1(\alpha_3)$ and $var_2(\alpha_3) = Y$.

[8] Note that the arity of intensional predicates corresponding to integer exponents is always 2, since each factor with integer exponent is considered separately.

The query for a pseudo-regular language L coincides with the query corresponding to the predicate i_0 in $IDB(L)$:

Theorem 36 *Given a pseudo-regular language L, for every extensional database EDB for the extensional predicates in $IDB(G)$:*

$CQ_L(EDB) = Q_{i_0}^{IDB(L)}(EDB)$.

The proof of this theorem is analogous to the proof of theorem 31 but is fully given here for completeness of presentation.

Proof. By definition, $Q_{i_0}^{IDB(L)}(EDB) = \{(u,v) | i_0(u,v)$ belongs to the least Herbrand model of $EDB \cup IDB(L)\}$. Since SLD resolution is complete with respect to the least Herbrand model semantics [31], $Q_{i_0}^{IDB(L)}(EDB) = \{(u,v)|$ there is an SLD refutation for $i_0(u,v)$ in $EDB \cup IDB(L)\}$.

It is not difficult to see that there is a one-to-one correspondence between words $\alpha_1^{k_1} \dots \alpha_n^{k_n}$, for concrete values of $k_1, \dots, k_n$, of L and SLD derivations, in $IDB(L)$ (and therefore in $EDB \cup IDB(L)$), from goals $\leftarrow i_0(X,Y)$ to goals (assume each $\alpha_j = R_{j,1} \dots R_{j,l_j}$, for some l_j, given in the definition of L)

$$
\begin{array}{l}
\leftarrow r_{1,1}(X, X_1^{1,1}), r_{1,2}(X_1^{1,1}, X_2^{1,1}), \dots, r_{1,l_1}(X_{l_1-1}^{1,1}, X_{l_1}^{1,1}), \\
\quad r_{1,1}(X_{l_1}^{1,1}, X_1^{1,2}), \dots \quad \dots \quad \dots \quad \dots, r_{1,l_1}(X_{l_1-1}^{1,2}, X_{l_1}^{1,2}), \\
\quad \dots, \\
\quad r_{1,1}(X_{l_1}^{1,k_1-1}, X_1^{1,k_1}), \dots \quad \dots \quad \dots, r_{1,l_1}(X_{l_1-1}^{1,k_1}, X_{l_1}^{1,k_1}), \\
\quad r_{2,1}(X_{l_1}^{1,k_1}, X_1^{2,1}), \dots \quad \dots \quad \dots \quad \dots, r_{2,l_2}(X_{l_2-1}^{2,1}, X_{l_2}^{2,1}), \\
\quad \dots, \\
\quad r_{2,1}(X_{l_2}^{2,k_2-1}, X_1^{2,k_2}), \dots \quad \dots \quad \dots, r_{2,l_2}(X_{l_2-1}^{2,k_2}, X_{l_2}^{2,k_2}), \\
\quad \dots, \\
\quad r_{n,1}(X_{l_{n-1}}^{n-1,k_{n-1}}, X_1^{n,1}), \dots \quad \dots \quad \dots, r_{n,l_n}(X_{l_n-1}^{n,1}, X_{l_n}^{n,1}), \\
\quad \dots, \\
\quad r_{n,1}(X_{l_n}^{n,k_n-1}, X_1^{n,k_n}), \dots \quad \dots \quad \dots, r_{n,l_n}(X_{l_n-1}^{n,k_n}, Y)
\end{array}
$$

for some distinct variables $X_1^{1,1}, \dots, X_{l_n-1}^{n,k_n}$ (in turn distinct from X and Y). We will refer to any such goal as $\leftarrow$ $goal(\alpha_1^{k_1} \dots \alpha_n^{k_n})$. If $k_j = 0$, for some $j = 1, \dots, n$, then the conjuncts corresponding to α_j (i.e. the conjuncts in

the predicates $r_{j,i}$, for $i = 1,\ldots,l_j$) are missing and $X_{l_j}^{j,k_j} = X_{l_{j-1}}^{j-1,k_{j-1}}$. If $k_1 = \ldots = k_n = 0$ then $\leftarrow goal(\alpha_1^{k_1} \ldots \alpha_n^{k_n})$ is $\leftarrow X = Y$.

Then, for any such word $\alpha_1^{k_1} \ldots \alpha_n^{k_n}$ ($\alpha_j = R_{j,1} \ldots R_{j,l_j}$, for some l_j, given in the definition of L), assume $(u,v) \in CQ_L(EDB)$. Then, trivially there is a refutation for $\leftarrow goal(\alpha_1^{k_1} \ldots \alpha_n^{k_n})$ in EDB (and therefore in $EDB \cup IDB(L)$) returning the substitution $\{X/u, Y/v\}$, i.e. $(u,v) \in Q_{i_0}^{IDB(L)}(EDB)$.

Conversely, for any derivation from $\leftarrow i_0(X,Y)$ to $\leftarrow goal(\alpha_1^{k_1} \ldots \alpha_n^{k_n})$, assume there is a refutation for $\leftarrow goal(\alpha_1^{k_1} \ldots \alpha_n^{k_n})$ in EDB (and therefore in $EDB \cup IDB(L)$) returning the substitution $\{X/u, Y/v\}$, i.e. $(u,v) \in Q_{i_0}^{IDB(L)}(EDB)$. Then, trivially $(u,v) \in CQ_L(EDB)$. $\square$

Lemma 37 *The Datalog program corresponding to any pseudo-regular chain query is piecewise linear.*

Proof. Let L be a pseudo-regular language and $IDB(L)$ be the corresponding Datalog program. Let $i_0, i_1, \ldots, i_m$, and, for each $j = 1,\ldots,m$, let $i_{j,1}, \ldots, i_{j,a_j}$ be the intensional predicates defined in $IDB(L)$. The clauses defining each i_{j,l_j}, $j = 1,\ldots,m$ and $l = 1,\ldots,a_j$, are all linear. Then, the only (potentially) non-linear clauses are those defining $i_0, i_1, \ldots, i_m$. The clause defining i_0 is (in general) non-linear but none of the intensional atoms in its body (whose predicates are $i_1, \ldots, i_m$) is mutually recursive with i_0. Finally, in the body of each clause defining a predicate i_j, for $j = 1,\ldots,m$, at most one atom is mutually recursive with the head predicate of the clause. Therefore, $IDB(L)$ is piecewise linear. $\square$

Therefore, the Datalog program corresponding to any pseudo-regular chain query is linearisable (see section 3).

5 (Non-Linearisable) Context-Free Chain Queries

Linearisability is not a property that many Datalog programs have. In fact, there are "simple" Datalog programs that are not linearisable. To support this claim we review some negative results from the literature.

Chain queries for *context-free* languages are referred to as *context-free chain queries.*

It is easy to map a context-free grammar G (generating a context-free language $L(G)$) in a natural way onto a Datalog program computing the chain query $CQ_{L(G)}$. We illustrate this mapping by an example:

Example 38 *Let G be the context-free grammar with production rules*

$I \rightarrow R_1 I R_2 I \mid \epsilon,$

initial (non-terminal) symbol I and terminal symbols R_1, R_2. Then $CQ_{L(G)}$ is computed by the Datalog program:

$$I(X,Y) \leftarrow R_1(X,Z_1), I(Z_1,Z_2), R_2(Z_2,Z_3), I(Z_3,Y).$$

$$I(X,X).$$

Datalog programs as above, i.e. programs with one initializing clause and one recursive clause, are called *elementary chain programs* [30]. All context-free chain queries can be mapped onto elementary chain programs.

It is well known that regular languages are context-free, but there exist context-free languages which are not regular. In addition, note that context-free languages might not be pseudo-regular, e.g. the languages $\{W|W$ has the same number of occurrences of R_1 and $R_2\}$ is context-free but not pseudo-regular. Moreover, pseudo-regular languages might not be context-free, e.g. see the language in example 32. However, some pseudo-regular languages are context-free, e.g. see the languages in examples 33 and 34.

Trivially from the results in the previous section 4, all context-free chain queries that are (pseudo-)regular are linearisable. However, there are chain

queries for context-free languages which are not linearisable, as proven in the literature.

Theorem 39 *[1] Let $L^0 \subseteq \{R_1, R_2\}^*$ be the context-free language:*

$L^0 = \{\rho | \rho$ has the same number of occurrences of R_1 and $R_2\}$.

Then, the chain query CQ_{L^0} is not linearisable.

Theorem 40 *[1] If L is generated by one of the context-free grammars below, then the chain query CQ_L is not linearisable:*

a) $I \rightarrow IR_1I(R_2IR_1I)^j \mid \epsilon$, where $j \geq 1$.

b) $I \rightarrow (IR_1)^iIR_2I(R_1I)^j \mid \epsilon$, where $i, j \geq 1$.

6 Conclusions and Future Work

We have investigated linearisability of Datalog programs:

a) We have shown constructively that any piecewise linear logic program into an equivalent linear program, by giving a procedure that performs the transformation.

b) We have defined pseudo-regular chain queries and have shown how to express them by means of Datalog programs that are piecewise linear, hence, by a), linearisable.

The procedure for piecewise linear programs relies heavily upon formal logic program transformation techniques known to preserve the meaning of programs. Correctness of the procedure is a direct consequence of the meaning-preserving nature of the transformation techniques. Thus, the results presented in this paper are also interesting in view of the fact that they attack the problem of applying program transformation techniques, when defining a priori the subclass of programs to which they are going to be applied.

Linearisability of Datalog programs/queries has been studied elsewhere in the literature. For example, [14] gives conditions for the linerisability of bilinear Datalog programs, i.e. non-linear programs with at most two intensional predicates in the body of each clause. Since each non-linear Datalog program can be equivalently expressed via a bilinear program [14], these results apply generally. [23,33] give necessary and sufficient conditions for linerisability, which are instances of the ones given in [14]. Rather than looking at generic non-linear queries, we have considered the special class of piecewise linear Datalog programs and some chain queries, interesting despite their simplicity [1,4].

We are currently working on defining a class of languages with the property of being *exactly* the class for which chain queries are linearisable. We believe that it is wider than the class of pseudo-regular languages: in fact we have made some progress in figuring out that this class can be defined via a special kind of automata that use a constant number of stacks and queues in a specific fashion.

Acknowledgement

The authors wish to thank Stavros Cosmadakis for helpful discussions and the anonymous reviewers for their insightful suggestions.

References

[1] F. Afrati and S. Cosmadakis. Expressiveness of restricted recursive queries. In *Proc. 21st ACM Symp. on Theory of Computing*, pages 113–126, 1989.

[2] F. Afrati, M. Gergatsoulis, and M. Katzouraki. On transformations into linear database logic programs. In D. Bjørner, M. Broy, and I. Pottosin, editors, *Perspectives of Systems Informatics (PSI'96), Proceedings*, Lecture Notes in Computer Science (LNCS) 1181, pages 433–444. Springer-Verlag, 1996.

[3] F. Afrati and C. H. Papadimitriou. The parallel complexity of simple chain queries. In *Proc. 6th ACM Symposium on Principles of Database Systems*,

pages 210–213. ACM Press, 1987.

[4] F. Afrati and C. H. Papadimitriou. Parallel complexity of simple logic programs. *Journal of the ACM*, 40(3):891–916, 1993.

[5] F. Afrati and F. Toni. Chain queries expressible by linear datalog programs. In U. Geske, C. Ruiz, and D. Seipel, editors, *Proc. of the 5th International Workshop on Deductive Databases and Logic Programming (DDLP'97)*, pages 49–58, 1997. GMD-studien Nr 317, Sankt Augustin.

[6] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. ACM Conf. on Management of Data*, pages 16–52, 1986.

[7] C. Beeri, P. C. Kanellakis, F. Bancilhon, and R. Ramakrishnan. Bounds on the propagation of selection into logic programs. In *Proc. 6th ACM Symp. on Principles of Database Systems*, pages 214–226. ACM Press, 1987.

[8] A. Bossi and N. Cocco. Basic transformation operations which preserve computed answer substitutions of logic programs. *The Journal of Logic Programming*, 16(1 & 2):47–87, May 1993.

[9] S. A. Cook. An observation on time-storage trade off. *Journal of Computer and System Sciences*, 9:308–316, 1974.

[10] S. S. Cosmadakis, H. Gaifman, P. C. Kanellakis, and M. Y. Vardi. Decidable optimization problems for database logic programs. In *Proc. 20th ACM Symp. on Theory of Computing*, pages 477–490, 1988.

[11] S. S. Cosmadakis and P. C. Kanellakis. Parallel evaluation of recursive rule queries. In *Proc. 5th ACM Symp. on Principles of Database Systems*, pages 280–293. ACM Press, 1986.

[12] M. Gergatsoulis. *Logic program transformations: Transformation rules and application strategies*. PhD thesis, Dept. of Computer Science, University of Athens, 1994. (In Greek).

[13] M. Gergatsoulis and M. Katzouraki. Unfold/fold transformations for definite clause programs. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming (PLILP'94), Proceedings*, Lecture Notes in Computer Science (LNCS) 844, pages 340–354. Springer-Verlag, 1994.

[14] Y. E. Ioannidis and E. Wong. Towards an algebraic theory of recursion. *Journal of the ACM*, 38(2):329–381, 1991.

[15] T. Kawamura and T. Kanamori. Preservation of stronger equivalence in unfold/fold logic program transformations. *Theoretical Computer Science*, 75:139–156, 1990.

[16] J-L. Lasser, M. J. Maher, and K. Marriott. Unification revisited. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann Publishers,Inc., 1988.

[17] J. F. Naughton. Data independent recursion in deductive databases. In *Proc. 5th ACM Symp. on Principles of Database Systems*, pages 267–279. ACM Press, 1986.

[18] J. F. Naughton and Y. Sagiv. A decidable class of bounded recursions. In *Proc. 6th ACM Symp. on Principles of Database Systems*, pages 227–236. ACM Press, 1987.

[19] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[20] A. Pettorossi and M. Proietti. Transformation of logic programs : Foundations and techniques. *The Journal of Logic Programming*, 19/20:261–320, May/July 1994.

[21] M. Proietti and A. Pettorossi. Synthesis of Eureka predicates for developing logic programs. In *Proc. of the 3rd European Symposium on Programming*, Lecture Notes in Computer Science (LNCS) 432, pages 306–325. Springer-Verlag, 1990.

[22] M. Proietti and A. Pettorossi. The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *The Journal of Logic Programming*, 16(1 & 2):123–162, May 1993.

[23] Y. P. Saraiya. Linearising non-linear recursion in polynomial time. In *Proc. 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 182–189. ACM Press, 1990.

[24] Y. P. Saraiya. On the efficiency of transforming database logic programs. *Journal of Computer and System Sciences*, 51(1):87–109, 1995.

[25] H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.

[26] H. Seki. Unfold/fold transformations for general logic programs for the well-founded semantics. *The Journal of Logic Programming*, 16(1 & 2):5–23, May 1993.

[27] H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In Sten-Åke Tarnlund, editor, *Proc. of the Second International Conference on Logic Programming*, pages 127–138, 1984.

[28] H. Tamaki and T. Sato. A generalized correctness proof of the unfold/fold logic program transformation. Technical Report No 86-4, Dept. of Information Science, Ibaraki University, Japan, 1986.

[29] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I & II. Computer Science Press, 1989.

[30] Jeffrey D. Ullman and Allen Van Gelder. Parallel complexity of logical query programs. In *Proc. 27th IEEE Symp. on Foundations of Comp. Sci.*, pages 438–454, 1986.

[31] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, Oct. 1976.

[32] M. Y. Vardi. Decidability and undecidablity results for boundedness of linear recursive queries. In *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 341–351. ACM Press, 1988.

[33] W. Zang, C. T. Yu, and D. Troy. A necessary and sufficient condition to linearise doubly recursive programs in logic databases. *ACM Transaction of Database Systems*, 15(3):459–482, 1990.