Datalog programs and their persistency numbers

FOTO AFRATI
National Technical University of Athens
STAVROS COSMADAKIS
University of Patras
and
EUGENIE FOUSTOUCOS
National and Capodistrian University of Athens

The relation between Datalog programs and homomorphism problems and, between Datalog programs and bounded treewidth structures has been recognized for some time and given much attention recently. Additionally, the essential role of persistent variables (of program expansions) in solving several relevant problems has also started to be observed. It turns out that to understand the contribution of these persistent variables to the difficulty of some expressibility problems, we need to understand the interrelationship among different notions of *persistency numbers* some of which we introduce and/or formalize in the present work.

This paper is a first foundational study of the various persistency numbers and their interrelationships. To prove the relations among these persistency numbers we had to develop some non trivial technical tools that promise to help in proving other interesting results too. More precisely, we define the *adorned dependency graph of a program*, a useful tool for visualizing sets of persistent variables, and we define automata that recognize persistent sets in expansions.

We start by elaborating on finer definitions of expansions and queries, which capture aspects of homomorphism problems on bounded treewidth structures. The main results of this paper are: a) a program transformation technique, based on automata-theoretic tools, which manipulates persistent variables (leading, in certain cases, to programs of fewer persistent variables), b) a categorization of the different roles of persistent variables; this is done by defining four notions of persistency numbers which capture the propagation of persistent variables from a syntactical level to a semantical one, c) decidability results concerning the syntactical notions of persistency numbers that we have defined and d) exhibition of new classes of programs for which boundedness is undecidable.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages—*Query lan-*

Authors' addresses:
Foto N. Afrati: National Technical University of Athens, 15773 Zografou, Athens, Greece. Email: afrati@cs.ece.ntua.gr.
Stavros Cosmadakis: University of Patras, 26500 Rio, Patras, Greece. Email: scosmada@cti.gr.
Eugénie Foustoucos (Corresponding Author), MPLA, Department of Mathematics, National and Capodistrian University of Athens, Panepistimiopolis, 15784 Athens, Greece. Email: aflaw@otenet.gr, eugenie@eudoxos.math.uoa.gr. Research supported by a Fellowship of the Greek Scholarship Foundation (IKY) and by MPLA (Graduate Program in Logic, Algorithms and Computation), National and Capodistrian University of Athens.

guages; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*Automata*; *Computability Theory*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*Decision Problems*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Hypergraphs*; D.1.6 [**Software**]: Programming Techniques—*Logic Programming*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Boundedness, bounded-tree width hypergraphs, Datalog, finite automata, persistency numbers, persistent variables, program transformations

1. INTRODUCTION

Datalog programs have been investigated extensively in the last two decades and several authors have observed that in order to derive stronger results a more thorough and fine investigation of their structure is needed [CGKV88; AC89; Af97]. Specifically it has been observed that query evaluation and optimization techniques, expressive power and decidability of certain properties may depend on the arity of recursive predicates and moreover on the number of occurrences of persistent variables (variables that appear in the head and in recursive predicates in the body of a rule). To mention a specific case, the boundedness problem is proven undecidable in the general case [GMSV87] whereas when the arity of the recursive predicates is equal to one it becomes decidable [CGKV88]. It is known that there exists a hierarchy of Datalog programs w.r.t. their arity [AC89; Gro94]; programs of larger arity are stricly more expressive. There exists also such a hierarchy w.r.t. their "persistency number" [Af97]; the definition of this number involves semantical notions and not only syntactic [Af97]. The concept of persistencies is implicit in [CGKV88; Var88] and a syntactical definition of persistency equal to zero is given in [C89]. Studying the persistencies leads us to a better understanding of the role that the syntactic parameters of Datalog programs play in the undecidability of boundedness; indeed the results we obtain here explicate a possible trade-off between the number of recursive rules and the number of persistencies. Another important issue related to the study of persistencies is the following: it is well-known that a Datalog program of size n, with IDB predicates of arity at most k, can be evaluated in time $O(nk)$; a substantial improvement to this upper bound seems unlikely, on complexity-theoretic grounds; on the other hand, proving a corresponding lower bound in the general case would be a major breakthrough, as it would imply separation of complexity classes [GSS01; PY97]. It therefore makes sense to look for lower bounds in restricted models of computation (see for instance [C02]). We expect our study of the persistency number will be relevant here, as the natural way to write Datalog programs requiring $\Omega(nk)$ time seems to require high persistency number ([Af97] has some typical examples).

In order to give some motivation of our research, we consider the following examples:

1) Let π_1 be the following program:
$r_1^1: \ T(x,y) \leftarrow E(x,y)$
$r_2^1: \ T(x,y) \leftarrow T(z,x), E(y,x), E(y,z).$
Clearly π_1 has one persistent variable, namely x in the second rule. It is easy to see, though, that this persistent variable disappears after unfolding the second rule.

Therefore, we can use this observation to construct an equivalent program with no persistent variables. The program π_2 that follows is such a program:
$r_1^2: T(x,y) \leftarrow E(x,y)$
$r_2^2: T(x,y) \leftarrow E(z,x), E(y,x), E(y,z)$
$r_3^2: T(x,y) \leftarrow T(z',z), E(x,z), E(x,z'), E(y,x), E(y,z).$

2) Now, we consider another example which illustrates a different situation. Consider the following program π which computes the transitive closure.
$r_1: T(x,y) \leftarrow E(x,y)$
$r_2: T(x,y) \leftarrow E(x,z), T(z,y).$
It has also one persistent variable, but this time it is not so easy to get rid of. But, still, there exists an equivalent program with no persistent variables, the following program π': $r_1': T(x,y) \leftarrow E(x,y)$
$r_2': T(x,y) \leftarrow E(x,z), E(z,y).$
$r_3': T(x,y) \leftarrow E(x,z_1), T(z_1,z_2), E(z_2,y).$

3) The third example depicts an intermediate situation. Consider the program π'' which also computes the transitive closure:
$r_1: T(x,y) \leftarrow E(x,y)$
$r_2: T(x,y) \leftarrow E(x,z), T(z,y).$
$r_2': T(x,y) \leftarrow E(x,z), E(z,y).$
$r_3': T(x,y) \leftarrow E(x,z_1), T(z_1,z_2), E(z_2,y).$
It has also one persistent variable but, unlike program π in example 2, it can be "transformed" into an equivalent program with no persistent variables by simply deleting rule r_2.

The first example shows that a persistent variable in the syntax of the program may not propagate after unfolding the rules a few times. This leads to an equivalent program with no persistent variables in the syntax. We say that program π_1 of example 1 has *syntactic persistency number* equal to 1 and *weak persistency number* equal to 0; informally, the weak persistency number of a program is defined by considering all its expansions and finding the maximum number of variables that propagate ad infinitum simultaneously. For program π_2 both numbers are equal to 0. In section 4, we give an algorithm which on input a program of weak persistency number m yields in the output an equivalent program of syntactic persistency number m. We prove that finding the weak persistency number of a given program is decidable.

In the second example, program π has both syntactic and weak persistency number equal to 1. However, there exists an equivalent program, namely π', with both syntactic and weak persistency number equal to 0. We say that program π has *persistency number-modulo equivalence* equal to 0.

In the third example, program π'' contains some of the ingredients that allow one to transform a program into an equivalent one; we define formally which are those ingredients by considering expansions of the program and their "usefulness". A Datalog program can be equivalently thought of as the logical disjunction of its expansions. Now, the set of expansions of program π'' contains as subsets both the set of expansions of program π and the set of expansions of program π'. As π and π' are equivalent programs, it is natural that some of the expansions of π'' are not "useful" in that π'' can be **equivalently** thought of as the logical disjunction

of only a proper subset of its expansions. We call this subset a *useful family* of expansions and define the *persistency number* of a program with respect to such a subset. Informally, the persistency number of a program is defined by viewing a useful family of expansions and finding the maximum number of variables that propagate ad infinitum simultaneously. To formalize, we need to define persistency numbers for a set of bounded tree-width structures. In this case, we say that program π'' has syntactic and weak persistency number 1 and persistency number 0, whereas program π in example 2 has also persistency number equal to 1. We conjecture that there is no general algorithm that computes the persistency number of a program.

We give also an algorithm which on input a program of persistency number m and the number of unfoldings after which no more than m variables persist simultaneously, yields in the output an equivalent program of syntactic persistency number m. It remains an open problem whether we can compute a number of unfoldings such as the one needed in the input; if we can, then there will exist an algorithm to derive an equivalent program of syntactic persistency number m **if** it is given that the input program is of persistency number m.

To formalize we had to develop new technical tools. Specifically, we first defined an enhanced notion of dependency graph, the adorned dependency graph (namely the known dependency graph with more information on the labels of the edges). We define the several notions of persistency number on this graph.

Given a Datalog query Q, we define the syntactic, weak and persistency number of the query as the minimum, over all programs expressing the query, of their syntactic, weak and persistency number respectively. Thus, the following theorem summarizes the results in this paper:

Theorem Given a Datalog query Q, the following three statements are equivalent:
1. Query Q has syntactic persistency number equal to m.
2. Query Q has weak persistency number equal to m.
3. Query Q has persistency number equal to m.

The structure of the paper is the following. In section 2 we recall some basic preliminaries about Datalog and formalize expansions as bounded tree-width hypergraphs with persistencies. In section 3 we introduce the notion of useful families of expansions and characterize boundedness in terms of this notion. In section 4 we give definitions of different notions of persistency numbers and study the relationship between the two syntactical notions (namely the syntactical persistency number and the weak persistency number). In sections 5 and 6 we elaborate new tools (section 5) and automata-theoretic techniques (section 6) for dealing with persistencies and, using these tools and techniques, we study, in section 7, the interrelationship between the syntactical and the semantical notions of persistency numbers. In section 8 we give the decidability of the weak persistency number and we exhibit new classes of programs for which boundedness is undecidable.

2. PRELIMINARIES: DATALOG PROGRAMS, EXPANSIONS AS BOUNDED TREE-WIDTH HYPERGRAPHS WITH PERSISTENCIES

A *database* over domain D is a finite relational structure $\mathcal{D} = (D, r_1, \ldots, r_n)$, where D is a finite set and each r_i is a relation over D. The sequence $(a_1, ..., a_n)$ of arities of the r_i's is the *type* of the database. The database $\mathcal{D} = (D, r_1, \ldots, r_n)$ has signature $(R_1, ..., R_n)$ where for $i = 1, ..., n$ R_i is a predicate symbol of arity a_i, naming relation r_i.

A *Datalog program* is a collection of rules of the form $\iota_0 \leftarrow \iota_1, \ldots, \iota_\kappa$ where ι_0 is the *head*, and $\iota_1, \ldots, \iota_\kappa$ form the *body* of the rule. Each ι_i is an expression of the form $R(x_1, \ldots, x_m)$, where the x_i's are variables and R is a predicate symbol (we say predicate for short): R is either an *extensional database* (EDB) predicate naming one of the database relations, or an *intensional database* (IDB) predicate which is defined by the program; IDB predicates are exactly the ones appearing in the heads of rules. If R is an EDB (resp. IDB) predicate, then $R(x_1, \ldots, x_m)$ is an EDB (resp. IDB) *atom*. The set of EDB predicates is the *signature* of the input database. A *recursive* rule is a rule with at least one IDB predicate in its body; otherwise it is an *initialization* rule. We denote by $max(arity)_\pi$ the maximum arity of the IDB predicates of program π. The *dependency graph* of a program π is a directed graph with vertices the IDB predicates of π such that there exists an (unlabelled) edge from P to P' whenever π has a rule with head predicate P and with an IDB atom of predicate P' in its body. Given a Datalog program π and a goal predicate P we may, starting with an atom over P, unwind the recursive rules of π to some finite depth, to obtain a *P-expansion* (or simply *expansion*) of π.

Before giving the formal definition of expansions, we first need to define the notion of skeleton trees associated to a program. As we will see later, this notion - which expresses the unfolding of rules - is central in the tree-decomposition of expansions viewed as hypergraphs.

Definition 2.1. (**Skeleton Tree**) A tree $\mathcal{T}$ is a *skeleton tree* associated to program π if $\mathcal{T}$ satisfies the following: 1) its nodes are labeled with rules of π and 2) if a node N with label r has exactly n sons N_1,..., N_n with respective labels r_1,...,r_n, then rule r has exactly n occurrences $P_1(\vec{x}_1)$,...,$P_n(\vec{x}_n)$ of IDB atoms in its body and, for every $i = 1, ..., n$, P_i is the head predicate of rule r_i.
The root has depth 0 and a node N has depth $n + 1$ if its father has depth n. The *maximal depth* (or simply *depth*) of the tree is the maximal depth of its nodes. The *minimal IDB (resp. EDB) depth* of the skeleton tree is the minimal depth of its leaves that are labeled with recursive (resp. initialization) rules. The *size* of the tree is its number of nodes.

We now introduce a new formal definition of both expansions and partial expansions as 3-tuples of the form $(P(\vec{x}), \mathcal{D}, \mathcal{I})$: the first component is called the *head part*, the second component the *database part* and the third component the *tail part* (the tail part of expansions is always equal to the empty set). With each (partial) expansion e of π we associate a particular skeleton tree of π, called the skeleton tree of e. (Partial) expansions and their skeleton trees are defined inductively. We will use the following notation:

Notation We denote a recursive rule r by $r : P(\vec{x}) \leftarrow \mathcal{D}, \mathcal{I}$ where $\mathcal{D}$ (resp. $\mathcal{I}$)

is the set of EDB (resp. IDB) atoms of $body(r)$; we denote an initialization rule r by $r : P(\vec{x}) \leftarrow \mathcal{D}$. Conversely we say that the partial expansion $e = (P(\vec{x}), \mathcal{D}, \mathcal{I})$ defines the recursive rule $r : P(\vec{x}) \leftarrow \mathcal{D}, \mathcal{I}$, and that the expansion $e = (P(\vec{x}), \mathcal{D}, \emptyset)$ defines the initialization rule $r : P(\vec{x}) \leftarrow \mathcal{D}$. This will allow us to see (partial) expansions as rules and rules as (partial) expansions.

Definition 2.2. We give a simultaneously inductive definition of **(partial) expansions and their skeleton trees** (expansions are a special case of partial expansions).
Base case. If $r : P(\vec{x}) \leftarrow \mathcal{D}, \mathcal{I}$ is an instance of a recursive rule of program π then the 3-tuple $e : (P(\vec{x}), \mathcal{D}, \mathcal{I})$ is a *partial P-expansion* of π; if $r : P(\vec{x}) \leftarrow \mathcal{D}$ is an instance of an initialization rule of π then the 3-tuple $e : (P(\vec{x}), \mathcal{D}, \emptyset)$ is a *P-expansion* of π; we say that e is a *(partial) 1-expansion*. The skeleton tree of e is the tree reduced to a single node labeled with rule r.
Inductive step (a. construction of expansions). Let $e_0 = (P(\vec{x}), \mathcal{D}, \mathcal{I})$ be a partial P-expansion of π where $\mathcal{I} = \{Q_1(\vec{y}_1), ..., Q_n(\vec{y}_n)\}$, [$e_0$ will grow by adding 1-expansions to $\mathcal{I}$] and let $e_1 : (Q_1(\vec{y}_1), \mathcal{D}_1, \emptyset), ..., e_n : (Q_n(\vec{y}_n), \mathcal{D}_n, \emptyset)$ be 1-expansions of π such that, for $i, j = 1, ..., n$, $Var(e_i) \cap Var(e_j) \subseteq Var(\vec{y}_i) \cup Var(\vec{y}_j)$ [i.e. the common variables of e_i and e_j are distinguished variables] and $Var(e_i) \cap Var(e_0) = Var(\vec{y}_i)$ [i.e. every non distinguished variable of e_i is a fresh variable]; if for $i = 0, ..., n$ e_i has skeleton tree $\mathcal{T}_i$ then $e = (P(\vec{x}), \mathcal{D} \bigcup \mathcal{D}_1 \cup ... \cup \mathcal{D}_n, \emptyset)$ is a P-expansion of π of skeleton tree $\mathcal{T}$ obtained by adding to $\mathcal{T}_0$ n leaves $\mathcal{T}_1$,...,$\mathcal{T}_n$ such that, for every $i = 1, ..., n$, the rule that labels the father of $\mathcal{T}_i$ contains $Q_i(\vec{y}_i)$ in its body. We say that e is a 1-unfolding of e_0 via $\mathcal{I}$ with e_1,...,e_n.
Inductive step (b. construction of partial expansions). Let $e_0 = (P(\vec{x}), \mathcal{D}, \mathcal{I}_1 \cup \mathcal{I}_2 \cup \mathcal{I}_3)$ be a partial P-expansion of π where $\mathcal{I}_1 = \{Q_1(\vec{y}_1), ..., Q_m(\vec{y}_m)\}$, $\mathcal{I}_2 = \{Q_{m+1}(\vec{y}_{m+1}), ..., Q_n(\vec{y}_n)\}$ [e_0 will grow by adding 1-expansions to $\mathcal{I}_1$ and partial 1-expansions to $\mathcal{I}_2$ while $\mathcal{I}_3$ remains unchanged], $\mathcal{I}_1$ and $\mathcal{I}_2$ cannot be both empty and $\mathcal{I}_2$ and $\mathcal{I}_3$ cannot be both empty; let for $i = 1, ..., m$, $e_i : (Q_i(\vec{y}_i), \mathcal{D}_i, \emptyset)$ be a 1-expansion of π and let for $j = m+1, ..., n$, $e_j : (Q_j(\vec{y}_j), \mathcal{D}_j, \mathcal{I}_j)$ be a partial 1-expansion of π such that, for $i, j = 1, ..., n$, $Var(e_i) \cap Var(e_j) \subseteq Var(\vec{y}_i) \cup Var(\vec{y}_j)$ and $Var(e_i) \cap Var(e_0) = Var(\vec{y}_i)$; if for $i = 0, ..., n$ e_i has skeleton tree $\mathcal{T}_i$ then $e = (P(\vec{x}), \mathcal{D} \cup \mathcal{D}_1 \cup ... \cup \mathcal{D}_n, \mathcal{I}_{m+1} \cup ... \cup \mathcal{I}_n \cup \mathcal{I}_3)$ is a partial P-expansion of π of skeleton tree $\mathcal{T}$ obtained by adding to $\mathcal{T}_0$ n leaves $\mathcal{T}_1$,...,$\mathcal{T}_n$ such that, for every $i = 1, ..., n$, the rule that labels the father of $\mathcal{T}_i$ contains $Q_i(\vec{y}_i)$ in its body. We say that e is a 1-unfolding of e_0 via $\mathcal{I}_1 \cup \mathcal{I}_2$ with e_1,...,e_n.

The depth (resp. the minimal IDB depth, the minimal EDB depth, the size) of an expansion or partial expansion is the depth (resp. the minimal IDB depth, the minimal EDB depth, the size) of its skeleton tree. Therefore, if e' has size s and e is a 1-unfolding of e' with n bubbles e_1,...,e_n, then e has size $s + n$. Notice that a (partial) expansion has size 1 if and only if it is a (partial) 1-expansion.

It is straightforward to recover from our definition 2.2 the folklore definition of an expansion as a first-order formula over $\exists, \wedge$ and precisely as a conjunction of extensional atoms with a set of distinguished variables and with the remaining variables being existentially quantified: the head part $P(\vec{x})$ gives the distinguished variables $\vec{x}$ and tells us that it is a P-expansion; it suffices then to write the database part $\mathcal{D}$ as a conjunction of extensional atoms and to take the existential closure of

this conjunction over all variables not belonging to $\vec{x}$.

Moreover if, in a (partial) expansion e, we replace every variable x with a constant a_x, we can view e as a hypergraph.

Definition 2.3. Let e be a (partial) expansion $(P(\vec{x}), \mathcal{D}, \mathcal{I})$; we view e as the hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{HE})$ with set of vertices $\mathcal{V}$ and set of hyperedges $\mathcal{HE}$ defined as follows: for every variable x of e there exists in $\mathcal{V}$ a vertex with label the constant a_x; for every EDB atom $E(x^1, ..., x^n)$ in $\mathcal{D}$ there exists in $\mathcal{HE}$ a hyperedge $(a_{x^1}, ..., a_{x^n})$ with label E; the set $\{a_x \mid x \text{ occurs in } P(\vec{x}) \cup \mathcal{I}\}$ is the set of distinguished constants of the hypergraph $\mathcal{H}$.

Recall from definition 2.2 that a (partial) expansion e is either a (partial) 1-expansion or it has size $s > 1$ and it is the 1-unfolding of some partial expansion e_0 of size $s-n$ (obtained by unfolding e_0 with 1-expansions and/or partial 1-expansions $e_1, ..., e_n$); if e is a (partial) expansion of size $s > 1$ and if we view e_i, for $i = 0, ..., n$, as the hypergraph $\mathcal{H}_i = (\mathcal{V}_i, \mathcal{HE}_i)$, then we view e as the hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{HE})$ such that $\mathcal{V} = \mathcal{V}_0 \bigcup \mathcal{V}_1 \bigcup ... \bigcup \mathcal{V}_n$ and $\mathcal{HE} = \mathcal{HE}_0 \bigcup \mathcal{HE}_1 \bigcup ... \bigcup \mathcal{HE}_n$. Thus the operation that takes place when expansions compose - via unfolding - to create a new expansion is the union of the corresponding hypergraphs.

We saw that a Datalog expansion has both a hypergraph structure and a tree-like structure. The tree-like structure comes dramatically into the picture when we need to investigate the properties of expansions related to expressibility and computability questions; luckily it coincides exactly with a notion from graph theory, namely the tree-decomposition of a hypergraph and the bounded tree-width hypergraphs defined below [C90; GroM99; GLS01].

Definition 2.4. Let $\mathcal{H} = (\mathcal{V}, \mathcal{HE})$ be a hypergraph (with set of vertices $\mathcal{V}$ and set of hyperedges $\mathcal{HE}$). A *tree-decomposition* of $\mathcal{H}$ is a pair $(\mathcal{T}, f)$, where $\mathcal{T}$ is a tree (with set of nodes $\mathcal{N}$) and $f : \mathcal{N} \to \mathcal{P}(\mathcal{V})$ maps every node i of $\mathcal{T}$ to a set $f(i)$ - called *bubble* - of vertices of $\mathcal{H}$ such that
(1) $\mathcal{V} = \bigcup\{f(i) \mid i \in \mathcal{N}\}$,
(2) every hyperedge of $\mathcal{H}$ has its vertices in some set $f(i)$,
(3) if $v \in f(i) \cap f(j)$, then $v \in f(k)$ for every k belonging to the unique path in $\mathcal{T}$ linking i to j.

Definition 2.5. The width of a tree-decomposition $(\mathcal{T}, f)$ of a hypergraph $\mathcal{H}$ is the maximal cardinality of its bubbles minus one, i.e. $max\{|f(i)| \mid i \in \mathcal{N}\} - 1$. The tree-width of $\mathcal{H}$ is the minimum width of a tree-decomposition of $\mathcal{H}$. A family $\mathcal{F}$ of hypergraphs is of bounded tree-width if there exists a constant k such that, for every hypergraph $\mathcal{H} \in \mathcal{F}$, there is a tree-decomposition of $\mathcal{H}$ of tree-width at most k.

We will talk of a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{HE})$ of bounded tree-width if this hypergraph is known to belong to a family of hypergraphs of bounded tree-width.

Using our definition 2.2 of (partial) expansions we prove formally the following proposition which is implicit from [KV98].

PROPOSITION 2.6. *1. The skeleton trees associated to a program π provide natural tree-decompositions of the (partial) expansions of π.*

2. The family of expansions and partial expansions of a program π is a family of hypergraphs of bounded tree-width.

PROOF. 1. We prove that if e is a (partial) expansion and $\mathcal{T}$ is the associated skeleton tree of e, then there exists a function f such that $(\mathcal{T}, f)$ is a tree-decomposition of the expansion e viewed as a hypergraph. The function f - which maps every node of $\mathcal{T}$ to a set of vertices of e viewed as a hypergraph - will be defined in the proof.
The proof is done by induction on the size s of (partial) expansions.
$s = 1$. Let e be a (partial) 1-expansion $(P(\vec{x}), \mathcal{D}, \mathcal{I})$ viewed as the hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{HE})$ and let $\mathcal{T}$ be the associated skeleton tree of e with set of nodes the singleton $\mathcal{N}$. Let $f : \mathcal{N} \rightarrow \mathcal{P}(\mathcal{V})$ be the function that maps the unique node i of $\mathcal{N}$ to the set $\mathcal{V}$ of all vertices of $\mathcal{H}$. It is straightforward to see that the pair $(\mathcal{T}, f)$ is a tree-decomposition, of bounded tree-width, of the expansion e viewed as the hypergraph $\mathcal{H}$.
Inductive step: Let e be a (partial) expansion of size $s > 1$, obtained as the 1-unfolding of a partial expansion e_0 of size $s - n$, with 1-expansions and/or partial 1-expansions $e_1, ..., e_n$. For $i = 0, ..., n$, let e_i be viewed as the hypergraph $\mathcal{H}_i = (\mathcal{V}_i, \mathcal{HE}_i)$, let $\mathcal{T}_i$ be the skeleton tree of e_i with set of nodes $\mathcal{N}_i$ and let $f_i : \mathcal{N}_i \rightarrow \mathcal{P}(\mathcal{V}_i)$ be a function such that $(\mathcal{T}_i, f_i)$ is a tree-decomposition of bounded tree-width, of the expansion e_i ($(\mathcal{T}_0, f_0)$ is the tree-decomposition of e_0 by induction hypothesis while $(\mathcal{T}_1, f_1)$,...,$(\mathcal{T}_n, f_n)$ are the natural tree-decompositions of e_1,...,e_n which are (partial) 1-expansions). Let e be viewed as the hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{HE})$ such that $\mathcal{V} = \mathcal{V}_0 \bigcup \mathcal{V}_1 \bigcup ... \bigcup \mathcal{V}_n$ and $\mathcal{HE} = \mathcal{HE}_0 \bigcup \mathcal{HE}_1 \bigcup ... \bigcup \mathcal{HE}_n$ and let $\mathcal{T}$ be the skeleton tree associated to e (with set of nodes $\mathcal{N}$). We define the function $f : \mathcal{N} \rightarrow \mathcal{P}(\mathcal{V})$ such that, for $i = 0, ..., n$, the restriction of f to $\mathcal{N}_i$ is f_i. It is not hard to verify that f satisfies conditions 1,2 and 3 of definition 2.4, which proves that $(\mathcal{T}, f)$ is a tree-decomposition, of bounded tree-width, of the expansion e.
We now give the complete proof of the fact that f satisfies condition 3.: let b_i (labeled with rule r_i), b (labeled with rule r) and b_j (labeled with rule r_j) be three bubbles of $\mathcal{T}$ such that only b_j is a leaf and b is the father of b_j and let a_x be a vertex of $\mathcal{H}$. Notice that (i) vertex a_x belongs to $f(b_i) \cap f(b_j)$ if and only if variable x appears both in r_i and r_j and (ii) vertex a_x belongs to $f(b_i) \cap f(b)$ if and only if variable x appears both in r_i and r. If x appears both in r_i and r_j then it follows from the structure of skeleton trees that x must necessarily appear in rule r; therefore, if vertex a_x belongs to $f(b_i) \cap f(b_j)$ then vertex a_x belongs to $f(b_i) \cap f(b) = f_0(b_i) \cap f_0(b)$; since $(\mathcal{T}_0, f_0)$ is a tree-decomposition, f_0 satisfies condition 3 of definition 2.4 i.e. a_x belongs to $f_0(b_k)$ for every bubble b_k belonging to the unique path p_0 in $\mathcal{T}_0$ linking b_i to b; but path p_0 is a path of $\mathcal{T}$ and $f_0(b_k) = f(b_k)$, therefore a_x belongs to $f(b_k)$ for every bubble b_k belonging to the unique path p_0 in $\mathcal{T}$ linking b_i to b. Therefore, if vertex a_x belongs to $f(b_i) \cap f(b_j)$ then a_x belongs to $f(b_k)$ for every bubble b_k belonging to the unique path p_0 in $\mathcal{T}$ linking b_i to b_j.
2. The assertion 2. of the proposition follows from the assertion 1. , according to definition 2.5. Indeed the size of the bubbles of any skeleton tree of π is bounded by the maximum size of any rule of the program (therefore the bound is a function of the program). □

We come now to the central notion of *persistent set* of an expansion; this notion

comes from the structure of expansions as hypergraphs as explained in the following definition.

Definition 2.7. Let $\mathcal{H}$ be a hypergraph and $(\mathcal{T}, f)$ be a tree-decomposition of $\mathcal{H}$. For every subtree L of $\mathcal{T}$ with $l > 1$ bubbles b_1, b_2,..., b_l such that $A = b_1 \cap b_2 \cap ... \cap b_l \neq \emptyset$, the set $A \subseteq \mathcal{V}$ is called *persistent set of length l*; the cardinality of A is called *size* of A. The elements of a persistent set are called *persistencies*. If $\mathcal{H}$ has a persistent set A of size n and length l, we say that $\mathcal{H}$ has n persistencies of length l. If the subtree L is reduced to a branch b with l bubbles b_1, b_2,..., b_l (where b_i is the father of b_{i+1}) then the corresponding persistent set of length l is called *linear persistent set on branch* (b_1, b_2,..., b_l).

In order to properly reason with persistencies in program expansions, we have to work with *normal Datalog programs* which are programs containing in their rules (and in their expansions consequently) only IDB atoms of the form $Q(\vec{y})$ where the argument $\vec{y}$ is a vector of *distinct* variables. When a program and/or its partial expansions contain atoms having repetitions in their argument, we transform such atoms into new ones which have no repetitions in their arguments; the transformation is canonical as explained in the following definition.

Definition 2.8. 1. The atom $Q(\vec{y})$ is a *bad atom* if there are repetitions of variables in $\vec{y}$. The *transform* of $Q(\vec{y})$ is the unique atom $Q_{\vec{l}}(\vec{z})$ where $\vec{z}$ is the vector of distinct variables of $\vec{y}$ ordered according to their order of appearance and $\vec{l}$ is the sequence of the positions of the first occurrence of every component of $\vec{y}$; $\vec{l}$ is called *label*.
2. Let D be a domain, let $\vec{a}$ be a vector of elements of D containing repetitions of elements; $Q(\vec{a})$ is a *bad atom* and the *transform* of $Q(\vec{a})$ is the unique atom $Q_{\vec{l}}(\vec{b})$ defined as in 1, where $\vec{b}$ contains no repetitions of elements of D.
3. Let $Q(\vec{y})$ be a bad atom with transform $Q_{\vec{l}}(\vec{z})$ and let r be a relation on D consisting of tuples $\vec{a}$ such that $Q(\vec{a})$ is an instance of $Q(\vec{y})$; we call *transform* of r the relation r' consisting of the transforms of the tuples of r (i.e. consisting of tuples $\vec{b}$ such that $Q_{\vec{l}}(\vec{b})$ is an instance of $Q_{\vec{l}}(\vec{z})$); r is called the *inverse-transform* of r'.

Example 2.9. Consider the bad atom $Q(x, y, x, z)$: its variables are ordered $x < y < z$ according to their order of appearance; in $Q(x, y, x, z)$, the first component x first occurs at position 1, the second one y first occurs at position 2, the third one x first occurs at position 1 and the fourth one z first occurs at position 4; therefore the transform of $Q(x, y, x, z)$ is the atom $Q_{1,2,1,4}(x, y, z)$ with label $(1, 2, 1, 4)$. From the transform $Q_{1,2,1,4}(x, y, z)$ of the bad atom $Q(x, y, x, z)$ we can retrieve $Q(x, y, x, z)$. Consider now the instance $Q(a, b, a, c)$ of $Q(x, y, x, z)$; the transform of $Q(a, b, a, c)$ is the tuple $Q_{1,2,1,4}(a, b, c)$. Let $r = \{(a, b, a, c), (b, d, b, e)\}$ be a relation corresponding to instances of the bad atom $Q(x, y, x, z)$; the transform of r is the relation $r' = \{(a, b, c), (b, d, e)\}$ corresponding to instances of the transform $Q_{1,2,1,4}(x, y, z)$ of $Q(x, y, x, z)$. □.

In the present paper we will assume that all Datalog programs are normal ones; this implies no loss of generality since every program π can be transformed into a

normal program π' such that π and π' are semantically equivalent; this is explained in the following proposition.

PROPOSITION 2.10. *Every non normal Datalog program π can be transformed into a normal program π' such that every IDB predicate of π' is of the form either P or $P_{\vec{l}}$ where P is some IDB predicate of π and $\vec{l}$ is a label.*
There exists a one-to-one correspondence f between the set $\mathcal{E}_\pi$ of expansions of π and the set $\mathcal{E}_{\pi'}$ of expansions of π', satisfying the following:
1) for every P-expansion e of π, either $f(e) = e$ or there is some label $\vec{l}$ such that $f(e)$ is a $P_{\vec{l}}$-expansion of π'; conversely, for every $P_{\vec{l}}$-expansion e' of π', $e' = f(e)$ for some P-expansion e of π.
2) The expansions e and $f(e)$ have the same distinguished variables and the same database part; the distinguished variables in $f(e)$ appear in the same order as in e but without repetitions.

It follows from 1) and 2) that for every IDB predicate P of π, the set of P-expansions of π defines the same query [1] as the union, over all labels $\vec{l}$, of the $P_{\vec{l}}$-expansions of π' and of the P-expansions of π'. More precisely, if Q is the query defined by π and predicate P and $r = Q(\mathcal{D})$, and if $Q'_{\vec{l}}$ (resp. Q') is the query defined by π' and $P_{\vec{l}}$ (resp. P) and $r_{\vec{l}} = Q_{\vec{l}}(\mathcal{D})$ (resp. $r' = Q'(\mathcal{D})$) then r is the union of the relation r' and of the inverse-transforms of the relations $r_{\vec{l}}$, for all labels $\vec{l}$ that appear in the new program π'.

PROOF. We recall the following: (a) rule (resp. atom) r' is an instance of rule (resp. atom) r if there exists a substitution θ such that $r' = r\theta$; r' is a most general instance of r with a certain property if, for every instance r'' of r such that r'' has the same property as r', there is a substitution $\theta_{r''}$ such that $r'' = r'\theta_{r''}$, (c) atom A is a common instance of atoms B and C if there are substitutions θ_1 and θ_2 such that $B\theta_1 = C\theta_2 = A$; atom A is a most general common instance of atoms B and C if, for any common instance D of B and C, there is a substitution θ_D such that $D = A\theta_D$, (b) two atoms $P(\vec{x})$ and $P(\vec{y})$ are called variants of each other if they are instances of each other.

We introduce two program transformations, *transformation A* and *transformation B*, both applicable to programs which are not normal. *Transformation A*, when applied to a program π and to a nonempty set *BadAtoms* of bad atoms (we suppose that *BadAtoms* contains atoms that are not variants of each other), transforms π into a new program π' constructed as follows: for every bad IDB atom $Q(\vec{y})$ occurring in *BadAtoms* and for every rule r of π defining Q, construct - whenever it is possible - a new rule r' which is the most general instance of rule r satisfying the following: $head(r')$ is the most general common instance of $head(r)$ and $Q(\vec{y})$: we call such a rule r', *good rule derived from rule r and atom $Q(\vec{y})$*. If π is a program containing bad atoms then, by iterating the application of transformation A to π, one produces all instances of rules of π, used in any expansion of π, and all bad atoms that appear in partial expansions of π. *Transformation B*, when applied to a program π, produces a new program π'', which is normal, by replacing in every rule of π, every bad atom with its transform; rules of π without bad atoms remain

[1] For a formal definition of the notion of query, see definition 3.5.

unchanged in the new program π''. We give now the *normalization algorithm* that produces from a program π which is not normal, a new program π' which is normal and is semantically equivalent to π.

Normalization algorithm
Begin
$OldProgram := \emptyset$
$Program := \pi$ *input program*
$BadAtoms :=$ {Bad atoms in π}
While $BadAtoms \neq \emptyset$ do
  begin
  $OldProgram := OldProgram \bigcup Program$
  $Program :=$ program produced by transformation A applied to π and $BadAtoms$
  $BadAtoms :=$ {Bad atoms in $Program$ having no variant occurring in $OldProgram$}
  end
$Program := OldProgram \bigcup Program$
$Output :=$ program produced by transformation B applied to $Program$ *output program*
End

The Normalization algorithm, on input program π produces output program π'. Let us call π'' the intermediate program produced when we definitely quit the while loop (i.e. when we cannot apply transformation A anymore). Notice that the output program π' is obtained in the last step of the algorithm by applying the transformation B to the intermediate program π''. It is not hard to verify that transformation B preserves semantical equivalence; therefore the two programs π'' and π' are semantically equivalent, since they satisfy the following: (1) if $P(\vec{x})$ is not a bad atom then $(P(\vec{x}), \mathcal{D}, \emptyset)$ is an expansion of π'' if and only if it is an expansion of π' and (2) if $P(\vec{x})$ is a bad atom with transform $P_{\vec{l}}(\vec{u})$ then $(P(\vec{x}), \mathcal{D}, \emptyset)$ is an expansion of π'' if and only if $(P_{\vec{l}}(\vec{u}), \mathcal{D}, \emptyset)$ is an expansion of π'.

Consequently, in order to show that the input program π is semantically equivalent to the output program π' it is sufficient to show that π is semantically equivalent to the intermediate program π'' produced when we definitely quit the while loop.

Termination The iteration of the while loop terminates since all possible bad atoms are instances of atoms of the input program π and these instances only use the set of variables of π, so their total number is finite.

Correctness It is not hard to see that (1) before entering the while loop, the expansions of $Program$ are expansions of the input program π and that (2) after entering the while loop, the expansions of $Program$ are still expansions of π. This loop invariant shows us that the expansions of π'' (produced when we definitely quit the while loop) are expansions of π.

Completeness It is not hard to prove - by induction on the depth of expansions (using the definition of transformation A) - that every expansion of π is an expansion of π''. This shows that the expansions of π are expansions of π''. □

To illustrate the idea of the above transformations, we give the following example.

Example 2.11. The following program π is not normal because of the set $BadAtoms_1$ $= \{P(x,y,x), Q(y,y,u), P(z,y,y), Q(z,y,y)\}$ of bad atoms that occur in it.

$r_1 : P(x,y,x) \leftarrow Q(y,y,u), E(x,x,y)$
$r_2 : P(x,y,z) \leftarrow P(z,y,y), E(x,z,u)$
$r_3 : Q(x,y,z) \leftarrow Q(z,y,y), E(u,x,y)$
$r_4 : Q(x,y,z) \leftarrow E(x,y,z)$
We first apply transformation A to program π and to the set $BadAtoms_1$ of bad atoms.
We do not have to create a *good rule* derived from r_1 and $P(x,y,x)$ since it is rule r_1 itself.
We create a *good rule* r_2' derived from r_2 and $P(x,y,x)$:
$r_2' : P(x,y,x) \leftarrow P(x,y,y), E(x,x,u)$
We create a *good rule* r_1' derived from r_1 and $P(z,y,y)$ and a *good rule* r_2'' derived from r_2 and $P(z,y,y)$:
$r_1' : P(x,x,x) \leftarrow Q(x,x,u), E(x,x,x)$
$r_2'' : P(x,y,y) \leftarrow P(y,y,y), E(x,y,u)$
We create a *good rule* r_3' derived from r_3 and $Q(y,y,u)$ and a *good rule* r_4' derived from r_4 and $Q(y,y,u)$:
$r_3' : Q(x,x,z) \leftarrow Q(z,x,x), E(u,x,x)$
$r_4' : Q(x,x,z) \leftarrow E(x,x,z)$
We create a *good rule* r_3'' derived from r_3 and $Q(z,y,y)$ and a *good rule* r_4'' derived from r_4 and $Q(z,y,y)$:
$r_3'' : Q(x,y,y) \leftarrow Q(y,y,y), E(u,x,y)$:
$r_4'' : Q(x,y,y) \leftarrow E(x,y,y)$.

These new rules form a new program π':
$r_1' : P(x,x,x) \leftarrow Q(x,x,u), E(x,x,x)$
$r_2' : P(x,y,x) \leftarrow P(x,y,y), E(x,x,u)$
$r_2'' : P(x,y,y) \leftarrow P(y,y,y), E(x,y,u)$
$r_3' : Q(x,x,z) \leftarrow Q(z,x,x), E(u,x,x)$
$r_3'' : Q(x,y,y) \leftarrow Q(y,y,y), E(u,x,y)$
$r_4' : Q(x,x,z) \leftarrow E(x,x,z)$
$r_4'' : Q(x,y,y) \leftarrow E(x,y,y)$.

Program π' contains two new bad atoms $P(y,y,y)$ and $Q(y,y,y)$ that did not occur in π; all other bad atoms of π' have variants that occur in π so we don't need to treat them again. We now apply transformation A to π and to the new set $BadAtoms_2 = \{P(y,y,y), Q(y,y,y)\}$ of bad atoms.
We don't have to create a *good rule* derived from r_1 and $P(y,y,y)$, since such a rule, namely rule r_1', still exists in π'.
We create a *good rule* r_2''' derived from r_2 and $P(y,y,y)$:
$r_2''' : P(x,x,x) \leftarrow P(x,x,x), E(x,x,u)$:
We create a *good rule* r_3''' derived from r_3 and $Q(y,y,y)$ and a *good rule* r_4''' derived from r_4 and $Q(y,y,y)$:
$r_3''' : Q(x,x,x) \leftarrow Q(x,x,x), E(u,x,x)$
$r_4''' : Q(x,x,x) \leftarrow E(x,x,x)$.

These new rules form a new program π'' that does not contain any new bad atom (the bad atoms that occur in π'' have variants in π', thus they have been treated already):
$r_2''' : P(x,x,x) \leftarrow P(x,x,x), E(x,x,u)$

$r_3''' : Q(x,x,x) \leftarrow Q(x,x,x), E(u,x,x)$
$r_4''' : Q(x,x,x) \leftarrow E(x,x,x).$

We now apply transformation B to the program $\pi \cup \pi' \cup \pi''$. We obtain the following normal program Π (each rule R_i of Π comes from a rule r_i of π, π' or π''):
$R_1 : P_{1,2,1}(x,y) \leftarrow Q_{1,1,3}(y,u), E(x,x,y)$
$R_1' : P_{1,1,1}(x) \leftarrow Q_{1,1,3}(x,u), E(x,x,x)$
$R_2 : P(x,y,z) \leftarrow P_{1,2,2}(z,y), E(x,z,u)$
$R_2' : P_{1,2,1}(x,y) \leftarrow P_{1,2,2}(x,y), E(x,x,u)$
$R_2'' : P_{1,2,2}(x,y) \leftarrow P_{1,1,1}(y), E(x,y,u)$
$R_2''' : P_{1,1,1}(x) \leftarrow P_{1,1,1}(x), E(x,x,u)$
$R_3 : Q(x,y,z) \leftarrow Q_{1,2,2}(z,y), E(u,x,y)$
$R_3' : Q_{1,1,3}(x,z) \leftarrow Q_{1,2,2}(z,x), E(u,x,x)$
$R_3'' : Q_{1,2,2}(x,y) \leftarrow Q_{1,1,1}(y), E(u,x,y)$
$R_3''' : Q_{1,1,1}(x) \leftarrow Q_{1,1,1}(x), E(u,x,x)$
$R_4 : Q(x,y,z) \leftarrow E(x,y,z)$
$R_4' : Q_{1,1,3}(x,z) \leftarrow E(x,x,z)$
$R_4'' : Q_{1,2,2}(x,y) \leftarrow E(x,y,y)$
$R_4''' : Q_{1,1,1}(x) \leftarrow E(x,x,x).$ □

3. USEFUL FAMILIES OF EXPANSIONS

In this section we introduce the important notion of useful set of expansions: given a program π, we call *useful* any subset of the set of expansions of π which allows to capture the semantics of the program i.e. the query that the program defines (this is explained in the first subsection). We use the terminology *useful family of expansions* or simply *useful family*. Trivially the set of all expansions of π is a useful family of expansions; we are obviously interested in useful families that are proper subsets of the set of all expansions of π. We will exhibit program properties such that each of these properties is inherently related to the existence of some useful family with specific features. First such example is the relationship between boundedness and the existence of a finite useful family (given in the second subsection of this section). A second example of this kind will be given in the next section, where we characterize the persistency number of a Datalog program in terms of useful families.

3.1 Preliminaries of queries via expansions and homomorphisms

We recall some well-known definitions of the logical notion of satisfiability and its equivalent formulation in terms of homomorphisms.

Definition 3.1. A *distinguished database* is a tuple $\mathcal{D}^* = (\mathcal{D}, a_1, ..., a_n)$ where $\mathcal{D} = (D, r_1, \ldots, r_n)$ is a database and $(a_1, ..., a_n)$ is a tuple of distinct elements of the domain D of $\mathcal{D}$.
Let π be a program; we say that the distinguished database $\mathcal{D}^* = (\mathcal{D}, a_1, ..., a_n)$ is *P-accepted* by expansion $e = (P(x_1, ..., x_n), \mathcal{C}, \emptyset)$ of π if there exists a valuation $\mathcal{V}$ from the set of variables of e to the domain D of $\mathcal{D}$ such that (1) for $i = 1, ..., n$, $\mathcal{V}(x_i) = a_i$ and (2) e, viewed as a first-order formula, is satisfied in $\mathcal{D}$ under the valuation $\mathcal{V}$. We say that the distinguished database $\mathcal{D}^* = (\mathcal{D}, a_1, ..., a_n)$ is P-accepted by program π if there exists an expansion $e = (P(x_1, ..., x_n), \mathcal{C}, \emptyset)$ of

π such that $\mathcal{D}^* = (\mathcal{D}, a_1, ..., a_n)$ is P-accepted by e. We equivalently say that expansion e (or program π) P-accepts $\mathcal{D}^*$. When P is clear from context, we talk about "acceptance" instead of "P-acceptance".

Every expansion $(P(x_1, ..., x_n), \mathcal{D}, \emptyset)$ can be seen as the distinguished database $((D, r_1, \ldots, r_n), a_{x_1}, ..., a_{x_n})$ with domain $D = \{a_x \mid x \text{ is a variable occurring in } \mathcal{D}\}$ where, for $i = 1, ..., n$, $\vec{a}_x \in r_i$ if and only if $R_i(\vec{x}) \in \mathcal{D}$; we will use the same notation $\mathcal{D}$ for both the database part of the expansion (which is a set of EDB atoms) and for the database $(D, r_1, \ldots, r_n)$: therefore we will say that the expansion $e = (P(x_1, ..., x_n), \mathcal{D}, \emptyset)$ is viewed as the distinguished database $(\mathcal{D}, a_{x_1}, ..., a_{x_n})$.

Definition 3.2. Let $\mathcal{D}^* = ((D, r_1, \ldots, r_m), a_1, ..., a_n)$ and $\mathcal{D}'^* = ((D', r'_1, \ldots, r'_m), a'_1, ..., a'_n)$ be two distinguished databases such that r_i has the same arity as r'_i for $i = 1, ..., m$. A *homomorphism* from $\mathcal{D}^*$ to $\mathcal{D}'^*$, is a total function $h : D \to D'$ such that (1) $h(a_i) = a'_i$ for $i = 1, ..., n$ and (2) if $(a_1, \ldots, a_{m_i}) \in r_i$, then $(h(a_1), \ldots, h(a_{m_i})) \in r'_i$.

The composition of two homomorphisms is a homomorphism.

PROPOSITION 3.3. *The distinguished database $\mathcal{D}^*$ is P-accepted by the P-expansion $e = (P(x_1, ..., x_n), \mathcal{C}, \emptyset)$ of π if there exists a homomorphism h from e - viewed as the distinguished database $(\mathcal{C}, a_{x_1}, ..., a_{x_n})$ - to $\mathcal{D}^*$.*
The distinguished database $\mathcal{D}^*$ is P-accepted by program π if there exists a P-expansion e of π such that $\mathcal{D}^*$ is P-accepted by e.

PROOF. Immediate from definitions 3.1 and 3.2. □

Example 3.4. Consider the distinguished database $(\mathcal{D}, a, a)$ where $\mathcal{D}$ has domain $D = \{a, b, c, d\}$ and relation $\{(a, b), (b, c)\}$, let π be the following program:

$$r_1 : T(x, y) \ :- \ E(x, z), T(z, y)$$
$$r_2 : T(x, y) \ :- \ E(x, z)$$

and let e_0, e_1 and e_2 be the following T-expansions of π for the goal $T(x, y)$, written in their folklore notation as existential first-order formulas: (1) $e_0 = \exists z E(x, z)$, (2) $e_1 = \exists z \exists u E(x, z) \wedge E(z, u)$ and (3) $e_2 = \exists z \exists z_1 \exists v E(x, z) \wedge E(z, z_1) \wedge E(z_1, v)$.

We formalize e_0 as the 3-tuple $(T(x, y), \mathcal{C}_0, \emptyset)$ and view it as the distinguished database $(\mathcal{C}_0, a_x, a_y)$ where $\mathcal{C}_0$ is the database with domain $C_0 = \{a_x, a_y, a_z\}$ and relation $\{(a_x, a_z)\}$. The distinguished database $(\mathcal{D}, a, a)$ is T-accepted by expansion e_0 via the homomorphism h_0 from $\mathcal{C}_0$ to $\mathcal{D}$, such that $h_0(a_x) = a$, $h_0(a_y) = a$, $h_0(a_z) = b$.

We formalize e_1 as the 3-tuple $(T(x, y), \mathcal{C}_1, \emptyset)$ and view it as the distinguished database $(\mathcal{C}_1, a_x, a_y)$ where $\mathcal{C}_1$ has domain $C_1 = \{a_x, a_y, a_z, a_u\}$ and relation $\{(a_x, a_z), (a_z, a_u)\}$. The distinguished database $(\mathcal{D}, a, a)$ is T-accepted by expansion e_1 via the homomorphism h_1 from $\mathcal{C}_1$ to $\mathcal{D}$, such that $h_1(a_x) = a$, $h_1(a_y) = a$, $h_1(a_z) = b$, $h_1(a_u) = c$.

Notice that the above homomorphisms h_0 and h_1 are related in the following way: there exists a homomorphism h from $\mathcal{C}_0$ to $\mathcal{C}_1$ such that $h_1 \circ h = h_0$; this homomorphism h is the identity on C_0.

We formalize e_2 as the 3-tuple $(T(x, y), \mathcal{C}_2, \emptyset)$ and view it as the distinguished database $(\mathcal{C}_2, a_x, a_y)$ where $\mathcal{C}_2$ has domain $C_2 = \{a_x, a_y, a_z, a_{z_1}, a_v\}$ and relation

$\{(a_x, a_z), (a_z, a_{z_1}), (a_{z_1}, a_v)\}$. Since there is no homomorphism from $\mathcal{C}_2$ to $\mathcal{D}$, we conclude that the distinguished database $(\mathcal{D}, a, a)$ is not accepted by expansion e_2. □

Definition 3.5. A *query* is a function Q from databases (of some fixed type) to relations of fixed arity such that the image of a database with domain D is a relation on D. A query has to be *generic*, i.e. invariant under renamings of the domain; this means that, for every domain D' such that there exists a one-to-one correspondance $i : D \longrightarrow D'$, $Q(i(\mathcal{D})) = i(Q(\mathcal{D}))$ [2].
1. A P-expansion $e = (P(x_1, x_2, \ldots, x_m), \mathcal{C}, \emptyset)$ of program π defines a query $Q^e_{\pi,P}$ as follows: for every database $\mathcal{D}$, $Q^e_{\pi,P}(\mathcal{D}) = \{(d_1, d_2, \ldots, d_m)$: the distinguished database $(\mathcal{D}, d_1, ..., d_m)$ is P-accepted by e, viewed as the distinguished database $(\mathcal{C}, a_{x_1}, ..., a_{x_m})\}$.
2. Let (π, P) be a pair consisting of a Datalog program π and an IDB predicate P of arity m (P is the *goal* predicate of π). The pair (π, P) defines a query $Q_{\pi,P}$ as follows: for every database $\mathcal{D}$, $Q_{\pi,P}(\mathcal{D}) = \{(d_1, d_2, \ldots, d_m)$: there exists a P-expansion e of π such that $(d_1, d_2, \ldots, d_m) \in Q^e_{\pi,P}(\mathcal{D})\}$; or equivalently $Q_{\pi,P}(\mathcal{D}) = \{(d_1, d_2, \ldots, d_m)$: the distinguished database $(\mathcal{D}, d_1, ..., d_m)$ is P-accepted by $\pi\}$.

Definition 3.5 shows that Datalog is a query language; moreover it relates the query defined by a pair (π, P) with the set of P-expansions of the Datalog program π. The other equivalent approach consists in viewing Datalog programs as a declarative query language with the following semantics: let $\mathcal{D}$ be a database, thought of as a collection of facts about the EDB predicates of a program π. Let $Q^k_{\pi,P}(\mathcal{D})$ be the collection of facts about an IDB predicate P that can be deduced from $\mathcal{D}$ by at most k applications of the rules in π (this defines the bottom up evaluation). If we consider P the *goal* predicate, then π expresses a query $Q_{\pi,P}$, where $Q_{\pi,P}(\mathcal{D}) = \bigcup_{k\geq 0} Q^k_{\pi,P}(\mathcal{D})$. The query $Q_{\pi,P}$ is expressed as the infinitary disjunction of the set of P-expansions of π.

3.2 Useful families and their relation to boundedness

We now introduce our notion of useful family of expansions. The idea behind the notion of useful family is the following: recall that the query $Q_{\pi,P}$ is expressed as the infinitary disjunction of the set of P-expansions of π. Any subset $\mathcal{E}$ of the set of P-expansions of π such that the query $Q_{\pi,P}$ is expressed as the disjunction d of the P-expansions in $\mathcal{E}$ is called *useful family of P-expansions*; notice that if $\mathcal{E}$ is finite the disjunction d is finitary. We give below a simple definition of the notion of useful family of expansions.

Definition 3.6. Let π be a program and let $\mathcal{E}$ be a set of P-expansions of π. We say that $\mathcal{E}$ is a P-useful family of expansions of π if every distinguished database P-accepted by π is P-accepted by some $e' \in \mathcal{E}$. When P is clear from context, we simply say "useful family" instead of "P-useful family".

The following proposition gives an alternative definition of the notion of P-useful family.

[2]Let i be a one-to-one correspondance $i : D \longrightarrow D'$, let r be a relation on D and let $\mathcal{D} = (D, r_1, \ldots, r_m)$ be a database. We denote by $i(r)$ the relation on $D' = i(D)$ such that $(b_1, ..., b_t) \in r$ if and only if $(i(b_1), ..., i(b_t)) \in i(r)$. We denote by $i(\mathcal{D})$ the database $(i(D), i(r_1), \ldots, i(r_m))$.

PROPOSITION 3.7. *Let π be a program and let $\mathcal{E}$ be a set of P-expansions of π. $\mathcal{E}$ is a P-useful family of expansions of π if and only if every P-expansion e of π (viewed as distinguished database) is P-accepted by some $e' \in \mathcal{E}$.*

PROOF. ($\Longrightarrow$) Suppose that $\mathcal{E}$ is a P-useful family of expansions of π. According to definition 3.6, every P-expansion of π (viewed as distinguished database) is P-accepted by some $e' \in \mathcal{E}$.
($\Longleftarrow$) Suppose now that every P-expansion of π (viewed as distinguished database) is P-accepted by some $e' \in \mathcal{E}$. By definition 3.2 every distinguished database $\mathcal{D}^*$ P-accepted by π, is P-accepted by some P-expansion of π and it follows (by composing the homomorphisms) that $\mathcal{D}^*$ is P-accepted by some $e' \in \mathcal{E}$. □

Definition 3.8. Let π_1 and π_2 be two Datalog programs and let P be any IDB predicate, of arity p, occurring both in π_1 and π_2. Programs π_1 and π_2 are *P-equivalent* if for every database $\mathcal{D}$ with domain D and for every p-tuple $(a_1, ..., a_p) \in D^p$, $(a_1, ..., a_p) \in Q_{\pi_1,P}(\mathcal{D})$ if and only if $(a_1, ..., a_p) \in Q_{\pi_2,P}(\mathcal{D})$. We also say that (π_1, P) and (π_2, P) are equivalent or simply that π_1 and π_2 are equivalent.

PROPOSITION 3.9. *Let π_1 and π_2 be two programs and let P be an IDB predicate occurring in both π_1 and π_2. The following statements are equivalent.*
1. Programs π_1 and π_2 are P-equivalent.
2. Pairs (π_1, P) and (π_2, P) define the same query.
3. Programs π_1 and π_2 P-accept the same distinguished databases.
4. Every P-expansion of program π_1 is P-accepted by π_2 and vice-versa.

PROOF. 1) $\Longrightarrow$ 2) Follows immediately from definitions 3.8 and 3.5 (2).
2) $\Longrightarrow$ 3) Follows immediately from definitions 3.5 (2) and 3.2.
3) $\Longrightarrow$ 4) Follows from the fact that, since 3) holds, all P-expansions of π_1 and all P-expansions of π_2 must be among the distinguished databases P-accepted by π_1 and P-accepted by π_2.
4) $\Longrightarrow$ 1) Since 4) holds, it follows - from definition 3.5 and by composing the homomorphisms - that, for every database $\mathcal{D}$ with domain D and for every p-tuple $(a_1, ..., a_p) \in D^p$, $(a_1, ..., a_p) \in Q_{\pi_1,P}(\mathcal{D})$ if and only if $(a_1, ..., a_p) \in Q_{\pi_2,P}(\mathcal{D})$, which means, according to proposition 3.8, that programs π_1 and π_2 are P-equivalent.

A sufficient condition assuring that program π is P-equivalent to program π' is when π and π' have the same set of P-expansions.

Definition 3.10. 1. We say that two programs are strongly P-equivalent if they have the same set of P-expansions, where P is an IDB predicate occurring in both π and π'.
2. We say that two programs are strongly equivalent if they have the same set of P-expansions, for every predicate P occurring in both π and π'.

PROPOSITION 3.11. *Let π and π' be two programs. If π and π' are strongly equivalent then they are P-equivalent for every IDB predicate P occurring both in π and π'.*

We now illustrate the notion of usefulness by giving a characterization of the boundedness property of Datalog programs in terms of useful families.

Definition 3.12. 1. A program π is *recursive* if there exists a cycle in the dependency graph of π.
2. A recursive program π is *bounded w.r.t. its IDB predicate* P if there exists an integer K such that every distinguished database accepted by some P-expansion of π is accepted by some P-expansion of depth $< K$; this property is called *predicate boundedness.*
3. A recursive program π is *bounded* if it is bounded w.r.t. all its IDB predicates; this property is called *program boundedness.*

Obviously a non recursive program is bounded. Suppose now that program π is recursive and that, in the dependency graph of π, there is no path starting from P and reaching some cycle; then obviously, π is bounded w.r.t. P. It has been proved that (1) a program is bounded w.r.t. P if and only if it is P-equivalent to a non recursive program and that (2) a program π is bounded w.r.t. all its IDB predicates P if and only if, for each IDB predicate P in π, program π is P-equivalent to a non recursive program. Notice that decidability of predicate boundedness implies decidability of program boundedness; but the converse does not hold. Notice also that the converse of proposition 3.11 does not hold: consider any recursive bounded program π and a nonrecursive program π' which is equivalent to π; π has an infinite set of expansions while π' has a finite set of expansions.

LEMMA 3.13. *1. A Datalog program π is bounded w.r.t. its IDB predicate P if and only if π has a finite P-useful family of expansions.*
2. A Datalog program π is bounded if and only if π has a finite P-useful family of expansions, for every IDB predicate symbol P.

PROOF. 1. Recall that a program π is bounded w.r.t. its IDB predicate P if there exists an integer K such that every distinguished database accepted by π is accepted by some P-expansion of depth $< K$. The set of all P-expansions of π of depth $< K$ forms a P-useful family of expansions of π; clearly this useful family is finite.
2. Obvious. □

From the previous characterization of boundedness in terms of useful families and from the undecidability of program boundedness and predicate boundedness [GMSV87], we obtain the following undecidability facts:

PROPOSITION 3.14. *1. There is no algorithm that takes an arbitrary recursive program π and an arbitrary IDB predicate P as input and determines if π has a finite P-useful family.*
2. There is no algorithm to determine if a recursive program has a finite P-useful family, for every P.

4. PERSISTENCY NUMBERS OF A DATALOG PROGRAM

In this section we introduce the *persistency number*, the *weak persistency number* and the *syntactic persistency number* of a Datalog program (written from the strongest to the weakest notion). These numbers concern various "levels" of presence of persistent variables: the weakest and more straightforward notion is that of syntactic persistency number, which is defined from the syntactic form of the

program rules (this notion appears first in [C89]); the new notion of weak persistency number is syntactic too, but concerns the presence of persistent variables in the set of all expansions of the program, while the notion of persistency number (first defined in [Af97]) is more related to the program semantics. In section 7 we introduce a fourth - more semantical - notion, the *persistency number-modulo equivalence* which is invariant up to program equivalence.

First we introduce the persistency numbers of a program π with respect to a given IDB predicate symbol P of π (for short we call them P-persistency numbers).

Definition 4.1. Let π be a program and let P be an IDB predicate symbol of π.
1. Program π has *syntactic P-persistency number m* if m is the maximum integer among those integers n satisfying the following: there exists in π a rule ρ defining predicate P such that n variables of $head(\rho)$ occur in some IDB atom of $body(\rho)$.
2. Program π has *weak P-persistency number m* if m is the minimum integer satisfying the following: there exists an integer k such that for every P-expansion e of π having $> m$ persistencies of length l [3], it is true that $l < k$. We call *characteristic integer of π w.r.t.-weak-P-persistency-number* the minimum integer satisfying the requirements of the integer k in the definition of the weak P-persistency number of π.
3. Program π has *P-persistency number m* if m is the minimum integer satisfying the following: there exists an integer k such that for every database D, P-accepted by π, there exists a P-expansion e that accepts D such that if e has $> m$ persistencies of length l then $l < k$.

One of our long term objectives is to simplify the occurrences of persistent sets. We have managed so far to accomplish this, in the case of persistent sets occurring in the family of all program's expansions but always with bounded length. This bound is the characteristic integer.

We introduce in a similar way the various persistency numbers of a program π, defined independently of any specific IDB predicate of π.

Definition 4.2. 1. Program π has *syntactic persistency number m* if m is the maximum among those integers n satisfying the following: there exists an IDB predicate P of π such that n is the syntactic P-persistency number of π.
2. Program π has *weak persistency number m* if m is the maximum among those integers n satisfying the following: there exists an IDB predicate P of π such that n is the weak P-persistency number of π. We call *characteristic integer of π w.r.t.-weak-persistency-number* the maximum among those integers l satisfying the following: there exists an IDB predicate P of π such that l is the characteristic integer of π w.r.t.-weak-P-persistency-number.
3. Program π has *persistency number m* if m is the maximum among those integers n satisfying the following: there exists an IDB predicate P of π such that n is the P-persistency number of π.

Observe that a non-recursive Datalog program has persistency number zero (with respect to any of its IDB predicates).

We now introduce the notion for a family of hypergraphs to be *of-m-persistencies*.

[3] Recall definition 2.7.

Definition 4.3. Let $\mathcal{L}$ be a (possibly infinite) family of hypergraphs with a given tree decomposition, for each member of the family. We say that $\mathcal{L}$ is of-m-persistencies if m is the minimum integer satisfying the following: there exists an integer k such that for every element $e \in \mathcal{L}$ if e has $> m$ persistencies of length l then $l < k$. If $\mathcal{L}$ is finite then $\mathcal{L}$ is of-0-persistencies.

LEMMA 4.4. *1. Program π has weak P-persistency number m if and only if the set $\mathcal{E}_\pi^P$ of all P-expansions of π is a family of-m-persistencies.*
2. Program π has weak persistency number m if and only if the set $\mathcal{E}_\pi$ of all expansions of π is a family of-m-persistencies.
3. Program π has P-persistency number m if and only if m is the minimum integer such that π has a P-useful family of expansions of-m-persistencies.

PROOF. The proof follows from definitions 3.6, 4.1, 4.2 and 4.3. □

Lemma 4.4 shows that the notion of family of expansions of-m-persistencies gives a nice characterization of the weak persistency number and the persistency number in terms of useful families; that characterization will be heavily used in the sequel of the paper (especially in the proof of lemma 7.2). It is worthwhile noticing that, according to lemma 4.4, the weak persistency number and the persistency number of a program π are both related - in a similar way - to the existence of a family $\mathcal{F}$ of expansions of m-persistencies for some well-chosen m: for the weak persistency number $\mathcal{F}$ is the set of all expansions (i.e the trivial useful family) of π, while for the persistency number, $\mathcal{F}$ is a useful family of-m-persistencies for which m is minimum; for instance, if π has no proper useful family of expansions then its persistency number coincides with its weak persistency number. Therefore the first impression is that the persistency number is a deeper notion than the weak persistency number; in forthcoming sections our results on the interrelationship between these two numbers and on the possibility to decide if each of them has a given value ("their decidability problem" treated in section 8) will prove that the first impression is correct.

We now give some preliminary results that relate the syntactic persistency number with the weak persistency number.

Definition 4.5. Let $\mathcal{D}$ be a set of EDB atoms, $\mathcal{I}$ be a set of IDB atoms and $r : P(\vec{x}) \leftarrow \mathcal{D}, \mathcal{I}$ be a rule. We say that rule r is a *witness of syntactic persistency* $> m$ if there exists an IDB atom $Q_i(\vec{y}_i) \in body(r)$ such that $> m$ variables in $\vec{y}_i$ occur in $\vec{x}$. We say that rule $r : P(\vec{x}) \leftarrow \mathcal{D}, \mathcal{I}$ is a witness of syntactic persistency $\leq m$ if r is not a witness of syntactic persistency $> m$. Every initialization rule is considered as a witness of syntactic persistency $\leq m$, for every integer m.

PROPOSITION 4.6. *1. For every program π and every integer m, every expansion of π, viewed as a rule, is a witness of syntactic persistency $\leq m$.*
2. For every program π and for every partial expansion e of π, e has a persistent set of size $> m$ and of length $\geq k$ if (a) e - viewed as a rule - is a witness of syntactic persistency $> m$ and (b) e has minimal IDB depth $\geq k$.
3. If program π has weak persistency number m and K is the characteristic integer of π w.r.t.-weak-persistency-number then every partial expansion e of π of minimal IDB depth $\geq K$, is - viewed as a rule - a witness of syntactic persistency $\leq m$.

PROOF. 1. immediate since every expansion defines an initialization rule.
2. obvious from definitions 4.5, 2.2 and 2.4 condition 3. Recall that the (partial) expansion $(P(\vec{x}), \mathcal{D}, \mathcal{I})$ can be viewed as the rule $P(\vec{x}) \leftarrow \mathcal{D}, \mathcal{I}$.
3. obvious from definition 4.2 and proposition 4.6 2. □

PROPOSITION 4.7. *For every program π, we denote by m^s_π (resp. m^w_π) the syntactic (resp. weak) persistency number of π. From every program π such that $m^s_\pi > m^w_\pi$, we can construct a program π' strongly equivalent [4] to π such that $m^s_{\pi'} = m^w_\pi$.*

PROOF. Let π be a program with weak persistency number equal to m^w_π and with characteristic integer w.r.t.-weak-persistency-number equal to K_0; notice that we can compute these two numbers (for the weak persistency number see subsection 8.1 and for the characteristic integer w.r.t.-weak-persistency-number see subsection 8.2). For any fixed K, $K \geq K_0$, we construct from π a new program π' and show that (1) $m^s_{\pi'} = m^w_\pi$ and (2) π' is strongly equivalent to π; the set $R_{\pi'}$ of rules of π' is the disjoint union of two sets A and B: the set A contains exactly the rules of π which are witnesses of syntactic persistency $\leq m^w_\pi$ (A contains at least the initialization rules of π) and the set B contains exactly all expansions of π of depth $\leq K$ and all partial expansions of π having both depth and minimal IDB depth equal to K; the elements of this set B are, according to proposition 4.6, witnesses of syntactic persistency $\leq m^w_\pi$. Thus every rule of π' is a witness of syntactic persistency $\leq m^w_\pi$, therefore π' has syntactic persistency number $\leq m^w_\pi$. Obviously every expansion of π' is an expansion of π. We show now that every expansion of π is an expansion of π'. We call $fact(m)$ the fact that every expansion of π of depth at most mK is an expansion of π'. We have to show that $fact(m)$ holds for every $m \geq 1$.
Basis $m = 1$: $fact(1)$ holds since by construction every expansion of π of depth at most K is a rule of π'.
Inductive step: suppose that $fact(m)$ holds; we show that $fact(m+1)$ holds. Let e, viewed as the hypergraph $(\mathcal{V}, \mathcal{HE})$ be an expansion of π of skeleton tree $\mathcal{T}$ and of depth k such that $mK < k \leq (m+1)K$. Consider the partial expansion e_0 with skeleton tree $\mathcal{T}_0$ such that $\mathcal{T}_0$ is obtained by erasing from $\mathcal{T}$ every node of depth $> K$; e_0 is a partial expansion having both depth and minimal IDB depth equal to K, therefore e_0 is a rule of π'. Let e_0 be viewed as the hypergraph $(\mathcal{V}_0, \mathcal{HE}_0)$ and let $e_1 = (\mathcal{V}_1, \mathcal{HE}_1)$, ...,$e_n = (\mathcal{V}_n, \mathcal{HE}_n)$ be the expansions such that $\mathcal{V} = \mathcal{V}_0 \bigcup \mathcal{V}_1 \bigcup ... \bigcup \mathcal{V}_n$ and $\mathcal{HE} = \mathcal{HE}_0 \bigcup \mathcal{HE}_1 \bigcup ... \bigcup \mathcal{HE}_n$; e_1,...,e_n have depth at most mK thus, by induction hypothesis, they are expansions of π'; therefore e is an expansion of π'.
This ends up the proof that every expansion of π is an expansion of π'; thus π and π' have the same set of expansions which means - according to definition 3.10 - that they are strongly equivalent. It follows that π' has weak persistency number m^w_π; thus π' has syntactic persistency number $\geq m^w_\pi$. Since we showed, at the beginning of the proof, that π' has syntactic persistency number $\leq m^w_\pi$ we conclude that π' has syntactic persistency number m^w_π. □

[4] Recall definition 3.10.

5. ADORNED DEPENDENCY GRAPH OF A PROGRAM: A TOOL FOR VISUALIZING PERSISTENT SETS

We visualize program expansions and their persistencies by means of the *adorned dependency graph* which is a labeled version of the well-known dependency graph of the program. On the labels of this new graph appear special functions, called *pattern transformations*, which express, for each rule r and each IDB atom $Q(\vec{y})$ occurring in $body(r)$, the argument vector $\vec{y}$ in terms of the argument vector $\vec{x}$ of $head(r)$.

Definition 5.1. For every pair (X, Y) of variable vectors where $X = (x_1, ..., x_p)$ (the variables $x_1, ..., x_p$ are distinct) and $Y = (y_1, ..., y_q)$ (the variables $y_1, ..., y_q$ are distinct) we define a total function $\theta : \{1, 2, ..., q\} \to \{1, 2, ..., p, \star\}$ (where $\star$ is a new symbol) as follows: $\theta(i) = j$ if $y_i = x_j$ and $\theta(i) = \star$ if $y_i \neq x_j \ \forall j = 1, ..., p$. The function θ defined by (X, Y) is called a *pattern transformation* of type (p, q) (θ is not necessarily an onto function). We say that $\theta : \{1, 2, ..., q\} \to \{1, 2, ..., p, \star\}$ is a *m-pattern transformation*, $0 \leq m \leq q$, if there are exactly m elements of $\{1, 2, ..., q\}$ such that their image under θ is different from $\star$.

Every pattern transformation $\theta : \{1, 2, ..., q\} \to \{1, 2, ..., p, \star\}$ of type (p, q) can be defined by infinitely many pairs of variable vectors in such a way that for every pair of variable vectors (X, Y) defining θ the following holds: if $X = (x_1, ..., x_p)$ is a vector of distinct variables and if every variable of Y is either a variable of X or a distinct variable u then $Y = (x_{\theta(1)}, ..., x_{\theta(q)})$ where $x_\star = u$; in that case the pattern transformation θ defined by the pair (X, Y) is an m-pattern transformation if and only if X and Y have exactly m variables in common.

Example 5.2. The pattern transformation defined by the pair $((x, y, z), (y, u, x, v))$ is the substitution $\theta = \{1|2, 2|\star, 3|1, 4|\star\} : \{1, 2, 3, 4\} \to \{1, 2, 3, \star\}$; notice that θ is also defined by the pair $((u, v, y), (v, w, u, t))$ for instance. However the two pairs $((x, y, z, t), (y, u, x, v))$ and $((x, y, z), (y, u, x, v))$ do not define the same pattern transformation: the pair $((x, y, z, t), (y, u, x, v))$ defines the pattern transformation $\theta' = \{1|2, 2|\star, 3|1, 4|\star\} : \{1, 2, 3, 4\} \to \{1, 2, 3, 4, \star\}$ with codomain $\{1, 2, 3, 4, \star\}$ which is different from the codomain $\{1, 2, 3, \star\}$ of θ. □

Definition 5.3. Let $\sigma_1 : \{1, 2, ..., p\} \to \{1, 2, ..., s, \star\}$ and $\sigma_2 : \{1, 2, ..., q\} \to \{1, 2, ..., p, \star\}$ be two pattern transformations of respective types (s, p) and (p, q), respectively defined by pairs (X, Y) and (Y, Z). Let $\sigma_1^* : \{1, 2, ..., p, *\} \to \{1, 2, ..., s, \star\}$ be the pattern transformation which is equal to σ_1 on $\{1, 2, ..., p\}$ and such that $\sigma_1^*(*) = *$. We denote by $\sigma_1\sigma_2 : \{1, 2, ..., q\} \to \{1, 2, ..., s, \star\}$ the composition of σ_1^* and σ_2 such that $\sigma_1\sigma_2(x) = \sigma_1^*(\sigma_2(x))$; the function $\sigma_1\sigma_2$ is a pattern transformation of type (s, q), defined by the pair (X, Z') where Z' is the vector of variables obtained from Z by replacing each variable $x \in X \setminus Y$ with a new variable.

Example 5.4. Suppose that $X = Z = (x)$, $Y = (y)$. Let $\sigma_1 = \{1|\star\} : \{1\} \to \{1, \star\}$ and $\sigma_2 = \{1|\star\} : \{1\} \to \{1, \star\}$ be two pattern transformations of respective types $(1, 1)$ and $(1, 1)$, respectively defined by pairs (X, Y) and (Y, Z); the function $\sigma_1\sigma_2$ is the pattern transformation $\{1|\star\}$ of type $(1, 1)$, defined by the pair (X, Z') where $Z' = (u)$. The pattern transformation of type $(1, 1)$, defined by the pair (X, Z) is the function $\{1|1\} \neq \sigma_1\sigma_2$. □

It is important to notice that the situation described in example 5.4 cannot happen in our context since in expansions each variable that disappears at some point, never appears again later on; therefore the composition $\sigma_1\sigma_2(x)$ of σ_1^* and σ_2 in definition 5.3 can be naturally formulated as "pattern transformation of type (s, q), defined by the pair (X, Z)".

Definition 5.5. The *adorned dependency graph* ADG_π of π has the set of IDB predicates as set of vertices and its set of edges is defined as follows: for every recursive rule r with head $P(\vec{x})$ and for every IDB atom $Q_i(\vec{y}_i)$ in $body(r)$ there is a directed edge from P to Q_i with label the pair (r, θ_i) where θ_i is the pattern transformation defined by $(\vec{x}, \vec{y}_i)$. The sequence (r_1,σ_1),...,(r_n,σ_n) such that $p : P \stackrel{(r_1,\sigma_1)}{\rightarrow} ... \stackrel{(r_n,\sigma_n)}{\rightarrow} Q$ is a path in ADG_π of length n starting from P is called the *P-trace* of path p; we say that (r_1,σ_1),...,(r_n,σ_n) is a P-trace of π (or simply a trace of π) of length n and of *label* $\sigma_1...\sigma_n$. We say that (r_1,σ_1),...,(r_n,σ_n) is a trace of the expansion e of π if $r_1, ..., r_n$ is a branch in the skeleton tree $Skel$ of e; if r_1 is the root of $Skel$ and r_n is the father of a leaf of $Skel$ then the trace is called *full trace* of e. If $t : (r_1,\sigma_1)$,...,(r_n,σ_n) is a trace then the sequence (r_i,σ_i),...,(r_j,σ_j), $1 \leq i \leq j \leq n$, is called *subtrace* of t.

We now show how paths in the adorned dependency graph allow to reason about program expansions and their persistent sets. Indeed every path $p : Q_0 \stackrel{(r_1,\sigma_1)}{\rightarrow} ... \stackrel{(r_n,\sigma_n)}{\rightarrow} Q_n$ in the adorned dependency graph ADG_π of program π provides two pieces of information: (1) the sequence $r_1 \rightarrow ... \rightarrow r_n$ corresponds to a branch $b : N_1 : r_1 \rightarrow ... \rightarrow N_n : r_n$ of skeleton trees (where by $N : r$ we denote that node N has label r) and (2) the m-pattern transformation $\sigma_2...\sigma_{n-1}$ determines, if $m > 0$, the existence of a linear persistent set of size m and of length n, which occurs in all expansions containing the branch b in their skeleton tree. According to definition 2.7, every persistent set has length at least 2. It follows from the definition 2.2 of expansions and from the canonical tree-decomposition of an expansion viewed as a hypergraph (see proof of proposition 2.6) that two bubbles b_1 and b_2 in the skeleton tree of expansion e, such that b_1 is the father of b_2, define a trivial persistent set of e of length 2. Therefore the length n of non-trivial persistent sets is $n \geq 3$.

PROPOSITION 5.6. *Let $e = (\mathcal{V}, \mathcal{HE})$ be an expansion of π with tree-decomposition $(\mathcal{T}, f)$.*
1. For every path $b : N_1 : r_1 \rightarrow ... \rightarrow N_n : r_n$ with $n > 1$ nodes in $\mathcal{T}$ there is a unique path $p : Q_1 \stackrel{(r_1,\sigma_1)}{\rightarrow} ... \stackrel{(r_{n-1},\sigma_{n-1})}{\rightarrow} Q_n$ of length $n-1$ in the adorned dependency graph ADG_π of π, where Q_n is the head predicate of rule r_n.
2. Moreover the set $S = f(N_1) \cap ... \cap f(N_n)$ is a linear persistent set (on path $(r_1, ..., r_n)$) of size $m > 0$ and of length n, if and only if $\sigma_2...\sigma_{n-1}$ is an m-pattern transformation.

PROOF. The proof of part 1. is by induction on the number $n > 1$ of nodes in b.
Basis $n = 2$
Let $b : N_1 : r_1 \rightarrow N_2 : r_2$ be a path of the skeleton tree $\mathcal{T}$, where the two rules r_1 and r_2 are such that $r_1 : Q_1(\vec{x}_1) \leftarrow ..., Q_2(\vec{y}_1), ...$ and $r_2 : Q_2(\vec{x}_2) \leftarrow ...$ and let σ_1 be the pattern transformation defined by the pair $(\vec{x}_1, \vec{y}_1)$. We associate to b the

path $p : Q_1 \overset{(r_1,\sigma_1)}{\rightarrow} Q_2$ of the adorned dependency graph ADG_π of π.
Inductive step:
Let $b : N_1 : r_1 \rightarrow ... \rightarrow N_n : r_n$ be a path with n nodes in the skeleton tree $\mathcal{T}$. Let $p_0 : Q_1 \overset{(r_1,\sigma_1)}{\rightarrow} ... \overset{(r_{n-2},\sigma_{n-2})}{\rightarrow} Q_{n-1}$ be the path of length $n-2$ in ADG_π which, by induction hypothesis, we associate to the initial segment (with $n-1$ nodes) $b_0 : N_1 : r_1 \rightarrow ... \rightarrow N_{n-1} : r_{n-1}$ of path b. The last step $b_1 : N_{n-1} : r_{n-1} \rightarrow N_n : r_n$ of path b is a path of $\mathcal{T}$ with 2 nodes, where the two rules r_{n-1} and r_n are such that $r_{n-1} : Q_{n-1}(\vec{x}_{n-1}) \leftarrow ..., Q_n(\vec{y}_{n-1}), ...$ and $r_n : Q_n(\vec{x}_n) \leftarrow ....$ We associate to b_1 the path $p_1 : Q_{n-1} \overset{(r_{n-1},\sigma_{n-1})}{\rightarrow} Q_n$ where σ_{n-1} is the pattern substitution defined by the pair $(\vec{x}_{n-1}, \vec{y}_{n-1})$. Finally we associate to b the concatenation of p_0 and p_1 i.e. the path $p : Q_1 \overset{(r_1,\sigma_1)}{\rightarrow} ... \overset{(r_{n-2},\sigma_{n-2})}{\rightarrow} Q_{n-1} \overset{(r_{n-1},\sigma_{n-1})}{\rightarrow} Q_n$.

We prove now part 2.
Recall that to the path $b : N_1 : r_1 \rightarrow ... \rightarrow N_n : r_n$ in $\mathcal{T}$ we associate the unique path $p : Q_1 \overset{(r_1,\sigma_1)}{\rightarrow} ... \overset{(r_{n-1},\sigma_{n-1})}{\rightarrow} Q_n$ in ADG_π and suppose that the set $S = f(N_1) \cap ... \cap f(N_n)$ is a persistent set of size $m > 0$. For every $i = 2, ..., n-1$, the rule r_i has a variant $Q_i(\vec{x}_i) \leftarrow ...Q_{i+1}(\vec{x}_{i+1}), ...$ such that the pattern transformation σ_i is defined by the pair $(\vec{x}_i, \vec{x}_{i+1})$. The m vertices of the set S correspond to the m variables of $\vec{x}_2 \cap ... \cap \vec{x}_n$. From the definition of the mapping f in a tree-decomposition, $\vec{x}_2 \cap ... \cap \vec{x}_n = \vec{x}_2 \cap \vec{x}_n$ and this exactly means that $\sigma_2...\sigma_{n-1}$ is an m-pattern transformation. Let now $\sigma_2...\sigma_{n-1}$ be the pattern transformation defined by the pair $(\vec{x}_2, \vec{x}_n)$, such that, for every $i = 2, ..., n-1$, the rule $Q_i(\vec{x}_i) \leftarrow ...Q_{i+1}(\vec{x}_{i+1}), ...$ is a variant of r_i. By hypothesis, $\sigma_2...\sigma_{n-1}$ is an m-pattern transformation which means that $\vec{x}_2$ and $\vec{x}_n$ have m variables in common and that $f(N_1)$ and $f(N_n)$ have m elements in common. Since $(\mathcal{T}, f)$ is a tree-decomposition $f(N_1) \cap f(N_n) = f(N_1) \cap ... \cap f(N_n)$; thus the set $S = f(N_1) \cap ... \cap f(N_n)$ is a persistent set of size $m > 0$ and of length n. The previous argument uses programs' normal form; indeed, in a program which is not normal, $f(N_1) \cap f(N_n)$ may be affected by the parts of $\mathcal{T}$ around path b (which means that some variables may have been identified in the path from the root until the beginning of b and therefore path b is not sufficient to determine the exact size of the persistent set). □

6. AUTOMATA RECOGNIZING PERSISTENT SETS

In the previous section we saw that from the traces of a program π we can determine linear persistent sets occurring in expansions of π. In fact we can - and will - reason on traces instead of reasoning on persistent sets as explained in the following corollary to proposition 5.6.

COROLLARY 6.1. *To every linear persistent set of size m and of length n occurring on path $(r_1, ..., r_n)$ in some expansion of π corresponds naturally some trace $t = (r_1, \sigma_1),...,(r_n, \sigma_n)$ such that $\sigma_2...\sigma_{n-1}$ is an m-pattern transformation. To every trace $t = (r_1, \sigma_1),...,(r_n, \sigma_n)$ in ADG_π such that $\sigma_2...\sigma_{n-1}$ is an m-pattern transformation corresponds the set of all linear persistent sets on path $(r_1, ..., r_n)$, of size m and of length n occurring in some expansion of π.*

An important property of traces is that they can be recognized by a finite automaton. First we need some definitions:

Definition 6.2. We call *type set of program* π the set $\mathcal{T}ype_\pi$ of all pairs (p,q), $p,q>0$, such that there exists a pattern transformation θ of type (p,q) in some label of the adorned dependency graph of π. We call *transformation set of* π the set $\mathcal{T}_\pi$ of all possible pattern transformations with type belonging to the type set $\mathcal{T}ype_\pi$ of π. If the adorned dependency graph of π contains a pattern substitution with type (p,q) then the transformation set $\mathcal{T}_\pi$ of π contains all pattern transformations with type (p,q) i.e all total functions $f:\{1,2,...,q\}\rightarrow\{1,2,...,p,\star\}$.
The sets $\mathcal{T}ype_\pi$ and $\mathcal{T}_\pi$ are both finite. We denote by $\mathcal{T}_m$ the subset of $\mathcal{T}_\pi$ consisting of all m-pattern transformations; we denote by $\mathcal{T}_{>m}$ the subset of $\mathcal{T}_\pi$ consisting of all n-pattern transformations for $n>m$.

Recall that traces have been introduced in definition 5.5.

Definition 6.3. 1. A trace t of π is called (m,K)-good when, if t has length $\geq K$ then the label of t does not belong to $\mathcal{T}_{>m}$. [5]
2. A trace t of π is called (m,K)-acceptable if t and all subtraces of t are (m,K)-good.

PROPOSITION 6.4. *Let π be a program and $\mathcal{U}$ be a family of expansions of π. The following propositions are equivalent:*
1. $\mathcal{U}$ is of-m-persistencies.
2. m is the minimum integer satisfying the following: there exists an integer K such that for every expansion $e\in\mathcal{U}$, every trace of e is (m,K)-good.
3. m is the minimum integer satisfying the following: there exists an integer K such that for every expansion $e\in\mathcal{U}$, every trace of e is (m,K)-acceptable.
4. m is the minimum integer satisfying the following: there exists an integer K such that for every expansion $e\in\mathcal{U}$, every full trace of e is (m,K)-acceptable.

PROOF. 1) $\Leftrightarrow$ 2) It follows from definitions 6.3 and 4.3.
2) $\Rightarrow$ 3) It follows immediately from definition 6.3.
3) $\Rightarrow$ 4) obvious since every full trace is a trace.
4) $\Rightarrow$ 2) obvious from definition 6.3 (2). □

LEMMA 6.5. *1. Let π be a program, let P be an IDB predicate symbol of π, let σ be a pattern transformation in the transformation set $\mathcal{T}_\pi$ of π and let $K>1$ and $0<m\leq max(arity)_\pi$ be integers (recall that $max(arity)_\pi$ is the maximum arity of the IDB predicates of π).*
1. There exists a finite non deterministic automaton $\mathcal{A}_{(\pi,P,\sigma)}$ which recognizes the set of all P-traces of π of label σ.
2. There exists a finite non deterministic automaton $\mathcal{A}_\pi$ which recognizes the set of all traces of π.
3. There exists a finite non deterministic automaton $\mathcal{B}_{(\pi,m,K)}$ which recognizes the set of all (m,K)-acceptable traces of π [6].

A. **Construction of the automaton $\mathcal{A}_{(\pi,P,\sigma)}$.**

[5] Recall definition 5.5.
[6] Recall definition 6.3.

- The set of states S of the automaton $\mathcal{A}_{(\pi,P,\sigma)}$ is the finite set of 2-tuples of the form (T,θ) where T is an IDB predicate symbol of π and θ is a pattern transformation belonging to the transformation set $\mathcal{T}_\pi$.

- The set of initial states of the automaton $\mathcal{A}_{(\pi,P,\sigma)}$ consists of the state (P,σ).

- The set of final states of the automaton $\mathcal{A}_{(\pi,P,\sigma)}$ consists of all states (T,ϵ) where T is an IDB predicate symbol of π and ϵ is the identity function. Notice that the transformation set $\mathcal{T}_\pi$ may contain more than one pattern transformation which is the identity function on its domain; we use ϵ for any of these identity functions since it is clear from the context how to distinguish them.

- The alphabet Σ of the automaton $\mathcal{A}_{(\pi,P,\sigma)}$ is the finite set of 2-tuples of the form (r,θ) where r is a rule of π and θ is a pattern transformation belonging to the transformation set $\mathcal{T}_\pi$.

- The transition relation $\Delta \subseteq S\times\Sigma\times S$ is defined as follows: $((T,\theta_1),(r,\theta),(T',\theta'))$ $\in \Delta$ if $\theta_1 = \theta\theta'$ and $T \overset{(r,\theta)}{\rightarrow} T'$ is an edge in the adorned dependency graph ADG_π of π. We will often use the notation $\rho((T,\theta_1),(r,\theta)) = (T',\theta')$ whenever $((T,\theta_1),(r,\theta),$ $(T',\theta')) \in \Delta$. In particular $\rho((T,\theta),(r,\theta)) = (T',\epsilon)$ if there is an edge $T \overset{(r,\theta)}{\rightarrow} T'$ in ADG_π.

In general the automaton $\mathcal{A}_{(\pi,P,\sigma)}$ is not deterministic as shown by the following remarks: (a) if the rules of π have more than one IDB in their bodies, suppose that in the adorned dependency graph ADG_π of π there exist two edges $T \overset{(r,\theta)}{\rightarrow} T'$ and $T \overset{(r,\theta)}{\rightarrow} T''$, $T' \neq T''$ and consider the state (T,θ_1) of the automaton such that $\theta_1 = \theta\theta'$ for some substitution θ'; in that case there exist two distinct transitions $\rho((T,\theta_1)(r,\theta)) = (T',\theta')$ and $\rho((T,\theta_1)(r,\theta)) = (T'',\theta')$ from the state (T,θ_1) both with label (r,θ).
(b) even if π is linear, for two given pattern transformations θ_1 and θ, there is not a unique θ' such that $\theta_1 = \theta\theta'$, which means that the corresponding transition is not unique.

Let $w = (r_1,\sigma_1)...(r_n,\sigma_n) \in \Sigma^*$. A *run of the automaton* $\mathcal{A}_{(\pi,P,\sigma)}$ *on* w is a sequence of states $(T_0,\theta_0),...,(T_n,\theta_n)$ such that (T_0,θ_0) is an initial state and $\rho((T_i,\theta_i),(r_i,\sigma_i)) = (T_{i+1},\theta_{i+1})$ for $i = 0,...,n-1$. A run of the automaton $\mathcal{A}_{(\pi,P,\sigma)}$ on w is *accepting* if (T_n,θ_n) is a final state. It is not hard to prove the following proposition by induction on the length of w using the definition of the automaton $\mathcal{A}_{(\pi,P,\sigma)}$.

PROPOSITION 6.6. *For every* $w = (r_1,\sigma_1)...(r_n,\sigma_n) \in \Sigma^*$ *there exists an accepting run* $(P,\sigma_1...\sigma_n),...,$
(T_n,ϵ) *of the automaton* $\mathcal{A}_{(\pi,P,\sigma)}$ *on* w *if and only if* w *is a* P*-trace of* π *of label* $\sigma_1...\sigma_n$.

B. Construction of the automaton $\mathcal{A}_\pi$.

From the automaton $\mathcal{A}_{(\pi,P,\sigma)}$ we define a new automaton $\mathcal{A}_\pi$ recognizing all traces of π. The automaton $\mathcal{A}_\pi$ has exactly the same characteristics as $\mathcal{A}_{(\pi,P,\sigma)}$ except that all states are initial states.

PROPOSITION 6.7. *For every* $w = (r_1,\sigma_1)...(r_n,\sigma_n) \in \Sigma^*$ *there exists an accepting run* $(T_0,\sigma),...,(T_n,\epsilon)$ *of the automaton* $\mathcal{A}_\pi$ *on* w *if and only if* w *is a* T_0*-trace of* π.

C. **Construction of the automaton $\mathcal{B}_{(\pi,m,K)}$.**

- The set of states S of the automaton $\mathcal{B}_{(\pi,m,K)}$ consists of the $(K+2)$-tuples of the form $(Q, \Sigma_1, ..., \Sigma_{K+1})$ where Q is an IDB predicate symbol and $\Sigma_1, ..., \Sigma_{K+1}$ are subsets of the transformation set $\mathcal{T}_\pi$.

- The set of initial states of the automaton $\mathcal{B}_{(\pi,m,K)}$ consists of the states $(Q, \emptyset, ..., \emptyset)$ where Q is an IDB predicate symbol.

- The set of final states of the automaton $\mathcal{B}_{(\pi,m,K)}$ consists of all states $(Q, \Sigma_1, ..., \Sigma_{K+1})$ where Q is an IDB predicate symbol and $\Sigma_1, ..., \Sigma_{K+1}$ are subsets of $\mathcal{T}_\pi$ such that $\Sigma_K \bigcap \mathcal{T}_{>m} = \emptyset$ and $\Sigma_{K+1} \bigcap \mathcal{T}_{>m} = \emptyset$ where $\mathcal{T}_{>m}$ is the subset of $\mathcal{T}_\pi$ consisting of all n-pattern transformations for $n > m$.

- The alphabet Σ of the automaton $\mathcal{B}_{(\pi,m,K)}$ is equal to the alphabet of the automaton $\mathcal{A}_{(\pi,P,\sigma)}$.

- The transition relation $\Delta \subseteq S \times \Sigma \times S$ is defined as follows: $((T, \Sigma_1, ..., \Sigma_{K+1}), (r, \theta), (T', \Sigma'_1, ..., \Sigma'_{K+1})) \in \Delta$ if (a) $T \overset{(r,\theta)}{\rightarrow} T'$ is an edge in the adorned dependency graph ADG_π of π and (b) $\Sigma'_1 = \{\theta\}$, $\Sigma'_i = \{\sigma\theta \mid \sigma \in \Sigma_{i-1}\}$ for $i = 2, ..., K-1$, $\Sigma'_K = \{\sigma\theta \mid \sigma \in \Sigma_{K-1} \cup \Sigma_K\}$ and $\Sigma'_{K+1} = \Sigma_K \cup \Sigma_{K+1}$.

PROPOSITION 6.8. *Let $w = (r_1, \sigma_1)...(r_n, \sigma_n) \in \Sigma^*$ and let $r : (T_0, \emptyset, ..., \emptyset), (T_1, \Sigma^1_1, ..., \Sigma^1_{K+1}), ..., (T_n, \Sigma^n_1, ..., \Sigma^n_{K+1})$ be a run of the automaton $\mathcal{B}_{(\pi,m,K)}$ on w. For $i = 1, ..., K-1$, the set Σ^n_i consists of the labels of all suffixes t of w of length i (we call suffixes of the word $a_1...a_n$ the words $a_i a_{i+1}...a_n$ for $1 \leq i \leq n$). The set Σ^n_K consists of the labels of all suffixes t of w of length $\geq K$. The set Σ^n_{K+1} consists of the labels of all subtraces t of w of length $\geq K$ such that t is not a suffix of w.*

PROOF. By induction on the length n of w.
Basis $(n = 1)$ $w = (r_1, \sigma_1) \in \Sigma^*$ and $(T_0, \emptyset, ..., \emptyset), (T_1, \Sigma^1_1, ..., \Sigma^1_{K+1})$ is a run on w. Since w has length 1, the set Σ^1_1 consists of σ_1 and for $i = 2, ..., K+1$, the set Σ^1_i is empty; we find these values for $\Sigma^1_1, ..., \Sigma^1_{K+1}$ knowing that $(((T_0, \emptyset, ..., \emptyset), (r_1, \sigma_1), (T_1, \Sigma^1_1, ..., \Sigma^1_{K+1})) \in \Delta$ and applying the definition of the transition relation Δ.
Inductive Step: Let $w = (r_1, \sigma_1)...(r_n, \sigma_n) \in \Sigma^*$ and let $r : (T_0, \emptyset, ..., \emptyset), (T_1, \Sigma^1_1, ..., \Sigma^1_{K+1}), ..., (T_n, \Sigma^n_1, ..., \Sigma^n_{K+1})$ be a run of the automaton $\mathcal{B}_{(\pi,m,K)}$ on w. The prefix $r^{n-1} : (T_0, \emptyset, ..., \emptyset), (T_1, \Sigma^1_1, ..., \Sigma^1_{K+1}), ..., (T_{n-1}, \Sigma^{n-1}_1, ..., \Sigma^{n-1}_{K+1})$ of the run r of the automaton $\mathcal{B}_{(\pi,m,K)}$ on w is a run of $\mathcal{B}_{(\pi,m,K)}$ on the prefix $w^{n-1} = (r_1, \sigma_1)...(r_{n-1}, \sigma_{n-1})$ of length $n-1$ of w. By induction hypothesis we know that, for $i = 1, ..., K-1$, the set Σ^{n-1}_i consists of the labels of all suffixes t of w^{n-1} of length i, that the set Σ^{n-1}_K consists of the labels of all suffixes t of w^{n-1} of length $\geq K$ and that the set Σ^{n-1}_{K+1} consists of the labels of all subtraces t of w^{n-1} of length $\geq K$ such that t is not a suffix of w^{n-1}. Now $(T_n, \Sigma^n_1, ..., \Sigma^n_{K+1}) = \rho((T_{n-1}, \Sigma^{n-1}_1, ..., \Sigma^{n-1}_{K+1}), (r_n, \sigma_n))$ therefore $\Sigma^n_1 = \{\sigma_n\}$, $\Sigma^n_i = \{\sigma\sigma_n \mid \sigma \in \Sigma^{n-1}_{i-1}\}$ for $i = 2, ..., K-1$, $\Sigma^n_K = \{\sigma\sigma_n \mid \sigma \in \Sigma^{n-1}_{K-1} \bigcup \Sigma^{n-1}_K\}$ and $\Sigma^n_{K+1} = \Sigma^{n-1}_K \bigcup \Sigma^{n-1}_{K+1}$; this precisely means that, for $i = 1, ..., K-1$, the set Σ^n_i consists of the labels of all suffixes t of w of length i, that the set Σ^n_K consists of the labels of all suffixes t of w of length $\geq K$ and that the set Σ^n_{K+1} consists of the labels of all subtraces t of w of length $\geq K$ such that t is not a suffix of w. □

A run $(T_0, \emptyset, ..., \emptyset), (T_1, \Sigma^1_1, ..., \Sigma^1_{K+1}), ..., (T_n, \Sigma^n_1, ..., \Sigma^n_{K+1})$ of the automaton $\mathcal{B}_{(\pi,m,K)}$ on $w = (r_1, \sigma_1)...(r_n, \sigma_n) \in \Sigma^*$ is *accepting* if, for $i = 1, ...n$, $\rho((T_{i-1}, \Sigma^{i-1}_1,$

$..., \Sigma^{i-1}_{K+1}), (r_i, \sigma_i)) = (T_i, \Sigma^i_1, ..., \Sigma^i_{K+1})$ and if $\Sigma^n_K \bigcap \mathcal{T}_{>m} = \emptyset$ and $\Sigma^n_{K+1} \bigcap \mathcal{T}_{>m} = \emptyset$.

PROPOSITION 6.9. *For every $w = (r_1, \sigma_1)...(r_n, \sigma_n) \in \Sigma^*$, there exists an accepting run $(T_0, \emptyset, ..., \emptyset), (T_1, \Sigma^1_1, ..., \Sigma^1_{K+1}), ..., (T_n, \Sigma^n_1, ..., \Sigma^n_{K+1})$ of the automaton $\mathcal{B}_{(\pi,m,K)}$ on w if and only if w is a (m, K)-acceptable trace of π.*

PROOF. The proof of proposition 6.9 follows from proposition 6.8 and from the fact that the two conditions $\Sigma^n_K \bigcap \mathcal{T}_{>m} = \emptyset$ and $\Sigma^n_{K+1} \bigcap \mathcal{T}_{>m} = \emptyset$ which characterize (m, K)-acceptable traces are satisfied because $(T_n, \Sigma^n_1, ..., \Sigma^n_{K+1})$ is a final state. More precisely:
($\Longrightarrow$) If there exists an accepting run $(T_0, \emptyset, ..., \emptyset), (T_1, \Sigma^1_1, ..., \Sigma^1_{K+1}), ..., (T_n, \Sigma^n_1, ..., \Sigma^n_{K+1})$ of the automaton $\mathcal{B}_{(\pi,m,K)}$ on the trace $w = (r_1, \sigma_1)...(r_n, \sigma_n)$ then $\Sigma^n_K \bigcap \mathcal{T}_{>m} = \emptyset$ and $\Sigma^n_{K+1} \bigcap \mathcal{T}_{>m} = \emptyset$ because $(T_n, \Sigma^n_1, ..., \Sigma^n_{K+1})$ is a final state; since $\mathcal{T}_{>m} \neq \emptyset$, the two conditions $\Sigma^n_K \bigcap \mathcal{T}_{>m} = \emptyset$ and $\Sigma^n_{K+1} \bigcap \mathcal{T}_{>m} = \emptyset$ are satisfied either when the length of w is less than K (i.e. $\Sigma^n_K = \Sigma^n_{K+1} = \emptyset$) or when the length of w is $\geq K$ and no (proper or not proper) subtrace of w of length $\geq K$ has its label in $\mathcal{T}_{>m}$; in both cases w is a (m, K)-acceptable trace.
($\Longleftarrow$) By definition of the automaton $\mathcal{B}_{(\pi,m,K)}$ we know that if w is trace of π then there exists a run $(T_0, \emptyset, ..., \emptyset), (T_1, \Sigma^1_1, ..., \Sigma^1_{K+1}), ..., (T_n, \Sigma^n_1, ..., \Sigma^n_{K+1})$ of $\mathcal{B}_{(\pi,m,K)}$ on w. If now w is a (m, K)-acceptable trace of π then either the length of w is less than K which means that $\Sigma^n_K = \Sigma^n_{K+1} = \emptyset$ or the length of w is $\geq K$ and the label of any subtrace of w of length $\geq K$ does not belong to $\mathcal{T}_{>m}$; in both cases the two conditions $\Sigma^n_K \bigcap \mathcal{T}_{>m} = \emptyset$ and $\Sigma^n_{K+1} \bigcap \mathcal{T}_{>m} = \emptyset$ are satisfied which means that $(T_n, \Sigma^n_1, ..., \Sigma^n_{K+1})$ is a final state and thus the run is accepting. □

7. THE VARYING ROLE OF PERSISTENCIES IN DATALOG PROGRAMS: THE MAIN LEMMA

In this section we present results that show the different role of each persistency number; moreover a fourth kind of persistency number comes out which is not equivalent to the strongest notion of the three.

We first give preliminary results that relate the persistency number with the weak persistency number, using the notion of (m, k)-acceptable trace and the automata-theoretic techniques which are both introduced in section 6.

LEMMA 7.1. *Let π be a program and let $max(arity)_\pi$ be the maximum arity of IDB predicate symbols occurring in π. For every $m \leq max(arity)_\pi$ and for every k, we can construct a program π' such (1) every IDB predicate T of π is an IDB predicate of π', (2) for every IDB predicate T of π, the T-expansions of π' are exactly the T-expansions of π having only (m, k)-acceptable traces.*

PROOF. Recall from proposition 6.9 that for every $m \leq max(arity)_\pi$ and for every k we can construct the automaton $\mathcal{B}_{(\pi,m,k)}$ which recognizes all (m, k)-acceptable traces of π. We now show that from $\mathcal{B}_{(\pi,m,k)}$ we can construct a program π' with set of expansions the expansions of π having only (m, k)-acceptable traces; obviously such a program π' satisfies the two conditions of the lemma. Program π' is constructed from π as follows: for every IDB predicate T of π, for every recursive rule $r : T(\vec{x}) \leftarrow T_1(\vec{y}_1), ..., T_s(\vec{y}_s), e_1, ..., e_l$ of π and for every set of transitions $t = \{(T, \Sigma_1, ..., \Sigma_n) \overset{(r,\theta_1)}{\rightarrow} (T_1, \Sigma^1_1, ..., \Sigma^1_n), ..., (T, \Sigma_1, ..., \Sigma_n) \overset{(r,\theta_s)}{\rightarrow} (T_s, \Sigma^s_1, ..., \Sigma^s_n)\}$ in

the automaton $\mathcal{B}_{(\pi,m,k)}$, create $s+1$ new IDB predicates: (1) $T^{(\Sigma_1,...,\Sigma_n)}$ which has the same arity as T, (2) $T_1^{(\Sigma_1^1,...,\Sigma_n^1)}$ which has the same arity as T_1,..., (s+1) $T_s^{(\Sigma_1^s,...,\Sigma_n^s)}$ which has same arity as T_s; then replace r with the new recursive rule $r^t_{(\Sigma_1,...,\Sigma_n)} : T^{(\Sigma_1,...,\Sigma_n)}(\vec{x}) \leftarrow T_1^{(\Sigma_1^1,...,\Sigma_n^1)}(\vec{y}_1), ..., T_s^{(\Sigma_1^s,...,\Sigma_n^s)}(\vec{y}_s), e_1, ..., e_l$ of π'. For every new IDB predicate symbol $T^{(\Sigma_1,...,\Sigma_n)}$ such that $(T, \Sigma_1, ..., \Sigma_n)$ is a final state and for every initialization rule $r : T(\vec{x}) \leftarrow e_1, ..., e_l$ of π, create the new initialization rule $r_{(\Sigma_1,...,\Sigma_n)} : T^{(\Sigma_1,...,\Sigma_n)}(\vec{x}) \leftarrow e_1, ..., e_l$ of π'. At last, for every new IDB predicate $T^{(\emptyset,...,\emptyset)}$, add to π' the new rule $T(\vec{x}) \leftarrow T^{(\emptyset,...,\emptyset)}(\vec{x})$. It is not hard to see that (i) the T-expansions of π' have only (m,k)-acceptable traces and that (ii) if $(T(\vec{x}), \mathcal{D}, \emptyset)$ is a T-expansion of π' then $(T(\vec{x}), \mathcal{D}, \emptyset)$ is a T-expansion of π. □

The following is one of the main results of the paper and it relates the three notions of persistency numbers, (1) the persistency number, (2) the weak persistency number and (3) the syntactic persistency number.

LEMMA 7.2. *Consider the following propositions:*
1) program π has P-persistency number m
2) program π has a P-equivalent program with P-weak persistency number m
3) program π has weak persistency number m
4) program π has a strongly equivalent program with syntactic persistency number m.
The implications 1) ⇒ 2) and 3) ⇒ 4) hold while the implications 2) ⇒ 1) and 4) ⇒ 3) do not hold.

PROOF. 1) ⇒ 2) Since program π has P-persistency number m, we know from lemma 4.4 that m is the minimum integer such that π has a P-useful family of expansions of-m-persistencies. Let $\mathcal{U}$ be such a P-useful family of expansions of-m-persistencies for program π. The integer m is the minimum integer such that - according to proposition 6.4 - there exists an integer K such that for every expansion $e \in \mathcal{U}$, every trace of e is (m,K)-acceptable; for such an integer K, consider the automaton $\mathcal{B}_{(\pi,m,K)}$. If we know K we can construct the automaton $\mathcal{B}_{(\pi,m,K)}$ and from this automaton we can construct, according to lemma 7.1, a program π' such that, for every IDB predicate of P of π, the set $\mathcal{E}^P_{\pi'}$ of P-expansions of π' consists of the P-expansions of π having only (m,K)-acceptable traces; thus - according to proposition 6.4 - $\mathcal{E}^P_{\pi'}$ is of-m-persistencies which means that π' has weak P-persistency number m. Every expansion of π' is, by construction, an expansion of π and, since $\mathcal{U} \subseteq \mathcal{E}_{\pi'}$ is a P-useful family of π, every P-expansion of π is P-accepted by a P-expansion of π'. Thus π' is P-equivalent to π according to proposition 3.9. Thus we proved that (1) program π has a P-equivalent program π' such that π' has P-weak persistency number m and that (2) if we know K then we can effectively construct π'.
3) ⇒ 4) this is precisely proposition 4.7.
The implications 2) ⇒ 1) and 4) ⇒ 3) do not hold: see the counter-examples given in the example 7.4 below. □

COROLLARY 7.3. *1. If program π has weak P-persistency number m then π has a strongly P-equivalent program with syntactic persistency number m.*
2. If program π has P-persistency number m then π has a P-equivalent program

with syntactic P-persistency number m.

PROOF. 1. This can be easily proved by a straightforward adaptation of the proof of proposition 4.7.
2. Follows immediately from 1. and from lemma 7.2. □

We will further discuss about the constructibility of lemma 7.2 in subsection 8.2. The following example is a counterexample showing that the implications 2) ⇒ 1) and 4) ⇒ 3) of lemma 7.2 do not hold.

Example 7.4. We consider two strongly equivalent programs π and π', both defining the transitive closure query.
Program π consists of the two rules
$r_1 : T(x,y) \leftarrow E(x,y)$
$r_2 : T(x,y) \leftarrow E(x,z), T(z,y).$
Clearly π has persistency number 1, weak persistency number 1 and syntactic persistency number 1.
Program π' consists of the three rules
$r'_1 : T(x,y) \leftarrow E(x,y)$
$r'_2 : T(x,y) \leftarrow E(x,z), E(z,y).$
$r'_3 : T(x,y) \leftarrow E(x,z_1), T(z_1,z_2), E(z_2,y).$
Clearly π' has persistency number 0, weak persistency number 0 and syntactic persistency number 0.
Notice that programs π and π' both have only one IDB predicate, namely T; therefore, for each of these programs, the notions of T-persistency number (resp. T-weak persistency number/ T-syntactic persistency number) and persistency number (resp. weak persistency number/ syntactic persistency number) coincide.
Program π is equivalent to π' which has weak persistency number 0, but π has persistency number 1. Program π' is equivalent to π which has syntactic persistency number 1, but π has weak persistency number 0. □

The previous example shows that P-equivalent programs don't necessarily have the same P-persistency number. In other terms: the P-persistency number does not characterize queries. Therefore, there is need to introduce a stronger notion of persistency number called the *P-persistency number–modulo equivalence*, which is invariant up to program equivalence and thus characterizes queries.

Definition 7.5. The *P-persistency number–modulo equivalence* of a program π is the minimum integer m such that π is P-equivalent to a program of P-persistency number m.
The *persistency number–modulo equivalence* of a query $\mathcal{Q}$ is the P-persistency number–modulo equivalence of any pair (π, P) defining $\mathcal{Q}$.

It follows from lemma 7.2 that it is equivalent to define the P-persistency number–modulo equivalence of π as the minimum integer m such that π is P-equivalent to a program of weak P-persistency number m; it is also equivalent to define the P-persistency number–modulo equivalence of π as the minimum integer m such that π is P-equivalent to a program of syntactic P-persistency number m (this result is stated as a theorem in the Introduction).

In example 7.4 program π has persistency number 1 but its persistency number–modulo equivalence is 0; therefore the transitive closure query has persistency number–modulo equivalence equal to 0.

It is not hard to prove the following proposition.

PROPOSITION 7.6. *1. Let π be a program with syntactic P-persistency number a, weak P-persistency number b, P-persistency number c and P-persistency number–modulo equivalence d; let $arity(P)$ be the arity of IDB predicate symbol P. Then $arity(P) \geq a \geq b \geq c \geq d$.*
2. Let π be a program with syntactic persistency number a, weak persistency number b, persistency number c and persistency number–modulo equivalence d; let $max(arity)_\pi$ be the maximum arity of IDB predicate symbols occurring in π. Then $max(arity)_\pi \geq a \geq b \geq c \geq d$.

8. DECIDABILITY RESULTS

8.1 Is it possible to decide if a given program has syntactic persistency number (resp. weak persistency number) equal to m?

We consider the corresponding decision problem, namely "given a positive integer m does the program have syntactic (resp. weak) persistency number equal to m?" The syntactical notions of persistency numbers are decidable, as shown below.

PROPOSITION 8.1. *The decision problems for the following numbers are decidable:*
a) the syntactic persistency number,
b) the weak persistency number.

PROOF. a) The decidability of the syntactic persistency number follows immediately from its definition.
b) The decidability of the weak persistency number follows from automata-theoretic considerations. Recall from lemma 6.5 the finite non deterministic automaton $\mathcal{A}_{(\pi,P,\sigma)}$ which recognizes the set of all P-traces of π of label σ. It is easy to transform $\mathcal{A}_{(\pi,P,\sigma)}$ into a finite non deterministic automaton $\mathcal{A}_{(\pi,\mathcal{T}_m)}$ which recognizes the set of all traces of π of label some m-pattern-transformation ($\mathcal{T}_m$ is the set of all m-pattern-transformations belonging to the transformation set $\mathcal{T}_\pi$). The automaton $\mathcal{A}_{(\pi,\mathcal{T}_m)}$ differs from the automaton $\mathcal{A}_{(\pi,P,\sigma)}$ in that a state (Q,σ) of $\mathcal{A}_{(\pi,\mathcal{T}_m)}$ is initial for any IDB predicate symbol Q if and only if $\sigma \in \mathcal{T}_m$. Notice that $\mathcal{T}_\pi$ (of definition 6.2) can be written as the finite disjoint union $\mathcal{T}_0 \cup \mathcal{T}_1 \cup ... \cup \mathcal{T}_{max(arity)_\pi}$ where $max(arity)_\pi$ is the maximum arity of IDB predicate symbols occurring in π. Program π has weak persistency number m if and only if the language recognized by the automaton $\mathcal{A}_{(\pi,\mathcal{T}_n)}$ is finite for those n exactly such that $m < n \leq max(arity)_\pi$. The decidability of the weak persistency number follows from the decidability of the finiteness of the language recognized by a finite automaton: recall that the language recognized by a finite automaton is infinite if and only if there exists a sequence s of - not all distinct - states starting from some initial state and ending up to a final state of the automaton (i.e the sequence s contains a loop). □

We prove in an analogous way that the decision problems for (a) the syntactic P-persistency number and (b) the weak P-persistency number are decidable, for any predicate symbol P.

8.2 Is lemma 7.2 constructive?

In this subsection, we first study the decision problem "does a given program have characteristic integer w.r.t.-weak-persistency-number $m \geq 1$?" [7]. This will allow us to show that in lemma 7.2 the proof of the implication 3) $\Longrightarrow$ 4) is constructive.

PROPOSITION 8.2. *Let π be a program. We can compute the characteristic integer of π w.r.t.-weak-persistency-number.*

PROOF. Recall from proposition 8.1 that program π has weak persistency number m if and only if the language recognized by the automaton $\mathcal{A}_{(\pi,\mathcal{T}_n)}$ is finite for those n exactly such that $m < n \leq max(arity)_\pi$; for every such n we can compute the maximum length l_n of the traces recognized by $\mathcal{A}_{(\pi,\mathcal{T}_n)}$ and we can also compute l_{max} the maximum among those integers l_n for $m < n \leq max(arity)_\pi$. Therefore every trace of π labeled with an s-pattern transformation for $s > m$ has length at most l_{max}. Recall from corollary 6.1 that every linear persistent set of size $s > m$ and of length l (occurring in some expansion of π) is determined by some trace of π of length $l-2$ with label an s-pattern transformation. Thus every persistent set of size $> m$ occurring in some expansion of π has length at most $l_{max}+2$, which means that the characteristic integer of π w.r.t.-weak-persistency-number is $l_{max}+3$. □

We can compute, in an analogous way, the characteristic integer w.r.t.-weak-P-persistency-number, for any predicate symbol P.

PROPOSITION 8.3. *(Refinement of lemma 7.2) If program π has weak persistency number m then we can construct a program π', strongly equivalent to π, such that π' has syntactic persistency number m.*

PROOF. Let us compute K_{weak} the characteristic integer of π w.r.t.-weak-persistency-number, according to proposition 8.2. In the proof 4 of lemma 7.2 3) $\Longrightarrow$ 4) we can take $K = K_{weak}$; thus we can construct the equivalent program π' as explained in proof 4. □

The next lemma is an important consequence of lemma 7.2 and proposition 8.3. When we say that a given program property $\mathcal{P}$ is (un)decidable, this means that *the decision problem "given program π, does π have property $\mathcal{P}$?" is (un)decidable.*

LEMMA 8.4. *Let $\mathcal{P}$ be a program property, invariant up to program equivalence (for instance the boundedness property). If $\mathcal{P}$ is decidable for the class of programs with syntactic persistency m then $\mathcal{P}$ is decidable for the class of programs of weak persistency number m.*

PROOF. Let π be a program of weak persistency number m. According to lemma 7.2 3) $\Longrightarrow$ 4) and to proposition 8.3 we can construct a program π' of syntactic persistency number m, which is equivalent to π; π has property $\mathcal{P}$ if and only if π' has property $\mathcal{P}$, and we can decide if π' has or not property $\mathcal{P}$; therefore we can decide if π has or not property $\mathcal{P}$. □

The effective construction of program π' from program π is the key to lemma 8.4. Another equivalent way of expressing the idea of lemma 8.4 is the following: if

[7] Recall definitions 4.1 and 4.2.

property $\mathcal{P}$ is undecidable for the class of programs of weak persistency number m then $\mathcal{P}$ is undecidable for the class of programs of syntactic persistency number m.

8.3 New classes of programs for which boundedness is undecidable

PROPOSITION 8.5. *1. Boundedness is undecidable for programs that have weak persistency number* 0.
2. Boundedness is undecidable for programs that have syntactic persistency number 0.

PROOF. The proof is based on the result that "program boundedness is undecidable for linear binary programs with a single IDB predicate" ([Var88] and theorem 2.3 in the journal version of [HKMV91]). The undecidability reduction in [HKMV91] constructs a program which has weak persistency number 0 and also syntactic persistency number 0. □

PROPOSITION 8.6. *1. Boundedness is undecidable for programs that have weak persistency number* 2.
2. Boundedness is undecidable for programs that have syntactic persistency number 2.

PROOF. The proof is based on the result that "program boundedness is undecidable for programs having two linear recursive rules and one initialization rule" (theorem 5.5 in the journal version of [HKMV91]). The undecidability reduction in [HKMV91] constructs a program which has weak persistency number 2 and also syntactic persistency number 2. □

PROPOSITION 8.7. *1. Boundedness is undecidable for programs with a single IDB predicate and that have weak persistency number* 3.
2. Boundedness is undecidable for programs with a single IDB predicate and that have syntactic persistency number 3.

PROOF. The proof is based on the result that "predicate boundedness is undecidable for programs having one linear recursive rule, one initialization rule and one projection" (theorem 6.3 in the journal version of [HKMV91]). The undecidability reduction in [HKMV91] constructs a program which has weak persistency number 3 and also syntactic persistency number 3. □

The results stated in propositions 8.5(2), 8.6(2) and 8.7(2) - already implicite in [HKMV91] - can also be seen as direct consequences of the corresponding statement (1) in each case, by applying lemma 8.4. This can be considered as a positive hint to further applications of lemma 8.4.

9. CONCLUSION AND OPEN PROBLEMS

In trying to analyze the role of persistencies, we observed that weaker and stronger notions of "persistency numbers" came out. One contribution of the paper is to categorize four different notions of persistencies and to give results about their interrelationship and their decidability. Moreover we studied the decidability of the boundedness problem on classes of programs defined according to the value of these persistency numbers; more precisely, we proved that boundedness is undecidable on the class of programs of weak (resp. syntactic) persistency number 0, 2 and

3. These results, together with the results of Marcinkowski [Mar99], give a better focus on the undecidability of boundedness, since they indicate a possible trade-off between (the decreasing of) the number of recursive rules and (the increasing of) the value of the weak persistency number: indeed Marcinkowski proved that program boundedness is undecidable for programs consisting of one linear rule and one initialization rule (theorem 4.10 in [Mar99]) improving previous undecidability results concerning programs with more than one linear recursive rule; we noticed however that this better result of Marcinkowski comes at a cost: that of increasing the weak persistency number of the program (indeed the - one linear rule and one initialization rule - programs $\pi_{Mar,p}$, constructed in the undecidability reductions of Marcinkowski have a high weak persistency number $p+3$ where p is a parameter defined according to definition 4.1 in [Mar99]); it is natural to consider as persistencies the constants appearing in programs and our formal definitions can be easily adapted to that case.

We briefly refer below to some subsequent work, related to the results of the present paper, and done after the present paper was submitted. We also mention some further research directions that are worth pursuing in the future.

—In proposition 8.1 we have given an algorithm - based on our automata of section 6 - for evaluating the weak persistency number of a Datalog program. We give a rough estimate of the complexity of this algorithm: first the transformation of a program π into a normal program π' increases its size at most exponentially, and can be done in exponential time (in the size of π); then in Proof 8.1 the automaton $\mathcal{A}_{(\pi',\mathcal{T}_m)}$ can then be constructed in time exponential in the size of π', and its size is at most exponential in the size of π'; and the finiteness problem can be solved in time polynomial in the size of $\mathcal{A}_{(\pi',\mathcal{T}_m)}$. Thus, the weak persistency number of π can be determined in exponential time, in the size of π. In subsequent work, we have determined the exact complexity of computing the weak persistency number as well as the exact complexity of computing the characteristic integer w.r.t.-weak-persistency-number, both for normal programs and for general form programs [CFS03].

—In subsequent work we have shown that determining the persistency number and the persistency number-modulo-equivalence are both undecidable [CF02].

—Lemma 8.4 says that for every integer m and for every program property $\mathcal{P}$ which is invariant up to program equivalence, the decidability of $\mathcal{P}$ on the class of programs of syntactic persistency m implies the decidability of $\mathcal{P}$ on the class of programs of weak persistency m. We are expecting that our lemma 8.4 will produce decidability results for some properties other than boundedness, but still invariant up to program equivalence.

—The formal results of the paper should lead to a deeper understanding of the behavior of Datalog programs. In particular, the fourth persistency number - "persistency number-modulo equivalence" - is the most deeply semantical one since it characterizes queries. It follows from our main lemma 7.2 that for every query $\mathcal{Q}$ having persistency number-modulo equivalence m, there exists (at

least) one program π expressing $\mathcal{Q}$ such that the four numbers of π coincide and are equal to m. We expect that further research on the persistency numbers (especially on the persistency number-modulo equivalence) will help to solve expressibility questions concerning Datalog.

ACKNOWLEDGMENTS

We would like to thank one anonymous referee for his useful remarks that helped us to improve the paper.

REFERENCES

Abiteboul, S., Hull, R., Vianu, V. *Foundations of Databases.* Addison-Wesley Publishing Company, 1995.

Abiteboul, S., Vianu, V. Expressive Power of Query Languages. In J.D. Ullman, editor, *Theoretical Studies in Computer Science*, Academic Press, 1991, pp. 207-251.

Afrati, F. Bounded arity Datalog($\neq$) queries on graphs. *Journal of Computer and Systems Sciences*, vol 55 on ACM Conference "Principles of Database Systems,1997.

Afrati, F., Cosmadakis, S. Expressiveness of Restricted Recursive Queries. In *Proc. 21st ACM Symp. on Theory of Computing*, 1989, pp. 113-126.

Ceri, S., Gottlob, G., Tanca, L. *Logic Programming and Databases.* Springer-Verlag, 1990.

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

Cosmadakis, S.S. On the First-Order Expressibility of Recursive Queries. In *Proc. 8th ACM Symp. on Principles of Database Systems*, March 1989, pp. 311–323.

Cosmadakis, S.S. Inherent Complexity of Recursive Queries. In *JCSS 64(3)*, 2002, pp. 466–495.

Cosmadakis, S.S., Foustoucos, E. Undecidability results concerning Datalog programs and their persistency numbers. Technical Report No.TR2002/12/03, Research Academic Computer Technology Institute (CTI), Patras, Greece, December 2002.

Cosmadakis, S.S., Foustoucos, E., Sidiropoulos, A. Undecidability and intractability results concerning Datalog programs and their persistency numbers. Manuscript, 2003.

Courcelle, B. Graph rewriting: an algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, Elsevier, Amsterdam, 1990, pp. 193-242.

Gaifman, H., Mairson, H., Sagiv, Y., Vardi M.Y. Undecidable optimization problems for database logic programs. In *Proc. 2nd IEEE Symp. on Logic in Computer Science*, Ithaca, 1987, pp. 106–115. Also *Journal of the ACM*, volume 40 no. 3, 1993, pp. 683-713.

Gottlob, G., Leone, N., Scarcello, F. Hypertree decompositions: a survey. In *Mathematical Foundations of Computer Science* (MFCS 2001), LNCS 2136 pp. 37-57.

Grohe, M. Bounded–arity hierarchies in fixed–point logics. In E. Boerger, Y. Gurevich and K. Meinke, editors, *CSL'93: Computer Science Logic,* volume 832 of LNCS, Springer 1994, pp. 150–164.

Grohe, M., Marino, J. Definability and Descriptive Complexity on Databases of Bounded Tree-Width. In C. Beeri and P. Buneman, editors, *Proc. 7th Int. Conf. on Database Theory,* volume 1540 of LNCS, Springer 1999, pp. 70–82.

Grohe, M., Schwentick, T., Segoufin, L. When is the evaluation of conjunctive queries tractable ? *Proc. 32nd ACM Symposium on Theory of Computing (STOC'01)*, 2001, pp. 657–666.

Hillebrand, G. G., Kanellakis, P.C., Mairson, H.G., Vardi, M.Y. Undecidable Boundedness Problems for Datalog programs. In *Proc. 10th ACM Symp. on Principles of Database Systems*, 1991, pp. 1-12. Also *Journal of Logic Programming*, volume 25 no. 2, 1995, pp. 163-190.

Marcinkowski, J. Achilles, Turtle, and undecidable boundedness problems for small Datalog programs. *SIAM Journal of Computation 29(1)*, 1999, pp. 231-257.

Kolaitis, Ph., Vardi, M. Conjunctive-Query Containment and Constraint Satisfaction. In *Proc. 17th ACM Symp. on Principles of Database Systems*, 1998.

Papadimitriou, C. H., Yannakakis, M. On the complexity of database queries. In *Proc. 16th ACM Symp. on Principles of Database Systems*, 1997. Full version in JCSS, Vol. 58, 1999.

Ullman, J.D. *Database and Knowledge-Base Systems, Volumes I and II.* Computer Science Press, 1989.

Vardi, M.Y. Decidability and Undecidability Results for Boundedness of Linear Recursive Queries. In *Proc. 7th ACM Symp. on Principles of Database Systems*, Austin, 1988, pp. 341–351.

Received November 2002; revised July 2003; accepted January 2004