Arity Hierarchies for Fixed-Point Logics

Foto N. Afrati*
National Technical University of Athens
afrati@cs.ece.ntua.gr

Abstract

Fixed-point logic is the logic of inductively definable relations over structures of a specific signature. The inductive definitions are given via expressions of the first order logic or fragments of it. A natural question is whether we can increase the expressive power of the logic by allowing larger arities of the inductively defined relations. A number of results appeared lately that essentially solved that problem for the most important variants of fixed-point logic, by showing that the arity hierarchy formed is strict even in the case when the inductive definitions are over structures containing a single binary relation (i.e., graphs). We review these results and comment on the techniques that were used to prove them. In fact, for the query language Datalog, a strict hierarchy is also shown with respect to the *persistency*–arity of the inductively defined relations.

1 Introduction

One of the main objects of investigation in classical model theory has been the study of those relations which are explicitely definable in the language of first order logic. It has been observed, though, that first order logic is limited in expressive power (recall that the transitive closure relation is not expressible in this language [AU79]), essentially because it lacks recursion. A natural way to increase the expressive power of first order logic is by augmenting it with a fixed-point operator. In fact, there are various ways to define a fixed-point operator, resulting in several well-known fixed-point logics (partial fixed-point logic, least fixed-point logic).

Since fixed-point logics have played a major role in relational database theory, being viewed and studied as query languages, we adopt some of our terminology and examples from this discipline rather than from classical logic (it is more demonstrative and sometimes more familiar). A *query* is understood as a function from databases (=relational structures) to databases. A boolean query, thus, is a function from databases to $\{True, False\}$; it can be viewed, therefore, as describing a property of databases.

A well known fragment of least fixed-point logic in the database community is the query language *Datalog* [Ull89]. An example of a Datalog program which, moreover, is not expressible in first-order languages is the following:

$$\begin{aligned} path(x,y) &\ :-\ \ edge(x,y) \\ path(x,y) &\ :-\ \ path(x,z), path(z,y). \end{aligned}$$

*Research partially supported by a grant of the Greek General Secretariat of Research and Technology.

In this example we think of the database as a directed graph. The above program consists of two *rules*, one *recursive* rule and one *initialization* rule. The relation symbol *edge* is called the *extensional database predicate* (EDB), representing basic facts stored in the database. The *intensional database* (IDB) consists of the relational symbol *path* which represents facts deduced from the database via the rules of the program. The first is an *initialization* rule, and the second is a *recursive* rule. We refer to the left-hand side of a rule as the *body* of the rule and to the right-hand side of a rule as the *head* of the rule. Here we deduce $path(u, v)$ iff there is a path from vertex u to vertex v. Thus the above program computes the *transitive closure* of the directed graph represented by the database.

More formally, Datalog programs are given the following fixed-point semantics. Let $\mathcal{D}$ be a database structure, 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. If we consider P the *goal* predicate, then π expresses a query $Q_{\pi,P}$, where

$$Q_{\pi,P}(\mathcal{D}) = \bigcup_{k\geq 0} Q^k_{\pi,P}(\mathcal{D})$$

We will sometimes write Q_π instead of $Q_{\pi,P}$.

The above definition also gives a particular algorithm to compute $Q_{\pi,P}$: initialize the IDB predicates to be empty, and repeatedly apply the rules to add tuples to the IDB predicates, until no new tuples can be added. This is known as *bottom-up evaluation.*

Datalog essentially is equal, in expressive power, to the existential least fixed-point logic (EFP). Consider the following existential positive formula:

$$\varphi(x, y, path) = edge(x, y) \vee (\exists z)(path(x, z) \wedge path(z, y))$$

$\varphi^\infty(x, y, path)$ defines the same query as the Datalog program above, and in fact the computation procedure that is implied is the same in both cases. I.e., applying the least fixed point operator on the above first order formula is equivalent to the bottom-up evaluation: initialize the IDB predicates to be empty, and repeatedly evaluate the formula (on the new structure enhanced with the so far added tuples to the IDB predicates) to add tuples to the IDB predicates, until no new tuples can be added.

We get *least fixed-point logic* by allowing any positive formula of first order logic for the inductive definitions and applying the least fixed point operator; it is guaranteed that there exists a least fixed-point, because a positive formula is monotone (i.e., if it is true on a structure it is also true any extension of the structure). In the case where negation symbols appear in the formula, then we define, accordingly a fixed point operator, but, now, we get a *partial fixed-point operator*, defining the *partial fixed-point logic* (see Mo74 for detailed definitions).

Fixed-point logic (and especially simpler fragments of it) has gained a lot of attention lately as regard expressiveness and computational complexity issues. A number of interesting results are known that relate expressiveness of fixed-point logics to computational complexity issues. We know that least fixed-point logic equals partial fixed-point logic if and only if PTIME equals PSPACE [AV91]. On ordered databases, Datalog with negation on EDB predicates expresses *all* polynomial time queries [Pap85, Var82]. Without this restriction (i.e., of the order) on the input databases, though, there are simple polynomial time queries not expressible in Datalog: an example is the query that asks if the domain has even cardinality. On the other end, there are queries that are complete for polynomial time and are expressible in Datalog, such as the *path systems query* (this is the query in example 2.1) [Coo74].

We define the *width* of a fixed-point query to be the maximum arity of the IDB predicates. A natural question is whether the expressive power of the logic increases if we allow arbitrary width. For Datalog, this question is answered in [AC89] for a class of queries on graphs. For least fixed-point and partial fixed-point logics it is answered in [Gro94] for the same class of queries. The proof techniques, though, are different; the first result is proven by homomorphic techniques, whereas the latter is proven via special pebble games which are defined in a more subtle way to capture the arity of the IDB predicates.

A question that relates to the arity of the predicates concerns persistencies, i.e., variables in the definition that appear both in the body and in the head of the definition. It has been observed for a while that persistencies play a major role in Datalog programs; e.g., while the problem of whether a program is bounded is undecidable in the general case (and in fact for very simple programs), it is shown decidable in the case there are no persistencies [CGKV88]. In that respect, a related question is whether the number of persistencies affect the expressibility of a Datalog query regardless of the width. This is answered in [Af94] for a simple and interesting [KV90] class of queries on graphs.

The rest of the paper is structured as follows: In section 2, we present another angle of viewing Datalog expansions which is implicit in [ACPS93]. In section 3, we review basic concepts and definitions and define the persistency arity of a program. In section 4, we discuss the results on arity-hierarchy, and in section 5 we discuss the results on persistency-arity hierarchy.

2 Expansions viewed as bounded tree-width graphs

Given a Datalog($\neq$) program π we may unwind its recursive rules to some finite depth, to obtain an *expansion* of π. Each such expansion may be viewed as a first-order formula over $\exists, \wedge$; inequalities will also appear, if they occur in π. The query Q_π is expressed as the infinitary disjunction of the set of expansions of π.

Example 2.1: Let π be the following Datalog program:

$$\begin{array}{lll} P(x) & :- & R(x) \\ P(x) & :- & S(x,y,z), P(y), P(z) \end{array}$$

The following are four expansions of π:

$$\begin{array}{llll} e_1: & R(x) & & \\ e_2: & S(x,y,z) & \wedge & R(y) \wedge R(z) \\ e_3: & S(x,y,z) & \wedge & (S(y,y_1,z_1) \wedge R(y_1) \wedge R(z_1)) \\ & & \wedge & R(z) \\ e_4: & S(x,y,z) & \wedge & R(y) \\ & & \wedge & (S(z,y_2,z_2) \wedge R(y_2) \wedge R(z_2)) \end{array}$$

In each expansion x is free, and the rest of the variables are existentially quantified. ∎

Definition 2.2: A *vocabulary* is a set $\sigma = \{R'_1, \ldots, R'_m\}$ of relational symbols of specified arities. A *structure* $\mathbf{B} = (B, R_1, \ldots, R_m)$ over σ consists of a finite set B and relations on B, such that the arity of B_i is equal to the arity of B'_i, for $i = 1, \ldots, m$. B is called the *domain* of structure $\mathbf{B}$.

A *database* is a finite structure.

A *graph* $G = (V, E)$ is a structure over a vocabulary consisting of a single binary symbol. The universe V is the set of *nodes* of G and the binary relation E is the set of *edges* of G. ∎

An expansion of a Datalog program may also be viewed as a database (i.e. finite structure), with domain the set of variables in the expansion, and tuples corresponding to the atomic formulas. Accordingly, if the program is over binary predicates only, an expansion can be viewed as a labelled directed graph. In this case, it is not difficult to observe that the family of all expansion of a program has some special well known graph-theoretic properties, i.e., it is a family of bounded treewidth graphs. Here are the appropriate graph-theoretic definitions [ACPS93, Bo93]:

Definition 2.3: [ACPS93] A $k+1$-clique is a *k-tree* and any *k-tree* with $n > k+1$ vertices can be constructed from a k-tree with $n-1$ vertices by adding a vertex adjacent to all vertices of a k-clique subgraph of that k-tree. Subgraphs of k-trees are called *partial k-trees.*

A graph G is of treewidth at most k if a family $\{X_n\}_{n \in N}$ of vertex subsets of G can be arranged as nodes in a tree T so that a) those nodes containing a given vertex (of G) induce a subtree of T (i.e., a connected subgraph of T), b) every pair of adjacent vertices share membership in some X_n and c) the size of all X_ns is $\leq k+1$. Such an arrangement is called a *tree–decomposition of width at most k.*

The class of partial k-trees is exactly the class of graphs of treewidth at most k. ∎

Consider a graph $G(V, E)$; a set of vertices, $S \subset V$, is a *separator* in G if the removal of S and the edges incident to its vertices disconnects G. The X_ns in the above definition are separators of the graph.

Observe now that all expansions of a Datalog program consist a family of graphs of treewidth at most s, where s is the maximum size of the rule in the program; therefore, they can be viewed as a family of graphs of *bounded treewidth.* Moreover the way an expansion is formed (by unfolding the rules of the program) induces in a straightforward way a *standard* tree decomposition with an associated tree T (in fact it is a rooted tree).

Definition 2.4: Consider an expansion, viewed as a graph $G(V, E)$ of treewidth $< s$, and the standard tree decomposition of the expansion; let T be the associated tree. For any pair of adjacent nodes in T, we define the intersection of the associated X_ns to be a *boundary* B_i of the expansion. It is convenient to view these boundaries as labels on the nodes of the tree T, obtaining, thus, the *trace tree*, T_{tr} of the expansion. (In particular, in every B_i, there are two nodes of T involved; choose to label with B_i the one closest to the root.)

Note that any tree decomposition of a graph induces a trace tree, regardless of the fact of being the expansion of some Datalog program or not. ∎

Note that each boundary is a separator set of the graph.

In general, a subset of a specific boundary might appear in other boundaries as well, all of them forming a connected component of the tree. If this is the case, we call this subset a *persistency set*; to denote the size of this subset as well, we refer ot it as an m-persistency set.

Definition 2.5: Consider any infinite family of graphs of treewidth $< k$ and the associated trace trees. If, for any positive integer l, there is a trace tree in the family with an m-persistency set appearing in more that l boundaries, then we say that this family of graphs *contains m persistencies.*

Since we can view the family of expansion of a Datalog program as a family of graphs of bounded treewidth, we define accordingly the notion of a *program containing m persistencies.* ∎

The association of bounded treewidth graphs to several logics has been observed by several authors lately, see e.g., [ACPS93].

3 Homomorphisms vs. Pebble Games

3.1 Homomorphisms

The following are well known facts for Datalog expansions:

Definition 3.1: A *homomorphism* from database $\mathcal{D}(D, r_1, r_2, \ldots)$ to database $\mathcal{D}'(D', r'_1, r'_2, \ldots)$ is a function $h : D \to D'$ such that, if $(a_1, \ldots, a_{m_i}) \in r_i$, then $(h(a_1), \ldots, h(a_{m_i})) \in r'_i$.

Query Q is *preserved under homomorphisms* if, whenever $\mathcal{D}, \mathcal{D}', h$ are as above, we have if $(a_1, \ldots, a_m) \in Q(\mathcal{D})$, then $(h(a_1), \ldots, h(a_m)) \in Q(\mathcal{D}')$. ∎

Lemma 3.2: *Datalog queries are preserved under homomorphisms.*

Without loss of generality and to simplify notation, we can think in what follows only in terms of Boolean queries. Thus definitions and concepts given below are simplified; extending them to all queries is mostly straightforward but tedious.

Definition 3.3: An expansion e defines a query Q_e as follows: for every database $\mathcal{D}$, $Q_e(\mathcal{D}) = True$ if there is a homomorphism h from e to $\mathcal{D}$, otherwise $Q_e(\mathcal{D}) = False$

In the case that $Q_e(\mathcal{D}) = True$, we say that *expansion e accepts the database.*

We say that a program π *accepts a database* iff there is an expansion of π which accepts the database. ∎

We can extend Datalog by allowing a built-in binary prediacate INEQ to denote inequality with the usual semantics, obtaining, thus, Datalog($\neq$).

The above extends to Datalog($\neq$). Rule bodies include a binary relation INEQ corresponding to inequalities $z \neq z'$. We additionally require that homomorphism h satisfies

if $(a, a') \in$ INEQ, then $h(a) \neq h(a')$.

Definition 3.4: If a Datalog π program is equivalent (i.e., expresses exactly the same queries) to some Datalog program with finitely many expansions, then we say that program π is *bounded* (i.e., the recursion depth is bounded). ∎

Consider the family of all expansions of a bounded program; in general, it might be non-finite, but only a finite number of expansions "give information on the expressiveness of this program".

Definition 3.5: More formally, consider any family F of expansions of a given program (i.e., a subset of all expansions). We define an expansion, e in F, to be *useful within the family* if and only if there is no shorter expansion in the family that can be homomorphically mapped on e. ∎

Definition 3.6: We define a family of expansions to be a *minimally useful family* if and only if all expansions are useful within the family. ∎

Thus, we can rephrase the definition of boundedness: A program is bounded if and only if all minimally useful families of expansions contain a bounded number of elements.

In that respect, we say that a Datalog program contains m *strong* persistencies if and only if all minimally useful families of expansions contain m persistencies.

Definition 3.7: The *persistency number* or *persistency arity* of a Datalog program, π, is defined to be the maximum m such that π contains m strong persistencies. ∎

For more details on expansions see [AC89, ACY91] and on persistency number see [Af94].

Following the above, a technique to prove non-expressibility results for Datalog($\neq$) is: Suppose we want to prove that query Q is not expressible in a fragment of Datalog. Consider a sufficiently large (with respect to the size of the program, i.e., the treewidth of the expansion family of this program) expansion e that accepts a database that belongs to the query Q. Then construct a homomorphism which maps e on a database that does not belong to the query. The property of expansions being a family of graphs of bounded treewidth, is used heavily in the construction of such homomorphisms [Af94].

3.2 Pebble Games

Following the discussion on expansions (now viewed again as conjunctive queries), it can be easily seen that Datalog queries are expressed as an infinitary disjunction of conjunctions (each conjunct being an expansion of the program). These are formulas of an extension of existential first order logic allowing arbitrarily long disjunctions and conjunctions, namely of *existential infinitary logic with finitely many variables.* Accordingly, any query in partial fixed-point logic can be expressed by a formula of infinitary logic with finitely many variables. Some more formal definitions below.

$L_{\infty\omega}$ is the extension of first order logic with arbitrary disjunctions and conjunctions. On finite structures, $L_{\infty\omega}$ can express *every* query. $L^k_{\infty\omega}$ is the collection of $L_{\infty\omega}$-formulas with at most k distinct variables. *Infinitary logic with finitely many variables* is defined to be: $L^\omega_{\infty\omega} = \bigcup_{k\geq 1} L^k_{\infty\omega}$.

A well known technique to prove non-expressibility results for $L^\omega_{\infty\omega}$, is some kind of Ehrenfeucht–Fraisse games played with a finite number of pebbles. Following the above discussion, a non-expressibility result for $L^\omega_{\infty\omega}$ yields an non-expressibilty result for partial fixed-point logic. These games are played on two structures of the same signature by two players, the spoiler and the duplicator. The duplicator wins if, in every move, can duplicate the move of the spoiler (keeping the pebbled substructures isomorphic); otherwise the spoiler wins. Several interesting results were obtained by this technique, see e.g., [KV90], [Gro94].

4 Arity Hierarchies

In this section, we shall discuss results which prove that by allowing arbitrary large width in any of the known fragments of fixed-point logic, we increase the expressive power, or in other words the arity hierarchy is strict. The question whether the arity hierarchy is strict for various fixed-point logics has been raised and partially addressed in several works, including [CH85] and [DM89]. It is solved in [AC89] and [Af94] for Datalog queries and in [Gro94] for least fixed-point and partial fixed-point queries.

The following is a natural generalization of *transitive closure*:

Definition 4.1: Let D be a set, and let r be a relation over D of width $2q$, where $q \geq 1$. A *path of width q from u to v*, where $u, v \in D$, is a sequence of elements $a^i_m \in D$, $i = 0, \dots, n$, $m = 1, \dots, q$, such that $u = a^0_1$, $v = a^n_1$, and $(a^i_1, \dots, a^i_q, a^{i+1}_1, \dots, a^{i+1}_q) \in r$, $i = 0, \dots, n-1$. ∎

Definition 4.2: For $q \geq 1$, $Path_q$ is a boolean query which, on a database $\mathcal{D} = (D, r, s, t)$ of type $(2q, 1, 1)$, returns *true* iff there exist elements $u, v \in D$ such that $u \in s$, $v \in t$, and there is a path of width q from u to v. ∎

The query $Path_q$ can be computed by the linear Datalog program below; the EDB predicates R, S, T denote the relations r, s, t respectively, and the goal IDB predicate is I.

$$\begin{array}{rcl} I() & :- & P(x_1, \dots, x_q), T(x_1) \\ P(x_1, \dots, x_q) & :- & P(y_1, \dots, y_q), R(y_1, \dots, y_q, x_1, \dots, x_q) \\ P(x_1, \dots, x_q) & :- & S(x_1) \end{array}$$

This program uses an IDB predicate P of width q; it is shown by homomorphic techniques in [AC89] that this is the smallest possible.

Theorem 4.3: *[AC89] The query $Path_q$ cannot be computed by any Datalog program of IDB-width less than q.*

For least-fixed point logic and partial fixed-point logic, the strictness of the arity hierarchy is proved using the same query [Gro94]. To prove the result, though, in [Gro94], a variant of the well known pebble game is defined to capture arity as well; this pebble game yields the following theorem:

Theorem 4.4: *[Gro94] Let $r, k \geq 0$ and $\mathcal{U}$ and $\mathcal{B}$ be two structures such that the duplicator has a winning strategy for the k-ary r-pebble game on $\mathcal{U}$, $\mathcal{B}$. Then the same sentences of partial fixed-point logic of width k and quantifier-rank $\leq r$ hold in $\mathcal{U}$ and $\mathcal{B}$.*

It is shown, thus, that this query cannot be expressed by any formula of partial fixed-point logic either:

Theorem 4.5: *[Gro94] The query $Path_q$ cannot be computed by a partial fixed-point formula of IDB-width less than q.*

To proof of this result requires a fine construction of certain families of structures which are somehow "homogeneous in the $k-1$-tuples" (thus fixed-point formulas of width $k-1$ become ineffective). On these structures, it is shown that the duplicator has a winning strategy.

To formulate the query $Path_q$, there were used relations whose width increases linearly with q. These "wide" relations are not necessary: it is possible to encode a relation of width $\Theta(q)$ as a *clique* of size $\Theta(q)$, and thus get by with relations of width at most two (in fact, a single binary relation suffices) [AC89, Gro94]. The proofs of the theorems go through, with appropriate modifications. Unary relations do not work, though: it can be shown that, if all the EDB predicates are unary, then every Datalog program is equivalent to a *recursion-free* Datalog program.

If we consider Datalog with inequalities, then the arity hierarchy is also strict as it is proven in [Af94]. In fact, this result falls from a stronger result proven there which is presented in the next section.

5 Persistency-arity Hierarchy

The strictness of the persistency-arity hierarchy is formal proof that persistencies play a fundamental role in the expressive power of Datalog($\neq$).

The following is a simple query which belongs in a more general class of queries which has been used to demostrate several interesting results [KV90, FHW80]: *Given a graph and a positive integer m, are there m disjoint paths, from node s to node t?* They belong to the more general class of queries, the H-subgraph homeomorphism queries:

Definition 5.1: Given directed graphs H and G and a one-to-one mapping map of the nodes of H into the nodes of G, we say that H is *homeomorphic* to a subgraph of G if there exists a mapping from the edges of H to pairwise simple node-disjoint paths in G such that the edge (s,t) is mapped to a simple path from $map(s)$ to $map(t)$. The *H-subgraph homeomorphism* query, for a fixed graph H, is to determine for an input graph G and a node mapping map whether H is homeomorphic to a subgraph of G. ∎

Kolaitis and Vardi ([KV90]) proved that an H-subgraph homeomorphism query is expressible in Datalog($\neq$) iff H belongs to the following class $\mathcal{C}$: $\mathcal{C}$ is the collection of all digraphs with a special node called the *root* and with the property that either the root is the head of every arc or the root is the tail of every arc in H. Using the max-flow min-cut theorem and the fact that the number of arcs in H is fixed, they demonstrated a program that expresses this query in Datalog($\neq$). It is shown that the persistency number of their program is equal to the number of edges in the graph H:

Theorem 5.2: *[Af94] The H-subgraph homeomorphism query on the class $\mathcal{C}$ of subgraphs H is expressible by a Datalog($\neq$) program of persistency number equal to the number of edges in H.*

In the following, we give an example of a query belonging in this class together with the program that expresses it and some discussion on expansions being viewed as bounded treewidth graphs.

Example 5.3: Consider the following program π_2 [KV90], where the predicate Q expresses the query:*Given a digraph, are there two disjoint paths, one from node s to node s_1 and the other from node s to node s_2?*

$$\begin{array}{rcl} T(s,s_1,t_1) & :- & E(s,s_1), s \neq t_1, s_1 \neq t_1. \\ T(s,s_1,t_1) & :- & T(s,w,t_1), E(w,s_1), s_1 \neq t_1. \\ Q(s,s_1,w_2) & :- & E(s,w_2), T(s,s_1,w_2). \\ Q(s,s_1,w_2) & :- & Q(s,s_1,w), E(w,w_2), T(s,s_1,w_2). \end{array}$$

Consider the following expansion e of π_2: Unfold the last rule with respect to predicate Q four times; then, apply the third rule on the single occurence of predicate Q; then, for each occurence of the predicate T, unfold the second rule six times; finally apply the first rule in all occurences of T. View this expansion as a graph and observe that it can be homomorphically mapped on the graph displayed in the top of Figure 1, therefore, this graph is accepted by expansion e and hence by program π_2.

Consider the above expansion and its tree decomposition; then the associated trace tree is displayed in the bottom of Figure 1. To make it more illustrative, on each node of the tree, we placed the associated predicate symbol too.

It so happens, in this example, that the trace tree displays *all* variables used in expansion e. Therefore, we can give the homomorphism h, which maps e onto the graph in the top. It is: $h(a) = A$, $h(b) = B$, $h(c) = C$, $h(c_i) = C_i$ for $i = 1,2,3,4$ and $h(b_{ji}) = B_i$ for all $i = 1,2,3,4,5$ and $j = 0,1,2,3,4$.

Observe that this trace tree contains a 1-persistency set, i.e., $\{a\}$; it also contains several 2-persistency sets namely $\{a,c_3\}$ (shaded subtree), $\{a,c\}$, and $\{a,c_i\}$ for $i = 1,2,4$.

Finally, it is not too difficult to prove that the above program is of persistency number two: Observe that a) in the example, the set $\{a,c\}$ appears in the label of 7 nodes of the trace tree and b) if we consider our example expansion in the form before applying the initialization rule (the first in the above program) and instead we go on with unfolding several times the second rule with respect to each occurence of T, then we get an expansion and an associated trace tree with the set $\{a,c\}$ appearing in the labels of any number of nodes we wish. ∎

The proof of the following theorem depends heavily on the expansions being viewed as bounded treewidth graphs, whereas the last theorem is a rather easy consequence of it.

Theorem 5.4: *[Af94] The H-subgraph homeomorphism query, where the number of arcs in H is equal to q cannot be computed by any Datalog($\neq$) program of persistency number less than $q-1$.*

Theorem 5.5: *[Af94] The H-subgraph homeomorphism query, where the number of arcs in H is equal to q cannot be computed by any Datalog($\neq$) program of IDB-width less than q.*

References

[Af94] Afrati, F. Bounded Arity of Datalog($\neq$) Queries on Graphs In *Proc. 13th ACM Symp. on Principles of Database Systems*, 1994, also in *Journal of Computer and System Sciences* special issue on PODS'94, to appear.

[AC89] Afrati, F., Cosmadakis, S. Expressiveness of Restricted Recursive Queries. In *Proc. 21st ACM Symp. on Theory of Computing*, 1989, pp. 113-126.

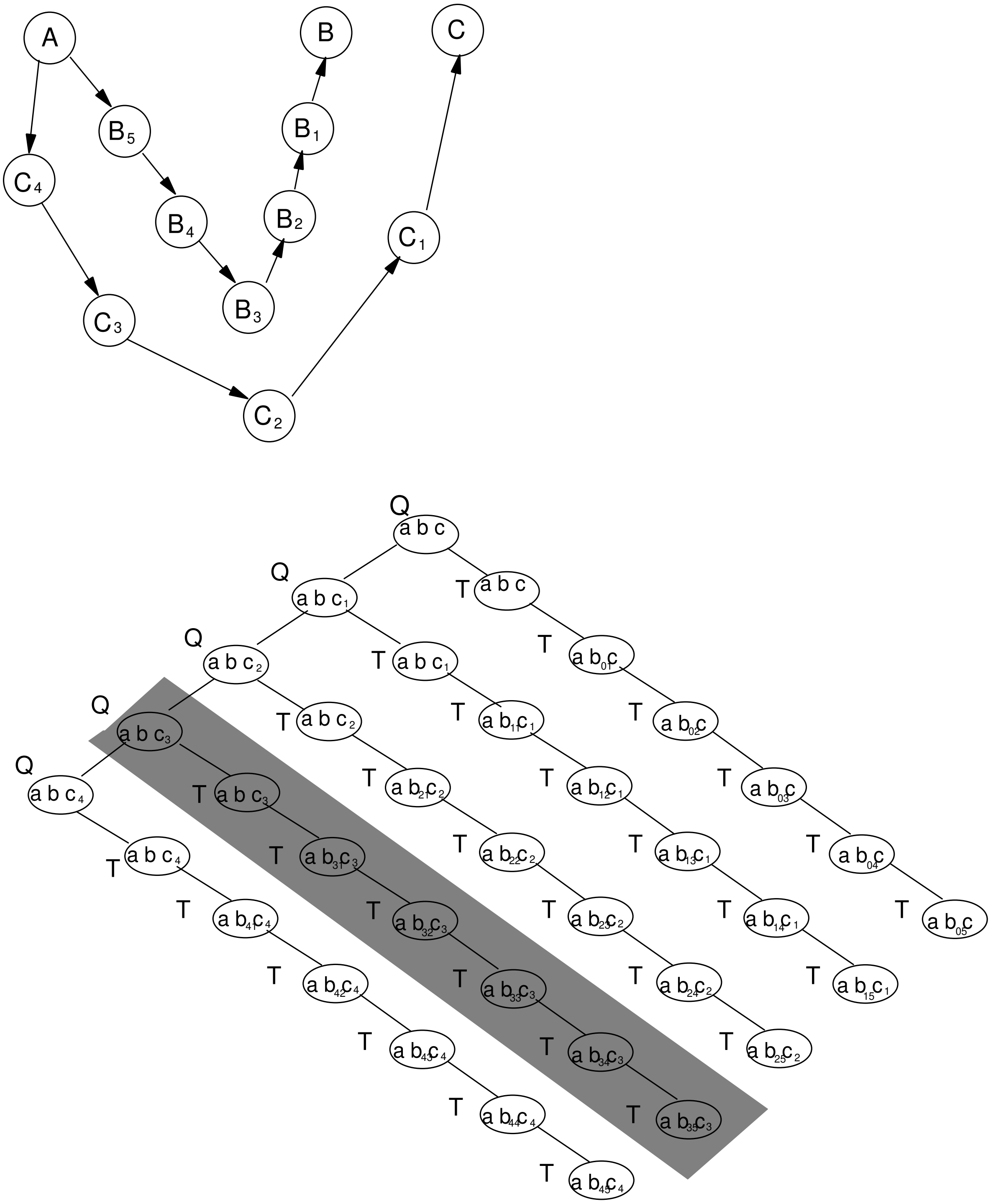

Figure 1: Trace of a skeleton tree (bottom); and a database (top) on which the corresponding expansion can be mapped.

[ACPS93] Arnborg, S., Courcelle, B., Proskurowski, A., Seese, D., An Algebraic Theory of Graph Reduction *Journal of the Association for Computing Machinery*, Vol. 40, No. 5, Nov. 1993, pp. 1134–1164.

[ACY91] Afrati, F., Cosmadakis, S., Yannakakis, M. On Datalog vs. Polynomial Time. In *Proc. 10th ACM Symp. on Principles of Database Systems*, 1991, pp. 113-126, also in *Journal of Computer and System Sciences*, special issue on PODS'91, Vol. 51, 1995, pp. 177–196.

[AV91] Abiteboul, S., Vianu, V. Generic Computation and its Complexity. In *Proc. 23rd ACM Symposium on Theory of Computing*, pp. 209–219, 1991

[AU79] Aho, V.H., Ullman, J.D.: Universality of data retrieval languages. *Proc. 6th ACM Symp. on Principles of Programming Languages*, 1979, pp. 110–117.

[Bo93] Bodlaender, H. L., A Tourist Guide trough Treewidth. Acta Cybernetica, Vol. 11, No 1–2, 1993, pp. 1–21.

[CGKV88] 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.

[CH85] Chandra, A.K., Harel, D. Horn-clause queries and generalizations. *J. Logic Programming* 1(1985), pp. 1–15.

[Coo74] Cook, S.A. An observation on time-storage trade off. *J. Computer and System Sciences* 9 (1974), pp. 308–316.

[DM89] Dublish, P., Maheshwari, S. Expressibility of Bounded Arity Fixed-point Queries. In *Proc. 8th ACM Symp. on Principles of Database Systems*, (1989), pp. 324–335.

[FHW80] Fortune, S., Hopcroft, J., Wyllie, J. The Directed Homeomorphism Problem. *Theoretical Computer Science*,10, 1980, pp. 111-121.

[Gro94] Grohe, M. Arity Hierarchies. *Annals of Pure and Applied Logic* 82, pp. 103–163, 1996.

[KV90] Kolaitis, Ph., Vardi, M. On the Expressive Power of Datalog: Tools and a Case Study. In *Proc. 9st ACM Symp. on Principles of Database Systems*, 1990, pp. 61-71, also in *Journal of Computer and System Sciences*, special issue on PODS'90, vol. 51, 1995, pp. 110–134.

[Mo74] Moschovakis, Y. *Elementary Induction on Abstract Structures.* North Holland, 1974.

[Pap85] Papadimitriou, C.H. A Note on the Expressive Power of Prolog. *Bulletin of EATCS*, 26, 1985, pp. 21-23.

[Ull89] Ullman, J.D. *Database and Knowledge-Base Systems, Volumes I and II.* Computer Science Press, 1989.

[Var82] Vardi, M.Y. The complexity of relational query languages. In *Proc. 14th ACM Symp. on Theory of Computing*, San Francisco, 1982, pp. 137–146.