On the Power of Aggregation in Relational Query Languages*

Leonid Libkin[1] **Limsoon Wong**[2]

[1] Bell Laboratories/Lucent Technologies, 600 Mountain Avenue, Murray Hill, NJ 07974, USA, Email: `libkin@research.bell-labs.com`
[2] BioInformatics Center & Institute of Systems Science, Singapore 119597, Email: `limsoon@iss.nus.sg`

1 Summary

It is a folk result that relational algebra or calculus extended with aggregate functions cannot compute the transitive closure. However, proving folk results is sometimes a nontrivial task. In this paper, we tell the story of the work on expressive power of relational languages with aggregate functions. We also prove by far the most powerful result that describes the expressiveness of such languages. There are four main features of our result that distinguish it from previous ones:

1. It does not rely on any unproven assumptions, such as separation of complexity classes.
2. It establishes a general property of queries definable with the help of aggregate functions. This property can be easily applied to prove many expressiveness bounds.
3. The class of aggregate functions is much larger than any previously considered.
4. The proof is "non-syntactic." That is, it does not depend on a specific syntax chosen for the language with aggregates.

Furthermore, our result gives a very general condition that implies inexpressibility of recursive queries such as the transitive closure in an extension of relational calculus with grouping and aggregation. This extension allows us to use rational arithmetic and operations such as summation and product over a column. So aggregation that exceeds what is allowed by most commercial systems is still not powerful enough to encode recursion mechanisms.

2 Expressive power of aggregation – brief history

It is a well-known result in database theory that the transitive closure query is not expressible in relational algebra and calculus [1]. This was proved by Aho

* Part of this work was done while the first author was visiting Institute of Systems Science.

and Ullman in [2]. A much simpler proof, in the presence of an order relation, was given by Gurevich [13]. Without the order relation, this result follows from many results on the expressive power of first-order logic [7, 9, 10, 11, 17, etc.]

Traditional query languages like SQL extend relational algebra by grouping and aggregation. It was widely believed that such plain SQL cannot express recursive queries like the transitive closure query. However, proving this "folk result" turned out to be quite difficult.

Consens and Mendelzon [5] were the first to provide formal evidence for the "folk theorem". In their ICDT'90 paper, they showed that DLOGSPACE $\neq$ NLOGSPACE would imply that the transitive closure is not definable in an aggregate extension of relational algebra. This follows from DLOGSPACE data complexity of their language, and NLOGSPACE-completeness of the transitive closure. Notably, their result cannot say anything about nontrivial recursive queries complete for DLOGSPACE, such as *deterministic* transitive closure [14]. This perhaps can be remedied by reducing the data complexity to, say, NC^1, and making a different assumption like $\mathrm{NC}^1 \neq$ DLOGSPACE. Nevertheless, their result does demonstrate that the assumed expressivity bounds on languages with aggregates are likely to be true.

It remained open though whether expressivity bounds for languages with aggregates can be proved without assuming separation of complexity classes. First such result appeared in 1994, when we produced a proof that the transitive closure is not definable in a language with aggregates [17], not assuming any unproven hypotheses from complexity theory. Since the two main distinguishing features of plain SQL are grouping and aggregation, we defined our theoretical reconstruction of SQL as the nested relational algebra [4] augmented with rational arithmetic and a general summation operator. This language can model the `GROUPBY` construct of SQL and can define familiar aggregate functions such as `TOTAL`, `AVG`, `STDEV`, etc.

The proof of [17] established the folk result above. However, it was far from ideal. It relied on proving a complicated normal form for queries that can only be achieved on a very special class of inputs. From that normal form, we derived results about the behavior of plain SQL on these inputs. That turned out to be enough to confirm the main conjecture. The proof of the normal form result relied on rewrite systems for nested relational languages developed earlier [22, 18, 19]. In particular, it made the proof very "syntactic." A change in syntax would require a new proof, although it is intuitively clear that the choice of a syntax for the language should be irrelevant.

Another problem with the proof of [17] is that, instead of establishing a *general principle* that implies expressiveness bounds, it only implied the desired result for a small number of queries. There was an attempt in [17] to find such a general principle. We introduced the notion of the *bounded degree property*, or BDP. Loosely speaking, a query has the BDP if its outputs are "simple" as long as their inputs are. We showed that (nested) relational algebra queries have the BDP. We also showed that for most recursive queries it is very easy to show how the BDP is violated, thus giving expressiveness bounds. We conjectured that the plain SQL has the BDP, but we did not prove it in [17].

We returned to the problem a few years later and proved, via a similar normal form argument, that plain SQL indeed has the BDP [6]. However, the normal form result is more complicated than that of [17] and the proof is also dependent on a particular syntax. In the same paper [6], we introduced a notion more general than the BDP. We defined *local* queries as those whose result on a given tuple can be computed by looking at a neighborhood of this tuple of a predetermined size. This notion is inspired by the classical locality theorem for first-order logic proved by Gaifman [11]. We showed in [6] that locality implies the BDP. However, continuing the pattern of setting our goals too high, we failed to prove locality of plain SQL queries, although we succeeded in proving the BDP for plain SQL queries.

The main problem in proving those results was the lack of techniques and results in finite-model theory for proving "local properties," with the exception of Gaifman's theorem and a result by Fagin, Stockmeyer and Vardi [10], that only applied to first-order logic. This changed when Nurmonen [21] showed that an analog of the result of [10] holds for first-order logic with counting quantifiers, $\mathcal{FO} + COUNT$ (as defined in [3, 8, 15]). Using Nurmonen's result, the first author proved that $\mathcal{FO} + COUNT$ is local [16] and has the BDP. As an application of these results, it was shown that a large class of queries defined in a sublanguage of plain SQL is local and has the BDP. This sublanguage was obtained by restricting the rational arithmetic of plain SQL to arithmetic of natural numbers: for example, aggregates `TOTAL` and `COUNT` were definable, but aggregates `AVG`, `STDEV` and the likes were not.

The technique of [16] was the following: it was shown that for each query Q from a given class, another query Q' can be found such that it shares most nice properties with Q (e.g., locality and the BDP) and can be expressed in $\mathcal{FO}+COUNT$. This suffices to conclude that many queries, such as the transitive closure, are not expressible.

This technique eliminates the complicated syntactic argument entirely. The differences in syntax do affect the encoding, but it is really the semantics of queries that makes the encoding possible.

In this paper, we show that the idea behind the proof in [16] can be extended to capture a much larger class of queries with aggregation. That is, we allow rational arithmetic and products over columns. Consequently, aggregates such as `AVG`, `STDEV` and many others are definable. This does complicate the proof quite a bit, but it is still much more intuitive than the syntactic one, because the overall structure of the proof remains quite straightforward, and all tedious details requiring a lot of work happen in the process of the encoding of queries in first-order logic with counting.

3 Defining the language

The goal of this section is to define a theoretical language that has the power of a relational language extended with aggregates. Following our previous approaches

$$\overline{+,*,-,\div,\mathit{exp},\mathit{root}:\mathbb{Q}\times\mathbb{Q}\to\mathbb{Q}} \qquad \overline{K0,K1:t\to\mathbb{Q}}$$

$$\overline{=:t\times t\to\{\mathbb{Q}\}} \qquad \overline{<:\mathbb{Q}\times\mathbb{Q}\to\{\mathbb{Q}\}}$$

$$\overline{\mathit{id}:t\to t} \qquad \frac{f:u\to t \quad g:s\to u}{f\circ g:s\to t}$$

$$\frac{f_i:t\to t_i, i=1,\ldots,n}{(f_1,\ldots,f_n):t\to t_1\times\ldots\times t_n} \qquad \frac{i\le n}{\pi_{i,n}:t_1\times\ldots\times t_n\to t_i}$$

$$\overline{K\{\}:t\to\{s\}} \qquad \overline{\mathit{empty}:\{t\}\to\{\mathbb{Q}\}} \qquad \overline{\eta:t\to\{t\}}$$

$$\overline{\cup:\{t\}\times\{t\}\to\{t\}} \qquad \frac{f:s\to\{t\}}{\mathit{ext}[f]:\{s\}\to\{t\}}$$

$$\frac{i\le n}{\rho_{i,n}:t_1\times\ldots\times\{t_i\}\times\ldots\times t_n\to\{t_1\times\ldots\times t_n\}}$$

$$\frac{f:s\to\mathbb{Q}}{\sum[f]:\{s\}\to\mathbb{Q}} \qquad \frac{f:s\to\mathbb{Q}}{\prod[f]:\{s\}\to\mathbb{Q}}$$

Fig. 1. Expressions of Aggr

to dealing with aggregation, we define this language to be an extension of *nested* relational algebra with arithmetic operators. Nesting accounts for grouping, as in `GROUPBY`, and arithmetic gives us the computing power for aggregates themselves. The difference between this paper and previous ones is that the arithmetic is *a lot* richer!

We define the language below. Assume the existence of two base types: Type $\mathbb{Q}$ of rational numbers, and an unspecified base type $\mathbf{b}$ whose domain is a countably infinite set D. Types of the language are given by the grammar

$$t \quad ::= \quad \mathbf{b} \mid \mathbb{Q} \mid t\times\ldots\times t \mid \{t\}$$

The semantics of type $t_1\times\ldots\times t_n$ are n-tuples such that the ith component is of type t_i. Objects of type $\{t\}$ are finite sets of objects of type t. Expressions of Aggr are defined in Figure 1.

The semantics follows that of [4, 12, 17]. We use $+$, $-$, $*$, $\div$ to denote the standard operations on rational numbers. $K0$ and $K1$ return 0 and 1 respectively.

$=$ is the equality test, where true is represented by $\{0\}$ and false by $\{\}$. With the same representation of true and false, $<$ defines the usual order on rational numbers. We use $exp(x, y)$ for x^y which is only defined if y is a natural number; and $root(x, y)$ for $\sqrt[x]{y}$ which again is undefined if x is not a natural number. These may seem a bit strange, but it does not hurt to *add* primitives if we want to prove *in*expressibility results.

The semantics for identity id, composition $\circ$, tupling, and projections π_i is standard. The result of $K\{\}$ is always the empty set. *empty* tests if a set is empty. η forms singleton sets. $\cup$ is set union. ρ_i is the "pair-with" operation. Given a function $f : s \to \{t\}$ and a set X of type $\{s\}$, $ext[f](X)$ evaluates to $\bigcup_{x \in X} f(x)$.

For the summation and product operators and $f : s \to \mathbb{Q}$, $\sum[f](X)$ is $\sum_{x \in X} f(x)$ and $\prod[f](X)$ is $\prod_{x \in X} f(x)$. For example, $\sum[K1]$ is the cardinality function, and $\prod[K1 + K1](X)$ returns $2^{card(X)}$. (Strictly speaking, $K1 + K1$ should be written as $+(K1, K1)$ but we shall often simplify notation when it does not lead to confusion.)

It is known that without the type of natural numbers, this language is equivalent to the standard nested relational algebra [4]. Furthermore, when input and output are usual flat relations (sets of atomic tuples), it expresses precisely the first-order queries. Summation, product, and arithmetic give it the power of aggregate functions. For example, the aggregate `TOTAL` is given by $\sum[id]$ and `AVG` is given by $\sum[id] \div \sum[K1]$.

Abbreviate $\mathbf{b} \times \ldots \times \mathbf{b}$, m times, as $\mathbf{b}^m$. A standard relational database is represented as an object of type $\{\mathbf{b}^{n_1}\} \times \ldots \times \{\mathbf{b}^{n_k}\}$. In other words, a relational database that consists of k relations, the ith one having arity n_i, is represented as an object of the above type. Types of this form are called *relational*. A query in AGGR is *relational* if both its input and output types are.

For example, a query that takes a graph whose nodes are in D and returns another graph is of type $\{\mathbf{b} \times \mathbf{b}\} \to \{\mathbf{b} \times \mathbf{b}\}$; that is, it is a relational query. If the transitive closure were definable in AGGR, it would have the type above. Thus, we concentrate on expressiveness of relational queries in AGGR. Note that for a relational query, types of its intermediate results need not be relational.

4 Local queries and the main theorem

Structures and neighborhoods A relational signature σ is a set of relation symbols $\{R_1, ..., R_l\}$, with an associated arity function. In what follows, $p_i (> 0)$ denotes the arity of R_i. We write σ_n for σ extended with n new constant symbols.

A σ-structure is $\mathcal{A} = \langle A, \overline{R}_1, \ldots, \overline{R}_l \rangle$, where A is a finite set, and $\overline{R}_i \subseteq A^{p_i}$ interprets R_i. The class of finite σ-structures is denoted by STRUCT[σ]. We adopt the convention that the carrier of a structure $\mathcal{A}$ is always denoted by A.

Given a structure $\mathcal{A}$, its *Gaifman graph* [7, 11, 10] $\mathcal{G}(\mathcal{A})$ is defined as $\langle A, E \rangle$ where (a, b) is in E iff there is a tuple $\vec{t} \in \overline{R}_i$ for some i such that both a and b are in $\vec{t}$. The distance $d(a, b)$ is defined as the length of the shortest path from a to b in $\mathcal{G}(\mathcal{A})$; we assume $d(a,a)=0$. Given $a \in A$, its *r-sphere* $S_r^{\mathcal{A}}(a)$ is $\{b \in A \mid d(a,b) \leq r\}$. For a tuple $\vec{t}$, define $S_r^{\mathcal{A}}(\vec{t})$ as $\bigcup_{a \in \vec{t}} S_r^{\mathcal{A}}(a)$.

Given a tuple $\vec{t} = (t_1, \ldots, t_n)$, its *$r$-neighborhood* $N_r^{\mathcal{A}}(\vec{t})$ is defined as a σ_n structure

$$\langle S_r^{\mathcal{A}}(\vec{t}), \overline{R}_1 \cap S_r^{\mathcal{A}}(\vec{t})^{p_1}, \ldots, \overline{R}_k \cap S_r^{\mathcal{A}}(\vec{t})^{p_k}, t_1, \ldots, t_n\rangle$$

That is, the carrier of $N_r^{\mathcal{A}}(\vec{t})$ is $S_r^{\mathcal{A}}(\vec{t})$, the interpretation of the σ-relations is obtained by restricting them from $\mathcal{A}$ to the carrier, and the n extra constants are the elements of $\vec{t}$. If the structure $\mathcal{A}$ is understood, we shall write $S_r(\vec{t})$ and $N_r(\vec{t})$.

Given a structure $\mathcal{A}$ and two m-ary vectors $\vec{a}$ and $\vec{b}$ of elements of A, we write $\vec{a} \approx_r^{\mathcal{A}} \vec{b}$ if $N_r^{\mathcal{A}}(\vec{a})$ and $N_r^{\mathcal{A}}(\vec{b})$ are isomorphic. That is, $\vec{a}$ and $\vec{b}$ are indistinguishable in $\mathcal{A}$ if we can only "see" up to radius r.

Local queries Assume that we have a formula in some logic, which comes with the associated notion of $\models$ between structures and formulae. Following [6, 16], we say that a formula $\psi(x_1, \ldots, x_m)$, in the logical language whose symbols are in σ, is *local* if there exists $r > 0$ such that, for every $\mathcal{A} \in \text{STRUCT}[\sigma]$ and for every two m-ary vectors $\vec{a}$, $\vec{b}$ of elements of A, $\vec{a} \approx_r^{\mathcal{A}} \vec{b}$ implies $\mathcal{A} \models \psi(\vec{a})$ iff $\mathcal{A} \models \psi(\vec{b})$. The minimum r for which this is true is called the *locality rank* of ψ.

It can be readily verified that transitive closure and deterministic transitive closure are *not* local [6, 16]. There are bounds on the expressive power of local queries that can be easily verified [6, 16]. Thus, it is rather simple to check if a query is local or not. It is particularly easy to verify that locality fails for most familiar recursive queries.

As noted above, we can represent a σ-structure as an object of type $\{\mathbf{b}^{p_1}\} \times \cdots \times \{\mathbf{b}^{p_l}\}$, where σ has l relations of arities p_1, ..., p_l. We denote this type by $\sigma_{\mathbf{b}}$.

We assume without loss of generality that the output of a relational query is one set of m-tuples. Then such a query is a mapping from σ-structures over D into finite subsets of D^m. It can be easily seen that for any such query Q definable in Aggr, an element $d \in D$ occurs in a tuple in $Q(\mathcal{A})$ for some structure $\mathcal{A}$ with carrier A only if $d \in A$. Thus, we define $\psi_Q(x_1, \ldots, x_m)$ by letting

$$\mathcal{A} \models \psi_Q(\vec{a}) \;\text{ iff }\; \vec{a} \in Q(\mathcal{A})$$

Then $Q(\mathcal{A}) = \{\vec{a} \in A^m \mid \mathcal{A} \models \psi_Q(\vec{a})\}$.

We say that Q is *local* if so is the associated formula ψ_Q. Our main result is:

Theorem 1. *Every relational query in* Aggr *is local.*

Since the transitive closure query (or deterministic transitive closure) is not local, we obtain:

Corollary 2. *Transitive closure is not expressible in* Aggr. $\square$

This was proved before for a language weaker than AGGR [17, 6]. In fact, the language of [17, 6] is AGGR without the product operator and with less arithmetic.

References [17, 6, 16] also discuss a closely related property, called the bounded degree property, or BDP. When specialized to graphs, it says that for any query q from graphs to graphs, there exists a function $f_q : \mathbb{N} \to \mathbb{N}$ such that, whenever all degrees of nodes in a graph G do not exceed k, the number of distinct in- and out-degrees in $q(G)$ does not exceed $f_q(k)$. This property is particularly easy to use to obtain expressiveness bounds, see [17, 6, 16]. According to [6], locality implies the BDP. Hence,

Corollary 3. *Every relational query in* AGGR *has the bounded degree property.* □

5 Flattening the language

Proving inexpressibility results for a language with nesting is hard because nesting essentially corresponds to second-order constructs. Fortunately, we can find a *flat* language $\text{AGGR}_{\text{flat}}$, that does not use nested sets, such that every relational query definable in AGGR is also definable in $\text{AGGR}_{\text{flat}}$. Furthermore, $\text{AGGR}_{\text{flat}}$ uses natural numbers instead of rationals, which makes it easier to encode its queries in an extension of first-order logic with counting.

The expressions of $\text{AGGR}_{\text{flat}}$ are given in Figure 2. There are a number of differences between AGGR and $\text{AGGR}_{\text{flat}}$. First, $\text{AGGR}_{\text{flat}}$'s types are $\mathbf{b}$, $\mathbb{N}$, record types of the form $s_1 \times \ldots \times s_k$ where each s_i is either $\mathbf{b}$ or $\mathbb{N}$, and set types $\{t\}$ where t is a record type. That is, no nested sets are allowed.

In the expressions in Figure 2, s, t, and t_i's range over record types, and S and T range over both record and set types. The operator *cartrpod* is the usual cartesian product of sets; it is definable in AGGR using *ext* and ρ. Given a function $f : S \times s \to \{t\}$ and a pair (X, Y), where X is of type S and Y is a set of type $\{s\}$, $ext_2[f](X, Y)$ evaluates to $\bigcup_{y \in Y} f(X, y)$. $ext_2[f](X, Y)$ in $\text{AGGR}_{\text{flat}}$ can be implemented in AGGR as $ext[f](cartprod(\eta(X), Y))$, which involves a nested set. The extra parameter of $ext_2[f]$ allows us to avoid the construction of a nested set. Note that we do not need to introduce the similar $\sum_2[f]$ or $\prod_2[f]$, because $\sum_2[f] = \sum[\pi_2] \circ ext_2[\eta \circ (\pi_2, f)]$ and similarly for $\prod_2[f]$. On the natural numbers, $root(n, m)$ evaluates to k if $k^n = m$; otherwise *root* evaluates to *zero*. $repr(n, m)$ gives the canonical representation of the rational number $\frac{n}{m}$; that is, $repr(n, m) = (n', m')$ iff $\frac{n}{m} = \frac{n'}{m'}$ and n', m' have no common divisors. This function is undefined if $m = 0$, and is identity if $n = 0$.

We now have:

Proposition 4. *Every relational query definable in* AGGR *is also definable in* $\text{AGGR}_{\text{flat}}$.

Proof sketch. We first define a language $\text{AGGR}^{\mathbb{N}}$ as $\text{AGGR}_{\text{flat}}$ without restriction to flat types (that is, all the operations are the same, and the type

$$\frac{}{+,*,\dot{-}, exp, root : \mathbb{N}\times\mathbb{N}\to\mathbb{N}} \quad \frac{}{K0, K1 : T \to \mathbb{N}} \quad \frac{}{repr : \mathbb{N}\times\mathbb{N}\to\mathbb{N}\times\mathbb{N}}$$

$$\frac{}{=_{\mathbf{b}}: \mathbf{b}\times\mathbf{b}\to\{\mathbb{N}\}} \quad \frac{}{=_{\mathbb{N}}: \mathbb{N}\times\mathbb{N}\to\{\mathbb{N}\}} \quad \frac{}{empty : \{t\}\to\{\mathbb{N}\}}$$

$$\frac{}{id : T \to T} \quad \frac{f : u \to t \quad g : s \to u}{f \circ g : s \to t}$$

$$\frac{f_i : t \to t_i, i = 1,\ldots,n}{(f_1,\ldots,f_n) : t \to t_1 \times \ldots \times t_n} \quad \frac{i \le n}{\pi_{i,n} : t_1 \times \ldots \times t_n \to t_i}$$

$$\frac{}{K\{\} : T \to \{s\}} \quad \frac{}{\eta : t \to \{t\}} \quad \frac{}{\cup : \{t\}\times\{t\}\to\{t\}} \quad \frac{f : S \times s \to \{t\}}{ext_2[f] : S \times \{s\} \to \{t\}}$$

$$\frac{}{cartprod : \{t_1\}\times\ldots\times\{t_n\}\to\{t_1\times\ldots\times t_n\}}$$

$$\frac{f : s \to \mathbb{N}}{\sum[f] : \{s\}\to\mathbb{N}} \quad \frac{f : s \to \mathbb{N}}{\prod[f] : \{s\}\to\mathbb{N}}$$

Fig. 2. Expressions of $\text{AGGR}_{\mathsf{flat}}$

system is $t ::= \mathbf{b} \mid \mathbb{N} \mid t \times \ldots \times t \mid \{t\}$. We show that every relational AGGR-query is definable in $\text{AGGR}^{\mathbb{N}}$. For that, we model every rational number r by a triple (s, n, m) of natural numbers such that $|r| = \frac{n}{m}$, $s = 0$ if $r < 0$ and $s = 1$ if $r \geq 0$, and n, m have no common divisors. Then it is easy to see that all rational arithmetic can be simulated with natural arithmetic, since we have *repr* in the language. Note that we need $\prod$ over natural numbers in order to simulate both $\sum$ and $\prod$ over the rationals. It further follows that $\text{AGGR}^{\mathbb{N}}$ has the conservative extension property (cf. [22, 18, 19]). Since every relational query has flat input and output, it can be expressed in the flat fragment of $\text{AGGR}^{\mathbb{N}}$, which is precisely $\text{AGGR}_{\mathsf{flat}}$. □

6 Proof sketch of the main theorem

In view of Proposition 4, we now have to show

Proposition 5. *Every relational query in* $\text{AGGR}_{\mathsf{flat}}$ *is local.*

We start by defining $\mathcal{FO} + COUNT$, the first-order logic with counting of [8]. The logic has two sorts: The domain for the first sort is D, and the domain for the second is $\mathbb{N}$. Over the first sort, we have the usual first-order logic. The following are available for the second sort: Constants 1 and max, where the meaning of max is the size of the finite model; the usual ordering $<$, and the BIT predicate. Second-sort quantifiers $\forall i$ and $\exists i$ are bounded, meaning for all (exists) i between 1 and max. The counting quantifier $\exists ix.\varphi(x)$ means that φ has at least i satisfiers, and again $1 \leq i \leq \max$. This binds x but not i; for example,

$$\exists i.\mathrm{BIT}(1, i) \wedge [\exists ix.\varphi(x) \wedge (\forall j \exists jx.\varphi(x) \rightarrow j \leq i)]$$

tests if the number of satisfiers of φ is nonzero and even. We use $\exists! ix.\varphi(x)$ for $\exists ix.\varphi(x) \wedge (\forall j \exists jx.\varphi(x) \rightarrow j \leq i)$; that is, that there exists exactly i satisfiers.

Now suppose we have a relational type $\sigma_{\mathbf{b}}$, corresponding to some relational signature σ, and a relational query of type $\sigma_{\mathbf{b}} \rightarrow \{\mathbf{b}^m\}$. Next, consider the type

$$(\sigma, S)_{\mathbf{b}} = \sigma_{\mathbf{b}} \times \{\mathbf{b} \times \mathbf{b}\}$$

Its objects are finite structure of the signature $\sigma \cup \{S\}$, where S is a binary relational symbol not in σ. If $\mathcal{A}$ is a σ-structure and S is a binary relation, we denote the corresponding $\sigma \cup \{S\}$ structure by $\mathcal{A}_S$. We use S for both relational symbol and its interpretation. Such structures are represented as AGGR-objects of type $(\sigma, S)_{\mathbf{b}}$.

In what follows, we use the convention that C stands for the set of elements of S, that is, the union of the two projections of S. Given a function $g : \mathbb{N} \rightarrow \mathbb{N}$, we define the following condition $(\star_g[\mathcal{A}_S])$ on a structure $\mathcal{A}_S$:

$$(\star_g[\mathcal{A}_S]) \qquad (A \cap C = \emptyset) \ \wedge \ (S \text{ is a linear order}) \ \wedge \ (card(C) > g(card(A)))$$

We also define a new query Q_g of type $(\sigma, S)_{\mathbf{b}} \rightarrow \{\mathbf{b}^m\}$ as:

$$Q_g(\mathcal{A}_S) \quad = \quad \begin{cases} Q(\mathcal{A}) & \text{if } (\star_g[\mathcal{A}_S]) \text{ holds} \\ \emptyset & \text{otherwise} \end{cases}$$

We start with three propositions.

Proposition 6. *All formulae in $\mathcal{FO}$ + COUNT with no free variable of the second sort are local.* □

Proposition 7. *Let Q be any relational query in* $\text{AGGR}_{\text{flat}}$. *Then for every g, it is the case that Q is local iff Q_g is local.* □

Proposition 8. *Let Q be any relational query in* $\text{AGGR}_{\text{flat}}$. *Then there is a function g such that ψ_{Q_g} is definable in $\mathcal{FO}$ + COUNT.* □

Now the main theorem can be obtained as follows. We consider a relational query Q of $\text{AGGR}_{\text{flat}}$ and use Proposition 8 to find g such that ψ_{Q_g} is definable in $\mathcal{FO}$ + *COUNT*. By Proposition 6, ψ_{Q_g} is local; hence Q_g is local. From Proposition 7 we conclude that Q is local as desired.

To complete the argument, we need to furnish proofs of the three propositions above. The proof of Proposition 6 can be found in [16]. The proof of Proposition 8 is in the appendix. The proof of Proposition 7 is as follows:

Let Q_g be local. Let r be its locality rank. Let the input structure to Q be $\mathcal{A}$. Let $\vec{a} \approx_r^{\mathcal{A}} \vec{b}$, where $\vec{a}$ and $\vec{b}$ are m-vectors of elements of A. Let $n = card(A)$ and let C be a subset of D such that $C \cap A = \emptyset$ and $card(C) > g(n)$. Let S be an arbitrary linear ordering on C. We define $\mathcal{A}_S$ as $\mathcal{A}$ extended with the binary relation S. Since $S_d^{\mathcal{A}_S}(\vec{a})$ does not contain any element of C, and neither does $S_d^{\mathcal{A}_S}(\vec{b})$ for any d, we obtain $\vec{a} \approx_r^{\mathcal{A}_S} \vec{b}$. Thus by the locality of Q_g, $\vec{a} \in Q_g(\mathcal{A}_S)$ iff $\vec{b} \in Q_g(\mathcal{A}_S)$. Since all the conditions in $(\star_g[\mathcal{A}_S])$ hold, we have $Q_g(\mathcal{A}_S) = Q(\mathcal{A})$. Hence, $\vec{a} \in Q(\mathcal{A})$ iff $\vec{b} \in Q(\mathcal{A})$, which proves that Q is local, and its locality rank is at most r. □

7 Conclusion

We have proved the most powerful result so far that gives us expressiveness bounds for relational queries with aggregation. In particular, recursive queries such as transitive closure are not definable with the help of grouping, summation and product over columns, and standard rational arithmetic.

After our PODS'94 paper in which inexpressibility of transitive closure in a weaker language was proved, there was a renewed activity in the area that resulted in 3 papers, improving both results and techniques: [6], presented at ICDT'97, [16], presented at LICS'97 paper, and this paper. So one may ask if this is the end of the story.

We believe that, to the contrary, this work is very far from being completed. Until very recently, it was widely believed that counting formalisms developed in finite-model theory are a *wrong* way to approach the problem aggregation. We hope to have convinced the reader that this is not necessarily the case, and logics with counting *are* useful. The connection though is not lying on the very surface, as the encoding is not completely trivial. A possible reason why it took a number of years to apply logics with counting to the study of query languages is that until recently, very few tools for $\mathcal{FO} + COUNT$ were available. The games of Immerman and Lander [15] were essentially the only such tool, and these are *not* convenient to use. The best known separation result proved via games was the one from the conference version of Etessami's paper [8]. It required a complicated combinatorial argument, even though the structures are extremely simple. Tools like those suggested in [21, 16] simplify the proofs considerably, as was demonstrated in [8, 16]. This makes first-order logic with counting much more attractive as a tool for studying aggregation.

But there is still a long way to go. For example, it seems likely that the class of allowed arithmetic functions and predicates should not affect the expressibility of, say, transitive closure. However, first-order logic with counting, which we used for encoding, limits the arithmetic operations. We finish the paper with two main *challenges* that we believe must be addressed to develop a good finite-model theory counterpart for languages with aggregation.

CHALLENGE 1 Find an extension of first-order logic with a counting mechanism that is a *natural* analog of relational languages with aggregation. Furthermore, such an extension must possess nice model-theoretic properties as to be applicable to the study of expressiveness of languages with aggregation. Note that $\mathcal{FO} + COUNT$ is not a good candidate. We have seen that the encoding in $\mathcal{FO} + COUNT$ is quite an unpleasant one, but we had to use $\mathcal{FO} + COUNT$ because of its nice known properties.

CHALLENGE 2 Find techniques that extend the results to *ordered* databases. By this we mean having an order relation on the elements of the base type, not only on rational (or natural) numbers.

The results on expressive power of relational calculus extend to the ordered setting. To be able to state results about *real* languages with aggregates, we must deal with the ordered case. However, none of the tools developed for logics such as $\mathcal{FO} + COUNT$ gives us any hints as to how to approach the ordered case.

Acknowledgement We thank anonymous referees and Rick Hull for their comments.

References

1. S. Abiteboul, R. Hull, V. Vianu, *Foundations of Databases*, Addison Wesley, 1995.
2. A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Proceedings of 6th Symposium on Principles of Programming Languages, Texas*, pages 110–120, January 1979.
3. D.A. Barrington, N. Immerman, H. Straubing. On uniformity within NC^1. *JCSS*, 41:274–306,1990.
4. P. Buneman, S. Naqvi, V. Tannen, L. Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, September 1995.
5. M. Consens and A. Mendelzon. Low complexity aggregation in GraphLog and Datalog, *Theoretical Computer Science* 116 (1993), 95–116. Extended abstract in *ICDT'90*.
6. G. Dong, L. Libkin, L. Wong. Local properties of query languages. *Proc. Int. Conf. on Database Theory*, Springer LNCS 1186, 1997, pages 140–154.
7. H.-D. Ebbinghaus and J. Flum. *Finite Model Theory.* Springer Verlag, 1995.
8. K. Etessami. Counting quantifiers, successor relations, and logarithmic space, *Journal of Computer and System Sciences* 54 (1996), 400–411. Extended abstract in *Structure in Complexity'95*.
9. R. Fagin. Easier ways to win logical games. In *Proc. DIMACS Workshop on Finite Model Theory and Descriptive Complexity*, 1996.
10. R. Fagin, L. Stockmeyer, M. Vardi, On monadic NP vs monadic co-NP, *Information and Computation*, 120 (1994), 78–92.
11. H. Gaifman, On local and non-local properties, *in* "Proceedings of the Herbrand Symposium, Logic Colloquium '81," North Holland, 1982.
12. S. Grumbach, L. Libkin, T. Milo and L. Wong. Query languages for bags: expressive power and complexity. *SIGACT News*, 27 (1996), 30–37.
13. Y. Gurevich. Toward logic tailored for computational complexity. In *Proceedings of Computation and Proof Theory*, Springer Lecture Notes in Mathematics, vol. 1104, 1984, pages 175–216.

14. N. Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16:760–778, 1987.
15. N. Immerman and E. Lander. Describing graphs: A first order approach to graph canonization. In *"Complexity Theory Retrospective"*, Springer Verlag, Berlin, 1990.
16. L. Libkin. On the forms of locality over finite models. In *LICS'97*, pages 204–215.
17. L. Libkin, L. Wong. New techniques for studying set languages, bag languages, and aggregate functions. In *PODS'94*, pages 155–166. Full version to appear in *JCSS*.
18. L. Libkin and L. Wong. Aggregate functions, conservative extension, and linear orders, *in* "Proceedings of 4th International Workshop on Database Programming Languages," Manhattan, New York, August 1993.
19. L. Libkin and L. Wong. Conservativity of nested relational calculi with internal generic functions. *Information Processing Letters*, 49 (1994), 273–280.
20. I. Niven and H.S. Zuckerman. *Introduction to the Theory of Numbers*. Wiley, 1980.
21. J. Nurmonen. On winning strategies with unary quantifiers. *J. Logic and Computation*, 6 (1996), 779–798.
22. L. Wong. Normal forms and conservative properties for query languages over collection types. *JCSS* 52(1):495–505, June 1996.

Appendix: Proof of Proposition 8

Preliminaries

We start with a few definitions. Given an object x, $\mathsf{adom}_{\mathbf{b}}(x)$ and $\mathsf{adom}_{\mathbb{N}}(x)$ stand for the active domains of type $\mathbf{b}$ and $\mathbb{N}$ of x respectively. That is, $\mathsf{adom}_{\mathbf{b}}(x)$ is the set of all elements of D (the domain of type $\mathbf{b}$) that occur in x; and $\mathsf{adom}_{\mathbb{N}}(x)$ is the set of all natural numbers that occur in x. We use $\mathsf{adom}(x)$ for $\mathsf{adom}_{\mathbf{b}}(x) \cup \mathsf{adom}_{\mathbb{N}}(x)$. We also assume that D and $\mathbb{N}$ are disjoint. The cardinalities of $\mathsf{adom}_{\mathbf{b}}(x)$, $\mathsf{adom}_{\mathbb{N}}(x)$, and $\mathsf{adom}(x)$ are denoted by $\mathsf{size}_{\mathbf{b}}(x)$, $\mathsf{size}_{\mathbb{N}}(x)$, and $\mathsf{size}(x)$ respectively.

Given a $\text{AGGR}_{\text{flat}}$ function f and an object x, we define the set $\mathsf{Int\_res}(f, x)$ of intermediate results of evaluation of f on x as

$$\begin{array}{rl}
\mathsf{Int_res}(g, h(x)) \cup \mathsf{Int_res}(h, x) & \text{if } f = g \circ h \\
\mathsf{Int_res}(f_1, x) \cup \ldots \cup \mathsf{Int_res}(f_n, x) & \text{if } f = (f_1, \ldots, f_n) \\
\{x\} \cup \{f(x)\} \cup \bigcup_{y \in x} \mathsf{Int_res}(g, y) & \text{if } f = \sum[g] \text{ or } f = \prod[g] \\
\{(X, Y)\} \cup \{f(X, Y)\} \cup \bigcup_{y \in Y} \mathsf{Int_res}(g, (X, y)) & \text{if } f = ext_2[g] \text{ and } x = (X, Y) \\
\{x, f(x)\} & \text{otherwise}
\end{array}$$

Intuitively, $\mathsf{Int\_res}(f, x)$ contains all intermediate results obtained in the process of evaluating f on x.

We now define

$$\mathsf{adom}_{\mathbf{b}}(f, x) = \bigcup_{y \in \mathsf{Int\_res}(f, x)} \mathsf{adom}_{\mathbf{b}}(y)$$

$$\mathsf{adom}_{\mathbb{N}}(f, x) = \bigcup_{y \in \mathsf{Int\_res}(f, x)} \mathsf{adom}_{\mathbb{N}}(y)$$

$$\mathsf{adom}(f, x) = \mathsf{adom}_{\mathbf{b}}(f, x) \cup \mathsf{adom}_{\mathbb{N}}(f, x)$$

and $\mathsf{size}_{\mathbf{b}}(f, x)$, $\mathsf{size}_{\mathbb{N}}(f, x)$ and $\mathsf{size}(f, x)$ as their cardinalities. That is, $\mathsf{adom}_{\mathbf{b}}(f, x)$ is the set of all elements of D that occur in the process of evaluating of f on x, and $\mathsf{adom}_{\mathbb{N}}(f, x)$ is the set of all natural numbers that occur in this process.

Since all operations in $\mathrm{AGGR}_{\mathrm{flat}}$ except *exp*, *root*, and $\prod[f]$ can be evaluated in polynomial time (cf. [4, 12, 17]), and those that are not ptime produce a single number, we obtain:

Lemma 9. *For any* $\mathrm{AGGR}_{\mathrm{flat}}$ *expression* f*, there exists a constant* k_f *such that for any object* x *with* $\mathsf{size}(x) = n > 1$*, on which* f *is defined,*

$$\mathsf{size}(f, x) \; < \; n^{k_f}$$

From this lemma, by a simple structural induction on $\mathrm{AGGR}_{\mathrm{flat}}$ expressions, we prove:

Lemma 10. *For any* $\mathrm{AGGR}_{\mathrm{flat}}$ *expression* f*, there exists a constant* C_f *such that for any object* x *with* $\mathsf{size}(x) = n > 1$*, on which* f *is defined, and for every* $m \in \mathsf{adom}_{\mathbb{N}}(f, x)$*, it is the case that*

$$m \; < \; n^{n^{C_f}}$$

In particular, if f is a relational query, we obtain $m < n^{n^{C_f}}$ for any $m \in \mathsf{adom}_{\mathbb{N}}(f, x)$, where $n = \mathsf{size}_{\mathbf{b}}(x)$. This gives us an upper bound on any natural number that can be encountered in the process of evaluating f on x.

For the rest of the proof we assume that any input to a relational query has size at least 2. At the end of the proof we'll explain how to deal with empty and one-element active domains.

Given a number $N > 1$, we call a function $enc_m(a_1, \ldots, a_m)$ an *encoding relative to* N if it uniquely encodes m-tuples of natural numbers less than N; that is, $\vec{a} \neq \vec{b}$ implies $enc_m(\vec{a}) \neq enc_m(\vec{b})$, whenever all components of $\vec{a}$ and $\vec{b}$ are below N. Such a function can be chosen so that it is a polynomial in $a_1, \ldots, a_m, N$ and its values are less than N^l for some l. For example, enc_2 can be defined as $enc_2(a, b) = aN + b$; thus, its values do not exceed $N^2 + N$ and are thus less than N^3. To encode m-tuples, we just apply enc_2 to the first component and an encoding of the remaining $m-1$-tuple.

According to [8], the predicates $+(i, j, k)$ and $*(i, j, k)$ meaning $i + j = k$ and $i * j = k$ are definable in $\mathcal{FO} + COUNT$, as long as i, j, k are elements of the second sort under max. Thus, we shall use polynomial (in)equalities in $\mathcal{FO} + COUNT$ formulae. For example, the parity test can be rewritten as

$$\exists k \exists i. k + k = i \wedge \exists ! i x. \varphi(x)$$

The encoding

Let Q be a relational query in $\text{AGGR}_{\text{flat}}$. We define g as

$$g(n) \quad = \quad n^{n^{n^c}}$$

where c is a constant to be determined later. We claim that there exists a constant c such that Q_g is definable in $\mathcal{FO} + COUNT$.

The idea of the encoding is that a number n is represented by the element $c_n \in C$ such that the cardinality of $\{x \mid S(x, c_n)\}$ is n. Then the counting power of $\mathcal{FO} + COUNT$ is applied on the relation S. The size of C, given by g, turns out to be enough to model all arithmetic that is needed in order to evaluate Q.

Given a $\sigma \cup \{S\}$-structure $\mathcal{A}_S$ (where S is the extra binary relation) and a number k, it is possible to write a $\mathcal{FO} + COUNT$ formula that checks if $(*_g[\mathcal{A}_S])$ holds where g is of the form above. Indeed, we first notice that there are first-order formulae $\mathsf{adom}_{\mathcal{A}}(x)$ and $\mathsf{adom}_S(x)$ that test if x is in the carrier of $\mathcal{A}$ (that is, $x \in A$), or x is a node in the binary relation S (that is, $x \in C$). For example, $\mathsf{adom}_S(x) = \exists y. S(y, x) \vee S(x, y)$. Also, there exists a first-order formula $LIN(S)$ stating that S is a linear order.

We next claim that there is a $\mathcal{FO} + COUNT$ definable predicate $exp(x, y, z)$ that holds iff $x, y, z \in C$ and $x^y = z$; that is, exp represents the graph of exponentiation, where n is encoded by $c_n \in C$ such that the cardinality of $\{x \mid S(x, c_n)\}$ is n. We use the notation $x^y = z$, but strictly speaking we mean that for numbers i, j, k represented by x, y, z we have $i^j = k$; in what follows we shall often write arithemtic formulae on the elements of C, to keep the notation simpler. We use the shorthand $\mathsf{is}(x, i)$ for $\exists ! iy. S(y, x)$; that is, $\mathsf{is}(x, i)$ means that x represents the number i.

To show that exp is definable, first notice that there is a formula $pow(x, y)$ stating that x is a power of y, provided c_y is prime:

$$pow(x, y) = \exists i \exists j. [\mathsf{is}(x, i) \wedge \mathsf{is}(y, j) \wedge (\forall k \forall l. k * l = i \rightarrow (k = 1 \vee (\exists k'. k' * j = k)))]$$

Now $exp_{pr}(x, y, z)$ defined as:

$$exp_{pr}(x, y, z) = pow(z, y) \wedge \exists i \exists j. [\mathsf{is}(y, i) \wedge (j = i + 1) \wedge \exists ! jv. (pow(x, v) \wedge S(v, z))]$$

states that $x^y = z$ for y prime. That is, $exp_{pr}(x, y, z)$ if z is a power fo x, and the number of powers of x that do not exceed z is $y + 1$.

Now we define two new formulae:

$$div(a, b) \quad = \quad \exists i, j, k \exists u. \mathsf{is}(a, i) \wedge \mathsf{is}(b, j) \wedge \mathsf{is}(u, k) \wedge i = j * k$$

$$prime(p) = \forall u \forall i. (S(u, p) \wedge \mathsf{is}(u, i)) \rightarrow (i = 1 \vee \neg div(p, u))$$

That is, $div(a, b)$ says that a is divisible by b, and $prime(p)$ says that p is prime. Next, we define $factor(p, a, x)$ meaning that p is prime, p^a divides x, but p^{a+1} does not divide x:

$$\begin{aligned} factor(p, a, x) = {} & prime(p) \wedge \exists v. \ [S(v, x) \wedge exp_{pr}(p, a, v) \wedge div(x, v) \wedge \\ & \forall w \forall i, j, k. (\mathsf{is}(w, i) \wedge \mathsf{is}(v, j) \wedge \mathsf{is}(p, k) \wedge i = j * k \rightarrow \neg div(x, w))] \end{aligned}$$

With this, we finally define $exp(x,y,z)$ as $exp_1(x,y,z) \wedge exp_2(x,y,z)$ where:

$$exp_1(x,y,z) \;=\; \forall p \forall a.factor(p,a,x) \rightarrow$$
$$(\exists b \exists i,j,k.\mathsf{is}(b,i) \wedge \mathsf{is}(y,j) \wedge \mathsf{is}(a,k) \wedge i = j * k \wedge factor(p,b,z))$$

$$exp_2(x,y,z) \;=\; \forall p \forall a.factor(p,a,z) \rightarrow \exists b.factor(p,b,x)$$

With this formula exp, we can define the condition that $card(C) > g(card(A))$ as:

$$\chi_g = \exists x \exists y.[(\mathsf{adom}_S(x) \wedge (\exists i.\mathsf{is}(x,i) \wedge \exists! iv.\mathsf{adom}_{\mathcal{A}}(v))) \wedge (\exists v.S(y,v)) \wedge \chi'(x,y)]$$

where $\chi'(x,y)$ expresses the condition that $card(C) > g(card(A))$ as the conjunction of first-order formula stating that C has at least c elements, and for the cth element, denoted by x_c, the following holds:

$$\exists v_1 \exists v_2.(\mathsf{adom}_S(v_1) \wedge \mathsf{adom}_S(v_2)) \wedge (exp(x,x_c,v_1) \wedge exp(x,v_1,v_2) \wedge exp(x,v_2,y))$$

That is, y represents the value of g on x (which is the cardinality of A), and there is an element S-bigger than y, that ensures strict inequality.

Finally, we use

$$test_* \;=\; LIN(S) \wedge (\forall x.\mathsf{adom}_{\mathcal{A}}(x) \rightarrow \neg \mathsf{adom}_S(x)) \wedge \chi_g$$

to test for $(\star_g[\mathcal{A}_S])$.

Thus, for the rest of the proof we assume that the input structure satisfies $(\star_g[\mathcal{A}_S])$. If we produce a $\mathcal{FO} + COUNT$ formula ψ that defines Q_g on such structures, the formula that defines Q on all structures is simply $\psi \wedge test_*$.

We now explain the encoding of objects and $\text{AGGR}_{\text{flat}}$ functions. The encoding is relative to the input structure $\mathcal{A}_S$. We assume that the first sort is the carrier of the finite structure ($\mathcal{A}_S$ in our case); thus, elements of type $\mathbf{b}$ are encoded by themselves. Each element of type $\mathbb{N}$, that is, a natural number n, is encoded by $c_n \in C$ such that $card(\{x \mid S(x,c_n)\}) = n$. Note that we do not use the second sort in the encoding; natural numbers are still encoded as elements of the first sort, and the counting power of $\mathcal{FO} + COUNT$ is only used in simulation of functions.

Suppose we have a function f of type $s \rightarrow t$. Then s is a product of record types and types of the form $\{s'\}$ where s' is a record type. Without loss of generality (and keeping the notation simple) we list types not under the set brackets first; that is, s is

$$\mathbf{b}^l \times \mathbb{N}^k \times \{t_1\} \times \ldots \times \{t_m\}$$

where $t_1, \ldots, t_m$ are record types, t_i being the product of w_i base types. We also assume that t is either $\mathbf{b}$, or $\mathbb{N}$, or $\{u\}$, where u is a record type, since functions into product of types will be modeled by a tuple of formulae.

Let x be an arbitrary object. We say that x is $\mathcal{A}_S$-compatible if $\mathsf{adom}_{\mathbf{b}}(x) \subseteq A$ and $i < card(C)$ for any $i \in \mathsf{adom}_{\mathbb{N}}(x)$. If this is the case, by $x_{\mathcal{A}_S}$ we denote an object obtained from x by replacing each natural number n that occurs in x

with c_n; its type is then obtained from the type of x by replacing each $\mathbb{N}$ with $\mathbf{b}$.

With each sequence $\mathcal{T} = (\{t_1\}, \ldots, \{t_m\})$, where t_is are record types, we associate a signature $\mathcal{T}_{\text{sig}}$ that consists of m relational symbols $S^1, \ldots, S^m$, with S^i having arity w_i. (In the process of encoding, at each step we shall assume a fresh collection of relation symbols.) By $\sigma(\mathcal{T})$ we denote the disjoint union of σ, $\{S\}$, and $\mathcal{T}_{\text{sig}}$.

Now we consider two cases.

Case 1: t is $\mathbf{b}$ or $\mathbb{N}$. Then f is encoded as a formula $\psi_f(\vec{x}, \vec{y}, z)$ in the language $\sigma(\mathcal{T})$, where $\vec{x}$ has l elements and $\vec{y}$ has k elements. The condition on ψ_f is the following.

Assume that $\mathcal{B}$ is an object of type $\{t_1\} \times \ldots \times \{t_m\}$ that is $\mathcal{A}_S$-compatible. Let $\mathcal{B}'$ be the $\sigma(\mathcal{T})$ structure that consists of $\mathcal{A}_S$ and $\mathcal{B}_{\mathcal{A}_S}$ (interpreting symbols in $\mathcal{T}_{\text{sig}}$). Let $\vec{x}$ and $\vec{y}$ be $\mathcal{A}_S$-compatible. Then, for every $\mathcal{A}_S$-compatible z, it is the case that

$$z = f(\vec{x}, \vec{y}, \mathcal{B}) \qquad \text{iff} \qquad \mathcal{B}' \models \psi_f(\vec{x}, \vec{y}_{\mathcal{A}_S}, z_{\mathcal{A}_S})$$

Case 2: t is $\{u\}$ where u is a record type with arity w. Then f is encoded as a formula $\psi_f(\vec{x}, \vec{y}, \vec{z})$ in the language $\sigma(\mathcal{T})$, where $\vec{x}$ and $\vec{y}$ are as before, and $\vec{z}$ is a w-vector of variables of the first sort. The condition is that, for every $\mathcal{A}_S$-compatible $\vec{x}$, $\vec{y}$ and $\mathcal{B}$ as above, the set

$$Z \quad = \quad f(\vec{x}, \vec{y}, \mathcal{B})$$

is $\mathcal{A}_S$-compatible, and

$$\{\vec{z} \in (A \cup S)^w \mid \mathcal{B}' \models \psi_f(\vec{x}, \vec{y}_{\mathcal{A}_S}, \vec{z})\} \quad = \quad Z_{\mathcal{A}_S}.$$

If t is a product of types, we encode f as the tuple of encodings of all projections.

We now show how to encode $\text{A}\textsc{ggr}_{\text{flat}}$ expressions so that the conditions 1 and 2 above are satisfied. First note that composition is rather straightforward and essentially corresponds to substitution. Next, consider natural arithmetic.

The function $K0$ is encoded as

$$\psi_{K0}(\_, x) = \mathsf{adom}_S(x) \wedge \neg\exists y. S(y, x);$$

that is, x is the smallest element of S. Similarly,

$$\psi_{K1}(\_, x) = \mathsf{adom}_S(x) \wedge \exists! y. S(y, x)$$

The encoding of operations on $\mathbb{N}$ is straightforward:

$$\psi_+(x, y, z) = \begin{array}{c} \mathsf{adom}_S(x) \wedge \mathsf{adom}_S(y) \wedge \mathsf{adom}_S(z) \\ \wedge\ \exists i \exists j \exists k. (\mathsf{is}(x, i) \wedge \mathsf{is}(y, j) \wedge \mathsf{is}(z, k)) \wedge (i + j = k), \end{array}$$

and similarly for other $*, \dot{-}$ and *exp* (since we know how to define *exp*). For *root*, we use

$$\psi_{root}(x, y, z) = exp(z, y, x) \vee (\neg exp(z, y, x) \wedge \neg\exists v. S(v, z))$$

That is, if the root does not exists, we return 0. For *repr* we use

$$\begin{aligned}&\psi_{repr}(x,y,x',y')\\ =\ &(\exists i,i',j,j',l.\mathsf{is}(x,i)\wedge\mathsf{is}(x',i')\wedge\mathsf{is}(y,j)\wedge\mathsf{is}(y',j')\wedge l=i*j'\wedge l=j*i')\\ &\wedge\forall z.\neg(div(x',z)\leftrightarrow div(y',z))\end{aligned}$$

Note that we have to compute the product of two numbers; thus, the size of S must be at least the square of maximum possible number that can be encountered in evaluating Q. We shall see later when we determine the function g that this is the case, and thus we can use the formula above.

The order on $\mathbb{N}$ is given by

$$\psi_<(x,y,z)\ =\ \mathsf{adom}_S(x)\wedge\mathsf{adom}_S(y)\wedge S(x,y)\wedge\neg\exists v.S(v,z)$$

The equality test is similarly defined:

$$\psi_=(x,y,z)\ =\ (x=y)\wedge\mathsf{adom}_S(z)\wedge\neg\exists y.S(y,x).$$

The operations on sets are very simple: for example, union is encoded as $\psi_\cup(\vec{z})=S^1(\vec{z})\vee S^2(\vec{z})$ (recall that S^1 and S^2 are symbols in the signature $\mathcal{T}$);

$$\psi_{empty}(z)\ =\ \neg\exists\vec{x}.S^1(\vec{x})\wedge\mathsf{adom}_S(z)\wedge\neg\exists y.S(y,x).$$

For the singleton, we have $\psi_\eta(x,y)=(x=y)$. The encoding of *cartprod* depends on the arities of types involved and their number. In general, we define

$$\psi_{cartprod_n}(\vec{x})\ =\ \exists\vec{x}_1\ldots\exists\vec{x}_n.S^1(\vec{x}_1)\wedge\ldots\wedge S^n(\vec{x}_n)\wedge\chi(\vec{x}_1,\ldots,\vec{x}_n,\vec{x})$$

where $\chi(\vec{x}_1,\ldots,\vec{x}_n,\vec{x})$ is a formula in the language of equality stating that $\vec{x}$ is concatenation of $\vec{x}_1,\ldots,\vec{x}_n$.

The encoding of $K\{\}$ is simply *false*.

To encode $ext_2[f]$, we use ψ_f encoding f and obtain

$$\psi_{ext_2[f]}(\vec{x},\vec{z})\ =\ \exists\vec{y}.S^1(\vec{y})\wedge\psi_f(\vec{x},\vec{y},\vec{z})$$

in the case when the first argument of f is a record type, and

$$\psi_{ext_2[f]}(\vec{z})\ =\ \exists\vec{y}.S^1(\vec{y})\wedge\psi_f(\vec{y},\vec{z})$$

in the case when the first argument is a set; then the formula ψ_f encoding f uses the symbol S^2 for that set.

Below we treat two most complex cases: $\sum[f]$ and $\prod[f]$.

Encoding $\sum[f]$ Now we consider the case of the summation operator. Assume for the moment that we can write a formula $\exists i\vec{x}.\varphi(\vec{x})$ meaning there are at least i vectors satisfying φ. Then we can also define $\exists! i\vec{x}.\varphi(\vec{x})$, which gives us the encoding of $\varphi_{\sum[f]}$ as follows:

$$\psi_{\sum[f]}(z) \;=\; \exists i.(\exists! i(\vec{x},y,v).S^1(\vec{x}) \wedge \psi_f(\vec{x},y) \wedge S(v,y)) \;\wedge\; \exists! iv.S(v,z).$$

This formula is saying that z is the ith element in S, where i is the number of tuples $(\vec{x},y,v)$ such that $\vec{x}$ is in the input relation S^1, y is the jth element of S where $j=f(\vec{x})$ and v is under y in S. It is easy to see that the number of such tuples is exactly $\sum[f](S^1)$.

Thus, it remains to show how to count tuples, provided that the number of such tuples does not exceed max. Note that for the summation, we need to count tuples of arity up to $m+2$, where m is the maximum arity of a record that can occur in the process of evaluating Q. We show below how to count pairs; counting tuples is similar (only the encoding scheme changes).

To define $\chi(i) = \exists i(x,y).\varphi(x,y)$, we first define

$$\alpha(x,x_0) \;=\; \exists k.[\exists! ky.\varphi(x,y) \wedge \mathsf{adom}_S(x_0) \wedge \exists! kv.S(v,x_0)].$$

Thus, $\alpha(x,x_0)$ holds iff x_0 represents the number of y such that $\varphi(x,y)$ holds. Next, define

$$\beta(x_0,y_0) \;=\; \exists j.[\exists! jz.\alpha(z,x_0) \wedge \mathsf{adom}_S(y_0) \wedge \exists! jv.S(v,y_0)].$$

Now we see that $\chi(i)$ holds iff

$$i \le \sum(k * j \mid \beta(x_0,y_0) \text{ holds, } x_0 \text{ represents } k, \; y_0 \text{ represents } j) = G(i).$$

Thus, if we have a formula $enc_4(x_1,x_2,x_3,x_4,N,z)$ that encodes 4-tuples (x_1,x_2,x_3,x_4) of numbers under N (that is, z is the encoding), where N is $n^{n^{C_f}}$ – the maximal number that can occur in the process of evaluating Q, we define

$$\chi(i) \;=\; \exists iz.\exists x_0 \exists y_0 \exists x' \exists y'.enc_4(x_0,y_0,x',y',N,z) \wedge \beta(x_0,y_0) \wedge S(x',x_0) \wedge S(y',y_0).$$

(Note that N is definable.) That is, we count the number of elements that code 4-tuples (x_0,y_0,x',y') such that $\beta(x_0,y_0)$ holds, x' is under x_0 in S and y' is under y_0 in S. It is easy to see that the number of such zs is precisely $G(i)$.

It is easy to extend this technique to counting m-tuples by counting $m-1$-tuples (by α) first; in particular, one can see that in such a counting one never needs enc_m for $m > 4$.

Encoding $\prod[f]$ The final case is that of $\prod[f]$. Before explaining a rather complex encoding scheme, we present it informally.

Assume that we have a set $X = \{\vec{x}_1, \ldots, \vec{x}_K\}$, and let $n_i = f(\vec{x}_i)$. Let $M = \prod[f](X)$. Then M is the number of k-element bags $B = \{n'_1, \ldots, n'_K\}$ such that $1 \leq n'_i \leq n_i$ for all i.

To find the number of such bags, we have to find their set representation first. Represent the bag $\{n_1, \ldots, n_k\}$ as a set of pairs $X_0 = \{(N_1, m_1), \ldots, (N_s, m_s)\}$ where N_is are among n_js and m_is are their multiplicities. Assume that $N_1 < \ldots < N_s$.

We now represent bags such as B by a set of quadruples $X_B = \{(i, n, j, k)\}$ such that i is a number between 1 and s, n is one of N_is, $k \leq n_i$ and $j < m_i$. This tuple asserts that B contains j occurrences of the value k at indicies l corresponding to the value N_i.

Suppose we can write a formula that says that X_B is indeed a representation of one of the bags B we need to count. Next step is to transform X_B into a set $X'_B = \{e_1, \ldots, e_t\}$ where each e_i is an encoding (as in the summation case) of a quadruple (i, n, j, k). As before, the encoding is relative to N, where N is given by Lemma 10, and is definable in $\mathcal{FO} + COUNT$. Given such a set, we define its Gödel encoding, to represent it as a number. Finally, M is the number of numbers (represented as elements of S) that are Gödel encodings of such sets.

We now describe the formula that defines M. By now, the reader must be convinced that any arithmetic on numbers can be transferred to the elements of S, so we shall now use those elements of S instead of second-sort variables. This will make the notation somewhat more bearable. The reader should be able to see easily how to do everything rigorously by using counting quantifiers and formulae $\mathsf{is}(x, i)$.

We define

$$\psi_{\prod[f]}(x) \quad = \quad \exists i.\mathsf{is}(x, i) \wedge \exists! iv.\mathit{encodes\_good\_set}(v),$$

where $\mathit{encodes\_good\_set}(v)$ means that v is a Gödel encoding of a set of the form X'_B. To define this, we assume four other formulae: $\mathit{is\_enc}(v)$ means that v is an encoding of some set, *in_set(m,v)* means that m is in the set given by its encoding v, $\mathit{good}(m)$ means that m is the encoding of a valid quadruple (i, n, j, k), and *good_sum(v)* meaning that when v is decoded all the way to the set of form X_B, the sum of js corresponding to the ith group is indeed the multiplicity of the corresponding value in X_0. With this, we define $\mathit{encodes\_good\_set}(v)$ as

$$\mathit{is\_enc(v)} \ \wedge \ \mathit{good\_sum(v)} \ \wedge \ \forall m.\mathit{in\_set}(m, v) \rightarrow \mathit{good}(m).$$

The Gödel encoding of a set $\{k_1, \ldots, k_r\}$ with $k_1 < \ldots < k_r$ is $2^{k_1} \cdot 3^{k_2} \cdot \ldots \cdot p_r^{k_r}$ where p_r is the rth prime. Thus, a number V is an encoding if, whenever divisible by a prime p, is divisible by any other prime $p' < p$, and for m', m which are the largest numbers such that V is divisible by $(p')^{m'}$ and p^m, it holds: $m' < m$. This is clearly definable in $\mathcal{FO} + COUNT$, using the formula *factor* produced earlier.

To check if m is in the set encoded by v we therefore look for a prime p such that v is divisible by p^m but not by p^{m+1}. Assuming formulae *prime* testing for prime and $div(x,y)$ as a an abbreviation for $\exists z.z * y = x$, we write $in\_set(m,v)$ as

$$\exists p \exists z.prime(p) \wedge div(v,z) \wedge exp(p,m,z) \wedge (\forall z'.z' = z * p \rightarrow \neg div(v,z'))$$

To define $good(m)$, we must check the existence of (i,n,j,k) such that m encodes (i,n,j,k) with respect to the same base as in the case of summation such that the following holds:

1. n is a value of f on X; that is, $\exists \vec{x}.R(\vec{x}) \wedge \psi_f(\vec{x},n)$. We call this formula $value(n)$.
2. There exist at least i different values: $\exists iw.value(w)$.
3. n is the ith value in the usual order on natural numbers: $\exists l \exists ! lw.(value(w) \wedge w < n) \wedge i = l+1$.
4. j and k are within bounds: j is not more than the number of $\vec{x}$ on which f evaluates to n, and k is at most n. That is, $\exists l.(\exists ! l\vec{x}.R(\vec{x}) \wedge \psi_f(\vec{x},n)) \wedge j \leq l \wedge k \leq n$. Here we count vectors in exactly the same way we did it in the case of summation.

Finally, we have to check that the sum of all js for fixed i is correct; that is, the multiplicity of the ith value of f on $\vec{x}_l$s. To do this, we must check that for every w such that $value(w)$, the number of $\vec{x}$ such that $R(\vec{x}) \wedge \psi_f(\vec{x},w)$ is the same as the number of 5-tuples (i,w,j,k,j') such that (i,w,j,k) satisfies the conditions 1-4 above (substituting w for n), $1 \leq j' \leq j$ and (i,w,j,k) is an encoding of some m such that $in\_set(m,v)$ holds. This clearly can be done in $\mathcal{FO} + COUNT$ if we are allowed to count tuples. Hence, we count tuples as we did before in encoding $\sum[f]$, and this completes the encoding of $\prod[f]$.

This completes the description of the encoding of $\text{AGGR}_{\text{flat}}$ primitives. To encode a query Q, we assume that at the first step, when the function operates on the input structure $\mathcal{A}$, we use the symbols of σ instead of S_is. Then, for each composition, we generate a fresh set of relation symbols. Thus, the encoding of Q is a $\mathcal{FO} + COUNT$ formula in the language $\sigma \cup \{S\}$.

Given a relational query Q in $\text{AGGR}_{\text{flat}}$, the function g and the encoding ψ_Q of Q, we encode Q_g as $\Psi(Q,g) = \psi_Q \wedge test_*$.

Now a straightforward proof by induction on the structure of Q shows that $\Psi(Q,g)$, when given an input $\mathcal{A}_S$ satisfying $(\star_g[\mathcal{A}_S])$ for an appropriately chosen constant c, and having at least two elements in A, defines $Q(\mathcal{A})$. This is because $\mathsf{adom_b}(Q(\mathcal{A})) \subseteq A$ and, furthermore, all numbers produced by the counting formulae in the encoding are below max.

The latter requires verification in the case of summation and product operation. We know that any number produced in the process of evaluation of Q on $\mathcal{A}$ is at most $N = n^{n^{C_f}}$ for appropriately chosen C_f. Thus, the encodings of fixed lenght tuples of such elements are bounded by a value of some polynomial in N, that is, by $n^{n^{C'}}$ for some C'. For the Gödel encoding, we need upper bound on values $P = 2^{k_1} \cdot \ldots \cdot p_r^{k^r}$ where p_r is the rth prime, and both k_r and r are at most N. Thus, P is at most $p_N^{N^2}$. Since there exists a constant d such that $p_k \leq dk \log k$ [20], we obtain

$$P \;\leq\; (dN^2)^{N^2} \;<\; d(n^{n^{C_f}})^{2n^{2n^{C_f}}} \;=\; dn^{2n^{c_f}+n^{C_f}}$$

which shows that there exists a constant such that $P < n^{n^{n^c}}$. This completes the proof that there exists a function g such that $\Psi(Q,g)$ defines $Q(\mathcal{A})$ on inputs satisfying $(\star_g[\mathcal{A}_S])$. Since $test_*$ is a conjunct of $\Psi(Q,g)$, on inputs not satisfying $(\star_g[\mathcal{A}_S])$, $\Psi(Q,g)$ produces the empty set.

This completes the proof that Q_g is definable by a $\mathcal{FO} + COUNT$ formula $\Psi(Q,g)$ if the active domain of $\mathcal{A}$ has at least two elements.

Finally, we consider the case when the active domain of $\mathcal{A}$ is empty or has one element. In the first case the output is empty as well (we do not have access to the individual constants in D), and in the second case it is either a) empty, or b) has a single tuple $(a,\ldots,a)$ where a is the unique element of the active domain. Thus, in the case a) the formula $\psi(\vec{z})$ encoding Q_g for arbitrary $\mathcal{A}$ is defined by

$$[\exists x \exists y.\mathsf{adom}_{\mathcal{A}}(x) \wedge \mathsf{adom}_{\mathcal{A}}(y) \wedge \neg(x = y)] \;\wedge\; \Psi(Q,g)(\vec{z})$$

and in the case b), $\psi(z_1,\ldots,z_m)$ is given by

$$\begin{array}{ll} & (\exists x.\mathsf{adom}_{\mathcal{A}}(x)) \\ \wedge & [\;\; (\exists! x.\mathsf{adom}_{\mathcal{A}}(x) \wedge \bigwedge_{i=1}^{m}(z_i = x)) \\ & \vee\, (\exists x \exists y.\mathsf{adom}_{\mathcal{A}}(x) \wedge \mathsf{adom}_{\mathcal{A}}(y) \wedge \neg(x = y) \wedge \Psi(Q,g)(\vec{z}))\;]. \end{array}$$

This completes the proof of Proposition 8. $\square$

This article was processed using the LaTeX macro package with LLNCS style