Logics with Aggregate Operators

Lauri Hella[1]
University of Helsinki

Leonid Libkin[2]
Bell Labs

Juha Nurmonen[3]
University of Leicester

Limsoon Wong[4]
Kent Ridge Digital Labs

Abstract

We study adding aggregate operators, such as summing up elements of a column of a relation, to logics with counting mechanisms. The primary motivation comes from database applications, where aggregate operators are present in all real life query languages. Unlike other features of query languages, aggregates are not adequately captured by the existing logical formalisms. Consequently, all previous approaches to analyzing the expressive power of aggregation were only capable of producing partial results, depending on the allowed class of aggregate and arithmetic operations.

We consider a powerful counting logic, and extend it with the set of all aggregate operators. We show that the resulting logic satisfies analogs of Hanf's and Gaifman's theorems, meaning that it can only express local properties.

We then consider a database query language that expresses all the standard aggregate constructs found in commercial query languages, and show how it can be translated into the aggregate logic, thereby providing a number of expressivity bounds, that do not depend on a particular class of arithmetic functions, and that subsume all those previously known. We consider a restricted aggregate logic that gives us a tighter capture of database languages, and also use it to show that some questions on expressivity of aggregation cannot be answered without resolving some deep problems in complexity theory.

Keywords: Finite-model theory, database theory, aggregation, expressive power.

[1]Department of Mathematics, P.O. Box 4 (Yliopistonkatu 5), 00014 University of Helsinki, Finland, Email: lauri.hella@helsinki.fi. Supported in part by grant 40734 from the Academy of Finland.

[2]Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, USA, Email: libkin@bell-labs.com. Part of this work was done while visiting INRIA.

[3]Department of Mathematics and Computer Science, University of Leicester, University Road, Leicester LE1 7RH, UK, Email: j.nurmonen@mcs.le.ac.uk. Supported in part by EPSRC grant GR/K 96564.

[4]Kent Ridge Digital Labs, 21 Heng Mui Keng Terrace, Singapore 119613, Email: limsoon@krdl.org.sg. Part of this work was done while visiting Bell Labs.

1 Introduction

First-order logic over finite structures plays a fundamental role in several computer science applications, perhaps most notably, in database theory. The standard theoretical query languages – relational algebra and calculus – that are the backbone for the commercial query languages, have precisely the power of first-order logic. However, while this power is sufficient for writing many useful queries, in practice one often finds that it is quite limited for two reasons. Firstly, in first-order logic, one cannot do fixpoint computation (for example, one cannot compute the transitive closure of a graph). Secondly, one cannot express nontrivial counting properties (for example, one cannot compare the cardinalities of two sets).

From the practical point of view, fixpoint computation, although sometimes desirable, is of less importance in the database context than counting. Indeed, in the de-facto standard of the commercial database world, SQL, a limited recursive construct has only been proposed for the latest language standard (SQL3). At the same time, constructs such as cardinality of a relation or the average value of a column, known as *aggregate functions*, are present in any commercial implementation of SQL (they belong to what is called the entry level SQL92, which is supported by all systems).

On the theory side, however, fixpoint extensions of first-order logic and corresponding query languages are much better studied than their counting counterparts. A standard fixpoint extension considered in the database literature is the query language datalog, and practically every aspect of it – expressive power, optimization, adding negation, implementation techniques – were the subject of numerous papers. For the study of expressive power of query languages, which will interest us most in this paper, a very nice result of [19] showed that the infinitary logic with finitely many variables, $\mathcal{L}^{\omega}_{\infty\omega}$, has a 0-1 law over finite structures. As other fixpoint logics can be embedded into it, this result gives many expressivity bounds for datalog-like languages. In the presence of an order relation, it is again a classical result that various fixpoint extensions of first-order logic capture familiar complexity classes such as PTIME and PSPACE. See [1, 7] for an overview.

For extensions with counting and aggregate, much less is known, especially in terms of expressive power of languages. In an early paper [17] it was shown how to extend both relational algebra and calculus with aggregate constructs, but the resulting language did not correspond naturally to any reasonable logic. It is known how to integrate aggregation into datalog-like languages (both recursive and nonrecursive) [30, 33], and various aspects of such aggregate languages were studied (e.g., query optimization [26] and handling constraints involving aggregation [31]).

At this point, let us give an example of a typical aggregate query that would be supported by all commercial versions of SQL, and use it to explain problems that arise when one attempts to analyze expressiveness of the language. Suppose we have two database relations: a relation `R1` with attributes "employee" and "department", and a relation `R2` with attributes "employee" and "salary". Suppose we want to find the average salary for each department that pays total salary at least $\$10^6$. In SQL, this is done as follows.

```
SELECT R1.Dept, AVG(R2.Salary)
FROM R1, R2
WHERE R1.Employee = R2.Employee
GROUPBY R1.Dept
HAVING SUM(R2.Salary) > 1000000
```

Relations `R1` and `R2` separate the information about departments and salaries. This query joins them to put together departments, employees, and salaries, and then performs an aggregation over the salary column, for each department in the database, followed by selecting some of the resulting tuples.

While the features of the language given by the `SELECT`, `FROM` and `WHERE` clauses are well-known to be first-order, other features used in this example pose a problem. First of all, we permit computation of aggregate operators such as `AVG` and `SUM` over the entire column of a relation. This form of counting is rather different from the counting quantifiers or terms (see, e.g, [8, 16, 29]), normally supported by logical formalisms. Secondly, the `GROUPBY` clause creates an intermediate structure which is a set of sets – for each department, it groups together all its employees. Again, this feature does not get captured adequately by existing logical formalisms.

This shows why it is hard to capture aggregation in query languages by a logic whose expressive power is easy to analyze. Still, there exist some partial results. For example, [27] gives some bounds based on the estimates on the largest number a query can produce; clearly such bounds are not robust and do not withstand adding arithmetic operations. In [5] it is shown that the transitive closure of a graph is not expressible in the aggregate extension of first-order logic if DLOGSPACE $\neq$ NLOGSPACE. In [24], this is proved without any complexity assumptions; a generalization of [24] to many other queries is given in [6]. One problem with the proofs of [24, 6] is that they are very "syntactic" – they work for a particular presentation of the language, and rely heavily on complicated syntactic rewritings of queries, rather than on the semantic properties of those. An attempt to remedy this was made in [21] which considered a sublanguage that only permits aggregation over columns of natural numbers (for example, `AVG` is not allowed). Then [21] gave a somewhat complicated encoding of the language in first-order logic with counting quantifiers, for which expressivity bounds are known [21, 28]. The encoding of [21] was extended to aggregation over rational numbers [25]; it did allow more aggregates (e.g., `AVG`) and more arithmetic, at the expense of a very unpleasant and complicated encoding procedure.

This shows that first-order logic with counting quantifiers is inadequate as a logic for expressing aggregate query languages. It also brings up an analogy with the development of datalog-like languages and $\mathcal{L}^{\omega}_{\infty\omega}$, and raises the following question: Can we find a powerful logic into which aggregate queries can be *easily* embedded, and whose properties can be analyzed so that bounds for query languages can be derived?

Our main goal is to give the positive answer to this question. To do so, we combine a powerful infinitary counting logic from [22] with an elegant framework of [12] for adding aggregation. As the numerical domain, we choose the set of rational numbers $\mathbb{Q}$, although any other domain can be chosen. The resulting logic $\mathcal{L}_{\text{aggr}}$ defines every arithmetic operation and every aggregate function. We then show that it has very nice behavior: its formulae satisfy analogs of Hanf's [9, 13] and Gaifman's [11] theorems, meaning that it can only express *local* properties. In particular, properties such as connectivity of graphs cannot be expressed.

We then consider a theoretical language $\mathcal{RL}^{\text{aggr}}$, similar to those defined in [3, 24], and explain how it models all the features of SQL. Next, we show an embedding of $\mathcal{RL}^{\text{aggr}}$ into $\mathcal{L}_{\text{aggr}}$, which is much simpler than those previously considered for first-order with counting [21, 25]. This implies that the behavior of aggregate queries is *local* over a large class of inputs, no matter what family of aggregate and arithmetic operations the language possesses.

Not only is this result much stronger than all previous results on expressiveness of aggregation, it is also proved in a much nicer way. Furthermore, we believe that logics with aggregation are interesting on their own right, as they give a rather disciplined approach to modeling aggregation and can be used to study other aspects of it, for example, optimization of aggregate queries.

Organization We give notations, including two-sorted structures and a formal definition of aggregates in Section 2. In Section 3 we give the definition of the aggregate logic $\mathcal{L}_{\text{aggr}}$. In Section 4 we explain the locality theorems of Hanf and Gaifman and prove that $\mathcal{L}_{\text{aggr}}$ satisfies analogs of both of them.

In Section 5, we define an aggregate query language $\mathcal{NRL}^{\text{aggr}}$, on nested relations, that models both aggregation and grouping features of SQL. We show, using standard techniques, that queries from flat relations to flat relations in this language can be expressed in a simpler language called $\mathcal{RL}^{\text{aggr}}$, that does not use nested relations even as intermediate structures, and then we give a translation of $\mathcal{RL}^{\text{aggr}}$ into $\mathcal{L}_{\text{aggr}}$. This shows that $\mathcal{NRL}^{\text{aggr}}$ queries over flat databases that do not contain numbers are local. In Section 6 we consider a simpler logic L_{aggr}

and show that it captures the language $\mathcal{RL}^{\mathrm{aggr}}$. We also show that some basic questions about expressive power of $\mathcal{RL}^{\mathrm{aggr}}$ cannot be answered without resolving some deep problems in complexity theory, under the assumption that input databases are allowed to contain numbers.

2 Notation

Most logics we consider here are two-sorted, and they are defined on two-sorted structures, with one sort being numerical. We shall assume, throughout the paper, that the numerical sort is interpreted as $\mathbb{Q}$, the set of rational numbers. A two sorted relational signature σ is a finite collection $\{R_1(n_1, J_1), \ldots, R_l(n_l, J_l)\}$ where R_is are relation names, n_is are their arities, and $J_i \subseteq \{1, \ldots, n_i\}$ is the set of indices for the first sort. For example, $\{R(3, \{1, 2\})\}$ is a signature that consists of a single ternary relation so that in each tuple (a, b, c) in R, a, b are of the first sort and c is of the second sort.

We let $\mathcal{U}$ be an infinite set, disjoint from $\mathbb{Q}$, to be interpreted as the domain of the first sort. A *structure* of signature σ (or σ-structure) is $\mathcal{A} = \langle A, \mathbb{Q}, R_1^{\mathcal{A}}, \ldots, R_l^{\mathcal{A}}\rangle$, where $A \subseteq \mathcal{U}$ is the universe of the first sort for $\mathcal{A}$, and $R_i^{\mathcal{A}} \subseteq \prod_{k=1}^{n_i} \mathrm{dom}(i, k)$, where $\mathrm{dom}(i, k) = A$ if $k \in J_i$ and $\mathrm{dom}(i, k) = \mathbb{Q}$ if $k \notin J_i$. We shall always assume that A is *finite*.

We use the following notation: $\{\!|X|\!\}_n$ is the set of all n-element multisets (bags) over X. For the multiset containing precisely the elements $x_1, \ldots, x_n$, we use the notation $\{\!|x_1, \ldots, x_n|\!\}$.

Now, following [12], we define an *aggregate function* as a collection $\mathcal{F} = \{f_0, f_1, f_2, \ldots, f_\omega\}$ where $f_n : \{\!|\mathbb{Q}|\!\}_n \to \mathbb{Q}$, and $f_\omega \in \mathbb{Q}$. Each function f_i shows how the aggregate function behaves on an n-element input multiset of rational numbers, and the value f_ω is the result when the input is infinite.

Examples include the aggregates $\sum$ and $\prod$: $\sum = \{s_0, s_1, \ldots, s_\omega\}$ and $\prod = \{p_0, p_1, \ldots, p_\omega\}$ where $s_0 = 0$ and $s_n(\{\!|q_1, \ldots, q_n|\!\}) = q_1 + \ldots + q_n$, and $p_0 = 1$ and $p_n(\{\!|q_1, \ldots, q_n|\!\}) = q_1 \cdot \ldots \cdot q_n$. (The values s_ω and p_ω are defined to be 0.) Standard database languages use other aggregates; in fact, standard ones for SQL are $\sum$, MIN, MAX defined as the minimum (maximum) element of the input bag, COUNT, whose ith function is the constant i, and AVG, whose ith function is s_i/i for $i > 0$.

3 An aggregate logic

Assume that we are given two signatures on $\mathbb{Q}$: one, denoted by Ω, of functions and predicates, and one, denoted by Θ, of aggregates. In addition we assume that there is a constant symbol c_q for each $q \in \mathbb{Q}$. We now define an aggregate logic $\mathcal{L}_{\mathrm{aggr}}(\Omega, \Theta)$, on two-sorted structures. We do it, similarly to [22], in two steps. We first define a larger logic $\mathbb{L}_{\mathrm{aggr}}(\Omega, \Theta)$ and then put a restriction on its formulae.

We define terms and formulae of the two-sorted logic $\mathbb{L}_{\mathrm{aggr}}(\Omega, \Theta)$, over two-sorted structures, by simultaneous induction. Every variable of the ith sort is a term of the ith sort, $i = 1, 2$. Every constant $c_q \in \mathbb{Q}$ is a term of the second sort. Given a pair (n, J) with $J \subseteq \{1, \ldots, n\}$, we say that an n-tuple of terms $\vec{t} = (t_1, \ldots, t_n)$ if of type (n, J), written $\vec{t} : (n, J)$, if t_i is a first-sort term for $i \in J$ and a second-sort term for $i \notin J$. For a formula $\varphi(\vec{x})$, we write $\varphi : (n, J)$ and say that its type is (n, J) if $\vec{x} = (x_1, \ldots, x_n)$ and $i \in J$ iff x_i is of the first sort.

Now for each $R_i(n_i, J_i)$ in σ, and $\vec{t} : (n_i, J_i)$, we let $R(\vec{t})$ be a formula. Formulae are then closed under *infinitary* disjunctions $\bigvee$ and conjunctions $\bigwedge$, negation $\neg$, and quantifiers over both first-sort domain A and second-sort domain $\mathbb{Q}$.

If $t_1, \ldots, t_n$ are second-sort terms, and f an n-ary function symbol from Ω, then $f(t_1, \ldots, t_n)$ is a second-sort term. For an n-ary predicate symbol P from Ω, $P(t_1, \ldots, t_n)$ is a formula, as well as $t_1 = t_2$ for terms of either sort.

Next, we add counting and aggregation to the logic. For any formula $\varphi(\vec{x}, \vec{y})$ with $\vec{y}$ being variables of the first sort, we let $t(\vec{x}) = \#\vec{y}.\varphi(\vec{x}, \vec{y})$ be a second-sort term. Let $\mathcal{F}$ be an aggregate from Θ. Let $\varphi(\vec{x}, \vec{y})$ be a formula, and $t(\vec{x}, \vec{y})$ a second-sort term. Then $\mathsf{Aggr}_{\mathcal{F}}\vec{y}.(\varphi, t)$ is a second-sort term with free variables $\vec{x}$.

We now discuss the semantics. A tuple $\vec{a} = (a_1, \ldots, a_n)$ is of type (n, J) if $a_i \in \mathcal{U}$ for $i \in J$ and $a_i \in \mathbb{Q}$ for $i \notin J$. For every two-sorted σ-structure $\mathcal{A}$, a formula $\varphi(\vec{x})$ or a term $t(\vec{x})$ of type (n, J) in the language of σ, and a tuple $\vec{a}$ over $A \cup \mathbb{Q}$ of type (n, J), we define the value $t^{\mathcal{A}}(\vec{a})$ of the term t on $\vec{a}$ in $\mathcal{A}$ and the relation $\mathcal{A} \models \varphi(\vec{a})$. The definition is standard, with only the case of counting terms and aggregation requiring explanation. For $t(\vec{x}) = \#\vec{y}.\varphi(\vec{x}, \vec{y})$, the value of $t(\vec{a})$ in $\mathcal{A}$ is the (finite) number of $\vec{b}$ over A such that $\mathcal{A} \models \varphi(\vec{a}, \vec{b})$.

Let $s(\vec{x}) = \mathsf{Aggr}_{\mathcal{F}}\vec{y}.(\varphi(\vec{x}, \vec{y}), t(\vec{x}, \vec{y}))$, and let $\vec{a}$ be of the same type as $\vec{x}$. Define $\varphi(\vec{a}, \mathcal{A}) = \{\vec{b} \mid \mathcal{A} \models \varphi(\vec{a}, \vec{b})\}$. Let $t(\varphi(\vec{a}, \mathcal{A}))$ be the *multiset* $\{\!| t^{\mathcal{A}}(\vec{a}, \vec{b}) \mid \vec{b} \in \varphi(\vec{a}, \mathcal{A}) |\!\}$. (This is a multiset since t may produce identical values on several $(\vec{a}, \vec{b})$.) Let n be the cardinality of this multiset. Then the value $s^{\mathcal{A}}(\vec{a})$ is defined to be $f_n(t(\varphi(\vec{a}, \mathcal{A})))$, where f_n is the nth component of $\mathcal{F}$. If the set $\varphi(\vec{a}, \mathcal{A})$ is infinite, then the value of $s^{\mathcal{A}}(\vec{a})$ is f_ω.

This concludes the definition of $\mathbb{L}_{\text{aggr}}(\Omega, \Theta)$. Next, we define the notion of a rank of formulae and terms, $\mathsf{rk}(\varphi)$ and $\mathsf{rk}(t)$. For a variable or constant t, $\mathsf{rk}(t) = 0$. For any formula $\varphi \equiv P(t_1, \ldots, t_n)$ with $P \in \Omega$, we have $\mathsf{rk}(\varphi) = \max_i \mathsf{rk}(t_i)$, and similarly for a term $f(\vec{t})$ with f from Ω. We then have $\mathsf{rk}(\bigvee \varphi_i) = \sup_i \mathsf{rk}(\varphi_i)$ and $\mathsf{rk}(\neg\varphi) = \mathsf{rk}(\varphi)$.

We let $\mathsf{rk}(\exists x \varphi) = \mathsf{rk}(\varphi) + 1$, for quantification over the first sort, and $\mathsf{rk}(\exists q \varphi) = \mathsf{rk}(\varphi)$ for quantification over the second sort. For counting and aggregate terms, $\mathsf{rk}(\#\vec{y}.\varphi) = \mathsf{rk}(\varphi) + \mid \vec{y} \mid$ and $\mathsf{rk}(\mathsf{Aggr}_{\mathcal{F}}\vec{y}.(\varphi, t)) = \max(\mathsf{rk}(\varphi), \mathsf{rk}(t)) + \mid \vec{y} \mid$.

Definition 3.1 *The formulae and terms of $\mathcal{L}_{\text{aggr}}(\Omega, \Theta)$ are precisely the formulae and terms of $\mathbb{L}_{\text{aggr}}(\Omega, \Theta)$ that have finite rank. If there is no restriction on the signature (that is, all functions and predicates are allowed, we write* All*. Thus, $\mathcal{L}_{\text{aggr}}(\mathsf{All}, \mathsf{All})$ is the aggregate logic in which every function, predicate, and aggregate function on $\mathbb{Q}$ is available.* □

Examples We first note that the counting terms can be expressed using the $\sum$ aggregate. Indeed, $\#\vec{y}.\varphi(\vec{x}, \vec{y})$ is equivalent to $\mathsf{Aggr}_\Sigma\vec{y}.(\varphi(\vec{x}, \vec{y}), c_1)$.

Next, we show how to express the example from the introduction in $\mathcal{L}_{\text{aggr}}(\{<\}, \{\sum, \mathsf{AVG}\})$. The signature σ has two relations: $R_1(2, \{1, 2\})$ and $R_2(2, \{1\})$, as only the last attribute of R_2 – salary – is numerical. The query is now expressed as a $\mathcal{L}_{\text{aggr}}$ formula $\varphi(x, q)$ with two free variables, one of the first sort, one of the second sort, as follows:

$$\begin{array}{ll} & (q = \mathsf{Aggr}_{\mathsf{AVG}}\ z.(\exists y. R_1(x, y) \wedge R_2(y, z), z)) \\ \wedge & (\mathsf{Aggr}_\Sigma\ z.(\exists y. R_1(x, y) \wedge R_2(y, z), z)\ >\ c_{10^6}). \end{array}$$

4 Aggregate logic: Expressive power

In this section we deal with expressiveness of the aggregate logic. Our main goal is to show that it satisfies a very strong locality property. Locality properties were introduced in model theory by Hanf [13] and Gaifman [11], and recently, following [9], they were a subject of renewed attention (see, e.g., [6, 21, 22, 24, 28] and references therein). Intuitively, those properties say that the behavior of logical formulae depends on the structure of small neighborhoods. They imply strong expressivity bounds for queries definable by logical formulae. For example, if we deal with queries on graphs, then the number of different degrees of nodes realized in the output does not exceed a bound that is determined only by a formula defining the query, and the maximum degree of the input graph, but not the size of the graph [24].

As there are several ways to define locality, we want to establish the strongest property. The relationship between various notions of locality was investigated in [15, 21], and it was shown that the one based on Hanf's theorem implies the one based on Gaifman's theorem, which in turn implies the property stated in the previous paragraph. Thus, our goal is to show (precise definition will be given a bit later in this section):

Expressiveness of $\mathcal{L}_{\text{aggr}}(\mathsf{All}, \mathsf{All})$: *Over first-sort structures, formulae of $\mathcal{L}_{\text{aggr}}(\mathsf{All}, \mathsf{All})$ are Hanf-local.*

It is known that many properties requiring fixpoint computation, such as the connectivity and acyclicity tests for graph, or computing the transitive closure, violate some forms of locality. Thus, as a corollary, we shall see that adding unlimited arithmetic and aggregation to first-order logic does not enable it to express those properties.

We start by showing how to embed $\mathcal{L}_{\text{aggr}}(\mathsf{All}, \mathsf{All})$ into a simpler logic $\mathcal{L}_{\mathbf{C}}$ that does not have aggregate operations. We then review the main notions of locality used in finite-model theory, and prove the strongest of them, Hanf-locality, of $\mathcal{L}_{\mathbf{C}}$.

4.1 Logic $\mathcal{L}_{\mathbf{C}}$

Definition 4.1 *The logic $\mathcal{L}_{\mathbf{C}}$ is defined to be $\mathcal{L}_{\text{aggr}}(\emptyset, \emptyset)$; that is, aggregate terms are not allowed.*

A weaker version of this logic was studied in [22]. That logic, denoted by $\mathcal{L}^*_{\infty\omega}(\mathbf{C})$, was defined as $\mathcal{L}_{\mathbf{C}}$ over one-sorted structures and the set $\mathbb{N}$ of natural numbers as the numerical domain.

For two logics we write $\mathcal{L}_1 \preccurlyeq \mathcal{L}_2$ if $\mathcal{L}_2$ is at least as powerful as $\mathcal{L}_1$. If for every formula in $\mathcal{L}_1$ there is an equivalent one in $\mathcal{L}_2$ of the same or smaller rank, we write $\mathcal{L}_1 \preccurlyeq_{\mathsf{rk}} \mathcal{L}_2$. We use $\mathcal{L}_1 \approx \mathcal{L}_2$ if $\mathcal{L}_1 \preccurlyeq \mathcal{L}_2$ and $\mathcal{L}_2 \preccurlyeq \mathcal{L}_1$, and likewise for $\mathcal{L}_1 \approx_{\mathsf{rk}} \mathcal{L}_2$.

Theorem 4.2 $\mathcal{L}_{\text{aggr}}(\mathsf{All}, \mathsf{All}) \approx_{\mathsf{rk}} \mathcal{L}_{\mathbf{C}}$.

Proof. See the appendix. □

4.2 Notions of locality in finite models

In this section, we only consider one-sorted finite structures $\mathcal{A} = \langle A, R_1^{\mathcal{A}}, \ldots, R_l^{\mathcal{A}} \rangle$ and two-sorted structures over signatures σ that that do not contain relation symbols with mixed sorts (i.e., we assume that $J_i = \{1, \ldots, n_i\}$ for every $R(n_i, J_i) \in \sigma$). We call such two-sorted structures *pure*. Note that each one-sorted finite structure $\mathcal{A}$ can be extended to pure two-sorted structure simply by adding the set $\mathbb{Q}$ as the second sort (and interpreting the constant symbols c_q in the canonical way). We denote this extension of $\mathcal{A}$ by $\mathcal{A}^*$.

Given a finite one-sorted structure $\mathcal{A}$, its *Gaifman graph* [7, 11, 9] $\mathcal{G}(\mathcal{A})$ is defined as $\langle A, E \rangle$ where (a, b) is in E iff there is a tuple $\vec{c} \in R_i^{\mathcal{A}}$ for some i such that both a and b are in $\vec{c}$. 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 $d(a, a) = 0$. If $\vec{a} = (a_1, \ldots, a_n)$ and $\vec{b} = (b_1, \ldots, b_m)$, then $d(\vec{a}, \vec{b}) = \min_{ij} d(a_i, b_j)$. Given $\vec{a}$ over A, its *r-sphere* $S_r^{\mathcal{A}}(\vec{a})$ is $\{b \in A \mid d(\vec{a}, b) \le r\}$. Its *r-neighborhood* $N_r^{\mathcal{A}}(\vec{a})$ is defined as a structure in the signature that consists of σ and n constant symbols:

$$\langle S_r^{\mathcal{A}}(\vec{a}), R_1^{\mathcal{A}} \cap S_r^{\mathcal{A}}(\vec{a})^{p_1}, \ldots, R_k^{\mathcal{A}} \cap S_r^{\mathcal{A}}(\vec{a})^{p_k}, a_1, \ldots, a_n \rangle$$

That is, the carrier of $N_r^{\mathcal{A}}(\vec{a})$ is $S_r^{\mathcal{A}}(\vec{a})$, the interpretation of the σ-relations is inherited from $\mathcal{A}$, and the n extra constants are the elements of $\vec{a}$. If $\mathcal{A}$ is understood, we write $S_r(\vec{a})$ and $N_r(\vec{a})$.

Given a tuple $\vec{a}$ of elements of A, and $d \ge 0$, by $ntp_d^{\mathcal{A}}(\vec{a})$ we denote the isomorphism type of $N_d^{\mathcal{A}}(\vec{a})$. Then $ntp_d^{\mathcal{A}}(\vec{a}) = ntp_d^{\mathcal{B}}(\vec{b})$ means that there is an isomorphism $N_d^{\mathcal{A}}(\vec{a}) \to N_d^{\mathcal{B}}(\vec{b})$ that sends $\vec{a}$ to $\vec{b}$; in this case we will also write $\vec{a} \approx_d^{\mathcal{A},\mathcal{B}} \vec{b}$. If $\mathcal{A} = \mathcal{B}$, we write $\vec{a} \approx_d^{\mathcal{A}} \vec{b}$. Given a tuple $\vec{a} = (a_1, \ldots, a_n)$ and an element c, we write $\vec{a}c$ for $(a_1, \ldots, a_n, c)$.

For two σ-structures $\mathcal{A}, \mathcal{B}$, we write $\mathcal{A} \leftrightarrows_d \mathcal{B}$ if there exists a bijection $f : A \to B$ such that $ntp_d^{\mathcal{A}}(a) = ntp_d^{\mathcal{B}}(f(a))$ for every $a \in A$. That is, every isomorphism type of a d-neighborhood of a point has equally many realizers in $\mathcal{A}$ and $\mathcal{B}$. We write $(\mathcal{A}, \vec{a}) \leftrightarrows_d (\mathcal{B}, \vec{b})$ if there is a bijection $f : A \to B$ such that $ntp_d^{\mathcal{A}}(\vec{a}c) = ntp_d^{\mathcal{B}}(\vec{b}f(c))$ for every $c \in A$.

Hanf-locality has been previously defined only for finite one-sorted structures. In the following we make a natural extension of its definition to the case of pure two-sorted structures.

Definition 4.3 (see [13, 9, 21, 15]) *A formula $\varphi(\vec{x})$ on pure two-sorted structures is called* Hanf-local *if there exist a number $d \ge 0$ such that for all finite one-sorted structures $\mathcal{A}$ and $\mathcal{B}$, $(\mathcal{A}, \vec{a}) \leftrightarrows_d (\mathcal{B}, \vec{b})$ implies $\mathcal{A}^* \models \varphi(\vec{a})$ iff $\mathcal{B}^* \models \varphi(\vec{b})$.*

The definition for open formulae is from [15]; most previous papers [13, 9, 21, 28] considered its restriction to sentences. It is known [9] that $\mathcal{A} \leftrightarrows_d \mathcal{B}$ implies $\mathcal{A} \leftrightarrows_r \mathcal{B}$ for $r \le d$; in particular, if $\mathcal{A} \leftrightarrows_d \mathcal{B}$, then

$|A|=|B|$. It is also known that every (one-sorted) first-order sentence Φ is Hanf-local and d can be taken to be $3^{\mathsf{qr}(\Phi)-1}$ [9]. This was generalized to first-order with counting in [28], and further to an extension with unary quantifiers in [15]; a tight upper bound of $2^{\mathsf{qr}(\Phi)-1}-1$ was proved in [22].

Definition 4.4 (cf. [21, 22]) *A formula $\varphi(\vec{x})$ on pure two-sorted structures is called* Gaifman-local *if there exists a number $r \geq 0$ such that, for any finite one-sorted structure $\mathcal{A}$ and any $\vec{a}, \vec{b}$ over A, $\vec{a} \approx_r^{\mathcal{A}} \vec{b}$ implies that $\mathcal{A}^* \models \varphi(\vec{a})$ iff $\mathcal{A}^* \models \varphi(\vec{b})$.*

Gaifman's theorem [11] implies this notion of locality for first-order formulae, with a $(7^{\mathsf{qr}(\varphi)}-1)/2$ bound for r; in [22] a tight bound of $2^{\mathsf{qr}(\varphi)}-1$ is established. Furthermore, [15, 21] show that on one-sorted finite structures, every Hanf-local formula is Gaifman-local. These results are not affected by the transfer to pure two-sorted structures.

It is known that connectivity of graphs is not a Hanf-local property [9], and that the transitive closure of a graph is not Gaifman-local [11, 6]. Locality – either Gaifman or Hanf – implies a number of results that describe outputs of local queries by relating degrees of elements in the input and output. For example, the number of degrees realized in the output is bounded by a number that depends on a formula defining a query and the maximum degree in the input [6, 22].

4.3 Locality of $\mathcal{L}_{\mathbf{C}}$

In [28], it is proved that the extension of first-order logic by all unary generalized quantifiers is Hanf-local. The proof of this is based on *bijective Ehrenfeucht-Fraïssé games* [14] which characterize equivalence of structures with respect to unary quantifiers. We now use these games here in proving the Hanf-locality of $\mathcal{L}_{\mathbf{C}}$.

Let $\mathcal{A}$ and $\mathcal{B}$ be two σ-structures, $\vec{a} \in A^n$, and $\vec{b} \in B^n$. The r-round bijective game $\mathrm{BEF}_r(\mathcal{A}, \vec{a}, \mathcal{B}, \vec{b})$ is played by two players, called the spoiler and the duplicator. In each round $i = 1, \ldots, r$, the duplicator selects a bijection $f_i : A \to B$, and the spoiler selects an element $c_i \in A$ (if $|A| \neq |B|$, then the spoiler wins). After each round i, these moves determine the relation $p_i = p_0 \cup \{(c_j, f_j(c_j)) \mid 1 \leq j \leq i\}$, where p_0 is the initial relation $\{(a_j, b_j) \mid 1 \leq j \leq n\}$ between the components of $\vec{a}$ and $\vec{b}$. The spoiler wins the game, if for some i, p_i is not a partial isomorphism $\mathcal{A} \to \mathcal{B}$; otherwise the duplicator wins.

Lemma 4.5 *Let $\mathcal{A}$ and $\mathcal{B}$ be finite one-sorted σ-structures, $\vec{a} \in A^n$, $\vec{b} \in B^n$, and let $\mathcal{A}^*$ and $\mathcal{B}^*$ be the corresponding pure two-sorted structures. If the duplicator has a winning strategy in $\mathrm{BEF}_r(\mathcal{A}, \vec{a}, \mathcal{B}, \vec{b})$, then for every formula $\varphi(x_1, \ldots, x_n)$ in $\mathcal{L}_{\mathbf{C}}$, with $\mathsf{rk}(\varphi) \leq r$ and all free variables of the first sort, $\mathcal{A}^* \models \varphi(\vec{a})$ if and only if $\mathcal{B}^* \models \varphi(\vec{b})$.* □

We now prove the desired locality result.

Theorem 4.6 *Over pure two-sorted structures, every formula of $\mathcal{L}_{\mathbf{C}}$ without free second sort variables is Hanf-local.*

Proof. Let $\varphi(\vec{x})$ be a formula of $\mathcal{L}_{\mathbf{C}}$, where $\vec{x}$ are first-sort variables and $\mathsf{rk}(\varphi) = r$. Let $\mathcal{A}$ and $\mathcal{B}$ be finite one-sorted σ-structures, and let $\vec{a} \in A^n$ and $\vec{b} \in B^n$. It was proved in [28] that if $(\mathcal{A}, \vec{a}) \leftrightarrows_d (\mathcal{B}, \vec{b})$ for $d = 3^r$, then the duplicator has a winning strategy in the bijective game $\mathrm{BEF}_r(\mathcal{A}, \vec{a}, \mathcal{B}, \vec{b})$, and hence by Lemma 4.5, $\mathcal{A}^* \models \varphi(\vec{a})$ if and only if $\mathcal{B}^* \models \varphi(\vec{b})$. Thus φ is Hanf-local. □

By Theorem 4.2, we get as a consequence the Hanf-locality of the full aggregate logic $\mathcal{L}_{\mathrm{aggr}}(\mathsf{All}, \mathsf{All})$.

Corollary 4.7 *Over pure two-sorted structures, all formulas of $\mathcal{L}_{\mathrm{aggr}}(\mathsf{All}, \mathsf{All})$ without free second-sort variables are Hanf-local.* □

As we said earlier, Hanf-locality is a very strong form of locality that implies others, and consequently it gives us many expressivity bounds. Some of them are listed below. For a general overview of deriving expressiveness results from locality, see [6, 9, 11, 21, 24].

Corollary 4.8 *a) Over pure two-sorted structures, all formulas of* $\mathcal{L}_{\text{aggr}}(\mathsf{All}, \mathsf{All})$ *without free second-sort variables are Gaifman-local.*
b) None of the following can be expressed in $\mathcal{L}_{\text{aggr}}(\mathsf{All}, \mathsf{All})$ *over graphs on the universe of the first sort: transitive closure, deterministic transitive closure, connectivity test, acyclicity test, the same-generation property for nodes in acyclic graphs, testing for balanced* k*-ary tree,* $k \geq 1$. □

Thus, despite its enormous counting power, $\mathcal{L}_{\text{aggr}}(\mathsf{All}, \mathsf{All})$ cannot express nonlocal properties, among them most properties requiring fixpoint computations.

5 Database query languages and $\mathcal{L}_{\text{aggr}}$

The goal of this section is to show how standard SQL features can be coded in $\mathcal{L}_{\text{aggr}}$, thereby providing bounds on the expressive power of database queries with aggregation. The coding that we exhibit here is not only more general but also much simpler and more intuitive than that of [21, 25], thanks to the design of $\mathcal{L}_{\text{aggr}}$ that does not limit available arithmetic operations and makes it easy to code aggregation.

We define a relational query language $\mathcal{RL}^{\text{aggr}}(\Omega, \Theta)$, which extends standard relational query languages, such as relational algebra and calculus, with aggregation constructs. The language is parameterized by a collection of allowed arithmetic functions and predicates Ω and a collection of allowed aggregates Θ. We assume that the usual arithmetic operations ($+$, $-$, $*$, $\div$) are always in Ω and the summation aggregate ($\sum$) is always in Θ.

The language is defined as a suitable restriction of a *nested* relational language $\mathcal{NRL}^{\text{aggr}}(\Omega, \Theta)$, in the same way it was done previously [24, 25]. The type system is given by

$$\begin{array}{rcl} \textsc{Base} & := & b \mid \mathbb{Q} \\ rt & := & \textsc{Base} \times \ldots \times \textsc{Base} \\ ft & := & rt \mid \{rt\} \\ t & := & \textsc{Base} \mid t \times \ldots \times t \mid \{t\} \end{array}$$

The base types are b and $\mathbb{Q}$, with the domain of b being an infinite set $\mathcal{U}$, disjoint from $\mathbb{Q}$. We use $\times$ for product types; the semantics of $t_1 \times \ldots \times t_n$ is the cartesian product of domains of types $t_1, \ldots, t_n$. The semantics of $\{t\}$ is the finite powerset of elements of type t. Types rt (record types) and ft (flat types) are used in restrictions that define $\mathcal{RL}^{\text{aggr}}$.

A database *schema* is a list of names of database relations (which may be nested relations) together with their types. We are particularly interested in the case of schemas consisting of *flat* relations, that is, those of types $\{rt\}$. Such a list of names of relations and their flat types naturally corresponds to a two-sorted signature. Indeed, a relation of type $t = \{b_1 \times \ldots \times b_n\}$, with each b_i being either b or $\mathbb{Q}$, corresponds to $R(n, J)$ where $J = \{i \mid b_i = b\}$.

We thus identify flat schemas and two-sorted signatures. Also, for each relational symbol $R(n, J)$ in a two-sorted signature σ, we write $\text{tp}_\sigma(R)$ for its type, that is, $\{b_1 \times \ldots \times b_n\}$ where $b_i = b$ for $i \in J$ and $b_i = \mathbb{Q}$ for $i \notin J$.

Expressions of the language (over a fixed schema σ) are shown in Figure 1. We adopt the convention of omitting the explicit type superscripts in these expressions whenever they can be inferred from the context. The complete definitions of the concept of a free variable of an expression and the semantics of the language are in the appendix; here we explain the main features. The set of free variables of an expression e is defined by induction on the structure of e and we often write $e(x_1, \ldots, x_n)$ to explicitly indicate that x_1, ..., x_n are free variables of e. Expressions $\bigcup\{e_1 \mid x \in e_2\}$, $\sum\{e_1 \mid x \in e_2\}$, and $\mathsf{Aggr}_{\mathcal{F}}\{e_1 \mid x \in e_2\}$ bind the variable x (furthermore, x is not allowed to be free in e_2 for this expression to be well-formed).

For each fixed schema σ and an expression $e(x_1, \ldots, x_n)$, the value of $e(x_1, \ldots, x_n)$ is defined by induction on the structure of e and with respect to a database (finite σ-structure) $\mathcal{A}$ and a substitution $[x_1 := a_1, \ldots, x_n := a_n]$ that assigns to each variable x_i a value a_i of the appropriate type. We write $e[x_1 := a_1, \ldots, x_n := a_n](\mathcal{A})$ to denote this value; if the context is understood, we shorten this to $e[x_1 := a_1, \ldots, x_n := a_n]$ or just e. For reason of economy, we use 0 and 1 to code Booleans (that is, $=(e_1,e_2)$ evaluates to 0 if the values of e_1 and e_2 are equal, and to 1 otherwise); the conditional tests for the value of e to be 0. There is the tupling operation $(e_1,\ldots,e_n)$ and projections $\pi_{i,n}$ applied to tuples. The value of $\{e\}$ is the singleton set containing the value of e; $e_1 \cup e_2$ computes the union of two sets, and $\emptyset$ is the empty set.

To define the semantics of $\bigcup$, $\sum$ and $\mathsf{Aggr}_{\mathcal{F}}$, assume that the value of e_2 is the set $\{b_1,\ldots,b_m\}$. Then the value of $\bigcup\{e_1 \mid x \in e_2\}$ is defined to be

$$\bigcup_{i=1}^{m} e_1[x_1{:=}a_1,\ldots,x_n{:=}a_n,x{:=}b_i](\mathcal{A}).$$

The value of $\mathsf{Aggr}_{\mathcal{F}}\{e_1 \mid x \in e_2\}$ is $f_m(\{\!|c_1,\ldots,c_m|\!\})$, where f_m is the mth function in $\mathcal{F} \in \Theta$, and each c_i is the value of $e_1[x_1{:=}a_1,\ldots,x_n{:=}a_n,x{:=}b_i]$, $i=1,\ldots,m$. For the case of summation aggregate $\sum$, the value is $c_1+\ldots+c_m$.

Language $\mathcal{RL}^{\mathrm{aggr}}$ The *flat* language $\mathcal{RL}^{\mathrm{aggr}}(\Omega,\Theta)$ is obtained from $\mathcal{NRL}^{\mathrm{aggr}}(\Omega,\Theta)$ by imposing the following type restrictions:

- each relation in σ is of type $\{rt\}$;
- each expression is of type *ft* (flat type);
- for each rule in (1), all t_is are replaced by Base;
- for each rule in (2), all occurrences of t should be replaced by *rt* (record types).

Thus, input databases for $\mathcal{RL}^{\mathrm{aggr}}$ expressions can be identified with finite σ-structures, when σ is a two-sorted signature. Furthermore, for a $\mathcal{RL}^{\mathrm{aggr}}(\Omega,\Theta)$ expression $e(x_1,\ldots,x_n)$, all free variables are of record types; thus, we shall write $e[x_1{:=}\,\vec{a}_i,\ldots,x_n{:=}\,\vec{a}_n](\mathcal{A})$ for the value of this expression, where $\vec{a}_i$ are tuples of the same type as x_i, and $\mathcal{A}$ is a σ-structure. We shall now assume, for the rest of the paper, that in σ-structures, all relations are finite.

$$\frac{}{0,1:\mathbb{Q}} \qquad \frac{R\in\sigma}{R:\mathrm{tp}_\sigma(R)}$$

$$\frac{e:\mathbb{Q} \quad e_1:t \quad e_2:t}{\textit{if } e \textit{ then } e_1 \textit{ else } e_2 : t}$$

$$\frac{e:\mathbb{Q}\times\ldots\times\mathbb{Q}\ (n \text{ times})}{f(e):\mathbb{Q} \qquad P(e):\mathbb{Q}}$$
for $f:\mathbb{Q}^n\to\mathbb{Q}$ and $P\subseteq\mathbb{Q}^n$ from Ω

$$\frac{e_1:t_1, \quad \ldots \quad e_n:t_n}{(e_1,\ldots,e_n):t_1\times\ldots\times t_n}$$

$$\frac{i\le n \quad e:t_1\times\ldots\times t_n}{\pi_{i,n}\ e:t_i} \qquad \frac{e_1:t \quad e_2:t}{=(e_1,e_2):\mathbb{Q}} \tag{1}$$

$$\frac{e:t}{\{e\}:\{t\}} \qquad \frac{e_1:\{t\} \quad e_2:\{t\}}{e_1\cup e_2:\{t\}} \qquad \frac{}{\emptyset^t:\{t\}}$$

$$\frac{e_1:\{t_1\} \quad e_2:\{t_2\}}{\bigcup\{e_1 \mid x^{t_2}\in e_2\}:\{t_1\}} \qquad \frac{e_1:\mathbb{Q} \quad e_2:\{t\}}{\sum\{e_1 \mid x^t\in e_2\}:\mathbb{Q}} \tag{2}$$

$$\frac{}{x^t:t} \qquad \frac{\mathcal{F}\in\Theta \quad e_1:\mathbb{Q} \quad e_2:\{t\}}{\mathsf{Aggr}_{\mathcal{F}}\{e_1 \mid x^t\in e_2\}:\mathbb{Q}}$$

Figure 1: Expressions of $\mathcal{NRL}^{\mathrm{aggr}}(\Omega,\Theta)$ over signature σ

Properties of $\mathcal{NRL}^{\mathrm{aggr}}(\Omega,\Theta)$ The relational part of the language (without arithmetic and aggregation) is known to have precisely the power of the *nested relational algebra*, the standard query language for nested relations, see [3]. (The language of [3] coded Boolean values as elements of type $\{\mathit{unit}\}$, where *unit* is a type having just one value. Here we code Booleans as 0 and 1, but it does not affect expressiveness, see [25].) The flat fragment of the language, without aggregation, has the power of the relational algebra, that is, first-order logic [34].

When the standard arithmetic and the $\sum$ aggregate are added, the language becomes powerful enough to code standard SQL aggregation features such as the `GROUPBY` and `HAVING` clauses, and aggregate functions such as `TOTAL`, `COUNT`, `AVG`, `MIN`, `MAX`, present in all commercial versions of SQL [32]. This was shown in [24]. The language we deal with here is a lot more powerful, as it puts no limitations on the class of allowed arithmetic operations and aggregate functions.

The following observation will be very useful for establishing expressivity bounds for $\mathcal{NRL}^{\mathrm{aggr}}$. Recall that $\prod$ stands for the product aggregate. We write $\prod\{e_1 \mid x \in e_2\}$ instead of $\mathsf{Aggr}_\Pi\{e_1 \mid x \in e_2\}$.

Lemma 5.1
$\mathcal{NRL}^{\mathrm{aggr}}(\mathsf{All}, \mathsf{All}) \approx \mathcal{NRL}^{\mathrm{aggr}}(\mathsf{All}, \{\sum, \prod\})$. $\square$

For the following result, we let $root(y, x)$ be any function $\mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$ such that, for any $n > 0$, $root(n, x) = sign(x) \cdot \sqrt[n]{|x|}$ if $\sqrt[n]{|x|} \in \mathbb{Q}$.

Proposition 5.2 (see [25]) *Let Ω include $+$, $*$, $-$, $\div$ and $root(y,x)$. Then $\mathcal{NRL}^{\mathrm{aggr}}(\Omega, \{\sum, \prod\})$ is conservative over flat types. That is, any expression of $e : ft$ of $\mathcal{NRL}^{\mathrm{aggr}}(\Omega, \{\sum, \prod\})$, having only free variables and relations of flat types, is definable in $\mathcal{RL}^{\mathrm{aggr}}(\Omega, \{\sum, \prod\})$.* $\square$

5.1 Encoding $\mathcal{RL}^{\mathrm{aggr}}(\Omega, \Theta)$ in $\mathcal{L}_{\mathrm{aggr}}$

Recall that any two-sorted schema σ naturally corresponds to a type of the form $\{rt_1\} \times \ldots \times \{rt_n\}$ where all rt_is are record types. We denote this type by σ, too. Thus, any two-sorted σ-structure can be considered as an object of type σ and we can speak of applying $\mathcal{NRL}^{\mathrm{aggr}}$ queries to it. Furthermore, any tuple $\vec{x}$ of free variables of a $\mathcal{L}_{\mathrm{aggr}}$ formula has a type, say (n, J), which corresponds to some record type rt. In this case we say that $\vec{x}$ has type rt. Our goal now is to show the following.

Theorem 5.3 *For any schema σ, and for any expression $e : \{rt\}$ of $\mathcal{RL}^{\mathrm{aggr}}(\Omega, \Theta)$ over σ without free variables, there exists a formula $\varphi(\vec{x})$ of $\mathcal{L}_{\mathrm{aggr}}(\Omega, \Theta)$, with $\vec{x}$ of type rt, such that for any σ-structure $\mathcal{A}$, $e(\mathcal{A}) = \{\vec{a} \mid \mathcal{A} \models \varphi(\vec{a})\}$.*

Proof sketch. We need a translation of $\mathcal{RL}^{\mathrm{aggr}}$ expressions into formulae and terms of $\mathcal{L}_{\mathrm{aggr}}$ that accounts for free variables of $\mathcal{RL}^{\mathrm{aggr}}$ expressions. We define contexts Γ as sets of variable assignments that relate $\mathcal{RL}^{\mathrm{aggr}}$ variables to those of $\mathcal{L}_{\mathrm{aggr}}$. Then for expression $e(x_1^{rt_1}, \ldots, x_m^{rt_m})$ of type $\{rt\}$ and a formula φ we write $\Gamma \vdash e \stackrel{\vec{z}}{\Longrightarrow} \varphi$ if for every assignment of values to free variables on e and corresponding (by Γ) tuples of free variables of φ other than $\vec{z}$, it is the case the value of e on $\mathcal{A}$ is the same as the set of all tuples $\vec{z}$ that make φ true. We also give an analogous definition of $\Gamma \vdash e \Longrightarrow t_1 * \ldots * t_p$ for the case when e is of type rt (and thus produces a tuple of terms). We then define these relations by induction on the structure $\mathcal{RL}^{\mathrm{aggr}}$ expressions, assuming certain consistency conditions for Γ. The translation is given in Figure 2 in the appendix. Its correctness implies the statement of the theorem. See appendix for more details. $\square$

5.2 Expressiveness of $\mathcal{NRL}^{\mathrm{aggr}}$

Each $\mathcal{NRL}^{\mathrm{aggr}}$ expression $e : t$ over schema σ defines a query (map) Q_e from finite σ structures to objects of type t. Combining Theorem 5.3, Lemma 5.1 and Proposition 5.2, we obtain:

Corollary 5.4 *For every $\mathcal{NRL}^{\mathrm{aggr}}(\mathsf{All}, \mathsf{All})$ expression $e : \{rt\}$ without free variables over a schema with all relations of flat types, the query Q_e defined by e can be expressed in $\mathcal{L}_{\mathrm{aggr}}(\mathsf{All}, \mathsf{All})$.* $\square$

We call a record type *relational* if it is of the form $b \times \ldots \times b$. We call a $\mathcal{NRL}^{\mathrm{aggr}}$ expression without free variables *relational* if it is of type $\{rt\}$ where rt is relational. Finally, a query Q_e defined by a relational expression is called *relational* if all relations in σ are of type $\{b \times \ldots \times b\}$. From Hanf-locality of $\mathcal{L}_{\mathrm{aggr}}(\mathsf{All}, \mathsf{All})$ we conclude:

Corollary 5.5 (Expressiveness of Aggregation) *Every relational query in $\mathcal{NRL}^{\mathrm{aggr}}(\mathsf{All}, \mathsf{All})$ is Hanf-local and Gaifman-local.* $\square$

This implies, for example, that $\mathcal{NRL}^{\mathrm{aggr}}(\mathsf{All}, \mathsf{All})$ cannot express any query listed in Corollary 4.8.

The main result on expressibility bounds – Corollary 5.5 – makes the assumption that the input structure is relational, that is, only contains elements of the base type b. One can relax this in two different ways. First, one can let input structures be nested (that is, of arbitrary type t). Second, one can permit flat structures of types $\{rt\}$ where rt is an arbitrary record type, not just $b \times \ldots \times b$. The natural question, then, is whether one can recover Corollary 5.5 under those relaxations.

The answer to the first question, given below, is relatively simple. The other question will be dealt with in the next section.

Proposition 5.6 *There exist $\mathcal{NRL}^{\text{aggr}}$ graph queries (not using arithmetic and aggregation) on graphs of type $\{\{b\} \times \{b\}\}$ that are neither Hanf-local nor Gaifman-local.* $\square$

6 Restrictions of $\mathcal{L}_{\text{aggr}}$

While $\mathcal{L}_{\text{aggr}}$ subsumed SQL-like languages, and gave us bounds on their expressive power, it is not particularly attractive for use as a direct analog of relational calculus for aggregate extensions, mostly because of its use of infinitary connectives and quantification over $\mathbb{Q}$. We now consider a finitary restriction of $\mathcal{L}_{\text{aggr}}$, and show that it in a certain sense captures the language $\mathcal{RL}^{\text{aggr}}$.

We need a standard definition of the *active domain* of a finite database [1], slightly modified here to deal with two base types. Given a σ-structure $\mathcal{A}$, the set of all elements of $\mathcal{U}$ that occur in $\mathcal{A}$ is denoted by $adom(\mathcal{A})$, and the set of all constants from $\mathbb{Q}$ that occur in $\mathcal{A}$ is denoted by $adom_{\mathbb{Q}}(\mathcal{A})$. Given a record type $rt = b_1 \times \ldots \times b_n$, by $adom_{rt}(\mathcal{A})$ we mean $A_1 \times \ldots \times A_n$ where $A_i = adom(\mathcal{A})$ whenever $b_i = b$ and $A_i = adom_{\mathbb{Q}}(\mathcal{A})$ whenever $b_i = \mathbb{Q}$.

Definition 6.1 *The logic $L_{\text{aggr}}(\Omega, \Theta)$ is defined to be the restriction of $\mathcal{L}_{\text{aggr}}(\Omega, \Theta)$ that does not permit infinitary conjunctions and disjunctions, and $0, 1$ are the only two constant terms of the rational sort. The semantics is modified so that $\mathcal{A} \models \exists x.\varphi(x, \cdots)$ means that $\mathcal{A} \models \varphi(x_0, \cdots)$ for some $x_0 \in adom_*(\mathcal{A})$, where $adom_*$ is adom for first-sort x, and $adom_{\mathbb{Q}}$ for second-sort x. Furthermore, in $\mathsf{Aggr}_{\mathcal{F}}\vec{z}.(\varphi, t)$, $\vec{z}$ ranges over $adom_{rt}(\mathcal{A})$ where rt is the type of $\vec{z}$.* $\square$

In contrast with $\mathcal{L}_{\text{aggr}}$, L_{aggr} formulae can be evaluated on finite two-sorted structures in the usual bottom-up way, assuming effectiveness of all functions and predicates in Ω and aggregates in Θ. To connect this logic with $\mathcal{RL}^{\text{aggr}}$, we need to impose some conditions on the aggregates from Θ.

Definition 6.2 *Let $\mathcal{M} = \langle \mathbb{Q}, \odot, \iota \rangle$ be a commutative monoid on $\mathbb{Q}$. A* monoidal *aggregate given by $\mathcal{M}$ is defined to be $\mathcal{F}_{\mathcal{M}}$ whose nth function is $f_n(\{\!|x_1, \ldots, x_n|\!\}) = x_1 \odot x_2 \odot \ldots \odot x_n$ for $n > 0$ and f_0 returns ι. (f_ω is arbitrary.) An aggregate signature is monoidal if every aggregate in it is.* $\square$

The usual aggregates $\sum$ and $\prod$ are monoidal, given by $\langle \mathbb{Q}, +, 0 \rangle$ and $\langle \mathbb{Q}, *, 1 \rangle$ respectively. In fact, most aggregates in the database setting are either monoidal or can be obtained from monoidal aggregates by means of simple arithmetic operations [10].

We now have to say what it means for a logic to capture a query language. In one direction, it is easy – every query must be definable by a logical formula. For the other direction, one has to deal with the standard database problem of *safety* [1]: while queries always return finite results, arbitrary formulae need not, as they may define infinite subsets of $\mathbb{Q}$. We circumvent this problem by using the following definition of capture.

Definition 6.3 *We say that $L_{\text{aggr}}(\Omega, \Theta)$* captures *$\mathcal{RL}^{\text{aggr}}(\Omega, \Theta)$ if the following two conditions hold for every signature σ. First, for every $\mathcal{RL}^{\text{aggr}}(\Omega, \Theta)$ expression $e : \{rt\}$ without free variables there exists a $L_{\text{aggr}}(\Omega, \Theta)$ formula $\varphi(\vec{x})$ with $\vec{x}$ of type rt such that $e(\mathcal{A}) = \{\vec{a} \mid \mathcal{A} \models \varphi(\vec{a})\}$. Second, for every $L_{\text{aggr}}(\Omega, \Theta)$ formula $\varphi(\vec{x})$ with $\vec{x}$ of type rt there exists a $\mathcal{RL}^{\text{aggr}}(\Omega, \Theta)$ expression $e(x^{rt}) : \mathbb{Q}$ such that the value of $e[x^{rt} := \vec{a}](\mathcal{A})$ is 0 if $\mathcal{A} \models \varphi(\vec{a})$ and 1 otherwise.* $\square$

Theorem 6.4 *Let Θ be monoidal. Then $L_{\text{aggr}}(\mathsf{All}, \Theta)$ captures $\mathcal{RL}^{\text{aggr}}(\mathsf{All}, \Theta)$. Moreover, $L_{\text{aggr}}(\Omega, \{\sum\})$ captures $\mathcal{RL}^{\text{aggr}}(\Omega, \{\sum\})$ if Ω contains $(+, -, *, \div)$, and $L_{\text{aggr}}(\Omega, \{\sum, \prod\})$ captures $\mathcal{RL}^{\text{aggr}}(\Omega, \{\sum, \prod\})$ if Ω contains $(+, -, *, \div, root)$.*

As a corollary, we answer the question about expressivity of $\mathcal{RL}^{\text{aggr}}$ over $\mathbb{Q}$. Since first-order logic with counting quantifiers is no more expressive than $L_{\text{aggr}}(\{+,*,\div,<\},\{\sum\})$, the results of [2] imply

Corollary 6.5 *Assume that the test for connectivity of graphs of type $\{\mathbb{Q}\times\mathbb{Q}\}$ is not definable in $\mathcal{RL}^{\text{aggr}}(\{+,-,*,\div,<\},\{\sum\})$. Then there exists a problem in NLOGSPACE for which there are no constant-depth polynomial-size unbounded fan-in circuits with threshold gates.* $\square$

Whether the class of problems definable with polynomial-size constant-depth threshold circuits (called TC^0) is different from NLOGSPACE (or even NP) remains an open problem in complexity theory. It now follows that we cannot answer questions about expressivity of aggregate query languages over $\mathbb{Q}$ without solving those hard complexity problems. The key difference between this situation and earlier results on expressive power of $\mathcal{NRL}^{\text{aggr}}$ is that the domain $\mathcal{U}$ is *unordered*, whereas over $\mathbb{Q}$ we do have an order. An analog of corollary 6.5 can be proved for inputs of type $\{b\times b\}$ assuming that the domain $\mathcal{U}$ of type b comes equipped with a linear order. Without an order, one retains the bounds of Corollary 5.5.

Remark In the full version, we consider *range-restriction* conditions for L_{aggr} that give us a precise capture of $\mathcal{RL}^{\text{aggr}}$ (that is, output of every formula corresponds to the output of some query.)

7 Conclusions

In this paper we studied the problem of adding aggregate operators to logics. We were primarily motivated by problems arising in database theory. Aggregation is indispensable in majority of real life applications, but the foundations of query languages that support it are not adequately studied. Here, we concentrated on the problem of expressive power. We first considered adding aggregation to logics that already have substantial counting power, and proved the the resulting logics have a very nice behavior: over pure relational structures, they can only define local properties. We then considered a query language, that models all the standard aggregation features of commercial query languages (and, in fact, more, as it permits *every* well-defined aggregate operator and *every* arithmetic function). We showed a simple embedding of this language into aggregate logic, and thus proved that over a large class of inputs, it is also local.

The previous best result on expressiveness of aggregation was [25] that showed a weaker form of locality for a much weaker language; furthermore, the encoding of queries in a logic with counting quantifiers, used in [25] to prove the main result, was very unpleasant. That paper presented two challenges: first, to find a nice encoding in a logic that is natural for dealing with aggregates, and second, to extend results to ordered types of database elements. We answered both challenges here. The logic $\mathcal{L}_{\text{aggr}}$ is indeed natural for dealing with aggregates, and the encoding of the language $\mathcal{RL}^{\text{aggr}}$ into it is simple and intuitive. Furthermore, we showed that questions on expressiveness in the ordered setting cannot be answered without resolving some deep problems in circuit complexity.

We believe that the use of logics like $\mathcal{L}_{\text{aggr}}$ and L_{aggr} is not limited to studying the expressive power of languages. They provide a disciplined approach to design of declarative languages for aggregation, and hopefully this can be used to study other problems, such as language design and optimization of aggregate queries. Known techniques for optimizing aggregate queries are quite ad hoc, and perhaps a clean theoretical framework can help here. We note that [20], starting with essentially the same motivation, designed a categorical calculus for aggregate queries. It will be interesting to see what are the connections between that calculus and $\mathcal{L}_{\text{aggr}}$. Among other possibilities for future work we would like to mention are extensions of the general approach to other datatypes used in applications, complexity and decidability problems for fragments of $\mathcal{L}_{\text{aggr}}$, extensions to logics that have a fixpoint mechanism as well as counting power (in particular, we would like to see if bounds such as those of [4] can be proved in the presence of aggregation).

References

[1] S. Abiteboul, R. Hull and V. Vianu. *Foundations of Databases*, Addison Wesley, 1995.

[2] D.A. Barrington, N. Immerman, H. Straubing. On uniformity within NC^1. *JCSS*, 41:274–306,1990.

[3] P. Buneman, S. Naqvi, V. Tannen, L. Wong. Principles of programming with complex objects and collection types. *TCS*, 149 (1995), 3–48.

[4] J. Cai, M. Fürer and N. Immerman. On optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12 (1992), 389–410.

[5] M. Consens and A. Mendelzon. Low complexity aggregation in GraphLog and Datalog, *TCS* 116 (1993), 95–116.

[6] G. Dong, L. Libkin and L. Wong. Local properties of query languages. *TCS*, to appear. Extended abstract in *ICDT'97*, 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, *JCSS*, 54 (1997), 400–411.

[9] R. Fagin, L. Stockmeyer and M. Vardi, On monadic NP vs monadic co-NP, *Information and Computation*, 120 (1994), 78–92.

[10] L. Fegaras and D. Maier. Towards an effective calculus for object query languages. In *ACM SIGMOD'95*, pages 47–58.

[11] H. Gaifman. On local and non-local properties, *Proceedings of the Herbrand Symposium, Logic Colloquium '81*, North Holland, 1982.

[12] E. Grädel and Y. Gurevich. Metafinite model theory. *Information and Computation* 140 (1998), 26–81.

[13] W. Hanf. Model-theoretic methods in the study of elementary logic. In J.W. Addison et al, eds, *The Theory of Models*, North Holland, 1965, pages 132–145.

[14] L. Hella. Logical hierarchies in PTIME. *Information and Computation*, 129 (1996), 1–19.

[15] L. Hella, L. Libkin and J. Nurmonen. Notions of locality and their logical characterizations over finite models. Manuscript, 1997.

[16] N. Immerman and E. Lander. Describing graphs: A first order approach to graph canonization. In *"Complexity Theory Retrospective"*, Springer Verlag, Berlin, 1990.

[17] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM* 29 (1982), 699–717.

[18] Ph. Kolaitis and J. Väänänen. Generalized quantifiers and pebble games on finite structures. *Annals of Pure and Applied Logic*, 74 (1995), 23–75.

[19] Ph. Kolaitis, M. Vardi. Infinitary logic and 0-1 laws. *Information and Computation*, 98 (1992), 258–294.

[20] K. Lellahi and V. Tannen. A calculus for collections and aggregates. In *CTCS'97*, pages 261–280.

[21] L. Libkin. On the forms of locality over finite models. In *LICS'97*, pages 204–215.

[22] L. Libkin. On counting logics and local properties. In *LICS'98*, pages 501–512.

[23] L. Libkin and L. Wong. Aggregate functions, conservative extensions, and linear orders. In *Proc. Database Programming Languages 1993*, Springer, 1994, pages 282–294.

[24] L. Libkin, L. Wong. Query languages for bags and aggregate functions. *JCSS* 55 (1997), 241–272.

[25] L. Libkin and L. Wong. On the power of aggregation in relational query languages. In *Proc. Database Programming Languages 1997*, Springer LNCS 1369, pages 260–280.

[26] I.S. Mumick, H. Pirahesh, R. Ramakrishnan. Magic of duplicates and aggregates. In *Conf. on Very Large Databases*, 1990, pages 264–277.

[27] I.S. Mumick and O. Shmueli. How expressive is stratified aggregation? *AMAI* 15 (1995), 407–435.

[28] J. Nurmonen. On winning strategies with unary quantifiers. *J. Logic and Computation*, 6 (1996), 779–798.

[29] M. Otto. *Bounded Variable Logics and Counting: A Study in Finite Models.* Springer Verlag, 1997.

[30] K. Ross and Y. Sagiv. Monotonic aggregation in deductive databases. In *PODS'92*, pages 114–126.

[31] K. Ross, D. Srivastava, P. Stuckey and S. Sudarshan. Foundations of aggregation constraints. *TCS* 193 (1998), 149–179.

[32] J. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press 1988.

[33] A. Van Gelder. The well-founded semantics of aggregation. In *PODS'92*, pages 127–138.

[34] L. Wong. Normal forms and conservative properties for query languages over collection types. *JCSS* 52(1):495–505, June 1996.

APPENDIX

Proof of Theorem 4.2

The embedding $\mathcal{L}_{\mathbf{C}} \preccurlyeq_{\mathsf{rk}} \mathcal{L}_{\text{aggr}}(\mathsf{All}, \mathsf{All})$ is trivial. For the other direction, we need the following lemma, which will be used in the encoding of aggregate terms in order to count tuples over both numerical and first-sort domains.

Lemma 7.1 *For every formula $\varphi(\vec{z}, \vec{y})$ of $\mathcal{L}_{\mathbf{C}}$ and every natural number m, there exists a $\mathcal{L}_{\mathbf{C}}$ formula $\nu_m[\varphi](\vec{z})$ such that $\nu_m[\varphi](\vec{a})$ holds iff the number of tuples $\vec{b}$ such that $\varphi(\vec{a}, \vec{b})$ holds is precisely m. Moreover, $\mathsf{rk}(\nu_m[\varphi]) = \mathsf{rk}(\varphi) + k$, where k is the number of first-sort variables in $\vec{y}$.*

Proof of the lemma. To simplify the notation in the proof, we use tuples of rationals $\vec{q} = (q_1, \ldots, q_n)$ instead of the official notation $(c_{q_1}, \ldots, c_{q_n})$ in $\mathcal{L}_{\mathbf{C}}$ formulae. We assume $m > 0$ (if $m = 0$, then $\nu_m[\varphi](\vec{z})$ can be taken to be $\neg\exists\vec{y}.\ \varphi(\vec{z}, \vec{y})$). If all variables in $\vec{y}$ are first-sort, then $\nu_m[\varphi](\vec{z})$ is $\#\vec{y}.\varphi = m$. If all variables in $\vec{y}$ are second-sort, then $\nu_m[\varphi](\vec{z})$ is

$$\bigvee_B \Big(\bigwedge_{\vec{b} \in B} \varphi(\vec{z}, \vec{b}) \ \wedge \ \bigwedge_{\vec{b} \notin B} \neg\varphi(\vec{z}, \vec{b}) \Big),$$

where the disjunction is taken over m-element sets B of tuples of rational numbers, of the same length as $\vec{y}$. Note that the rank of the above formula equals $\mathsf{rk}(\varphi)$.

Now assume that $\vec{y} = (\vec{y}_1, \vec{y}_2)$, where $\vec{y}_1$ are first-sort variables and $\vec{y}_2$ are second-sort variables. Let k be the length of $\vec{y}_1$ and p the length of $\vec{y}_2$, $k, p > 0$. For a given $m > 0$, let S_m be the set of all tuples $(l_1, \ldots, l_s)$ of positive integers such that $\sum_{i=1}^{s} l_i = m$. Then $\nu_m[\varphi](\vec{z})$ is given by

$$\bigvee_{(l_1,\ldots,l_s) \in S_m} \ \bigvee_{\substack{\vec{a}_1,\ldots,\vec{a}_s \in \mathbb{Q}^p \\ \vec{a}_1,\ldots,\vec{a}_s \text{ distinct}}} \left[\left(\forall\vec{y}_1 \forall\vec{y}_2\ \varphi(\vec{z}, \vec{y}_1, \vec{y}_2) \rightarrow \bigvee_{i=1}^{s} (\vec{y}_2 = \vec{a}_i) \right) \ \wedge \ \left(\bigwedge_{i=1}^{s} (\#\vec{y}_1.\varphi(\vec{z}, \vec{y}_1, \vec{a}_i) = l_i) \right) \right]$$

Clearly, $\mathsf{rk}(\nu_m[\varphi]) = \mathsf{rk}(\varphi) + k$. To see correctness, note that if there are m tuples $\vec{b}_i = (\vec{b}_1^i, \vec{b}_2^i)$ satisfying $\varphi(\vec{z}, \vec{b}_i)$ (divided into two subtuples by sort), then the number of distinct tuples among $\vec{b}_2^i$, say s, is at most m, and if for each of the s such tuples $\vec{b}_2^i$, l_i is the number of tuples $\vec{c}$ for which $(\vec{c}, \vec{b}_2^i)$ is among $\vec{b}_j$s, then $\sum_{i=1}^{s} l_i = m$. Since the converse to this is true as well, and the above formula codes this condition, we conclude the proof of the lemma. $\square$

To prove the theorem, we define for each second-sort term $t(\vec{x})$ a formula $\psi_t(\vec{x}, z)$ of $\mathcal{L}_{\mathbf{C}}$, and for each formula $\varphi(\vec{x})$ of $\mathcal{L}_{\text{aggr}}(\mathsf{All}, \mathsf{All})$ a formula $\psi_\varphi(\vec{x})$ of $\mathcal{L}_{\mathbf{C}}$ such that $\mathcal{A} \models \varphi_t(\vec{a}, q)$ iff $t^{\mathcal{A}}(\vec{a}) = q$, and $\mathcal{A} \models \varphi(\vec{a})$ iff $\mathcal{A} \models \psi_\varphi(\vec{a})$, for any $\mathcal{A}$ and $\vec{a}$ a tuple of $\mathcal{A}$ of the same type as $\vec{x}$. This is done by simultaneous induction.

Let $t(\vec{x})$ be a second-sort term. Then $\psi_t(\vec{x}, z)$, where z is a second-sort variable not contained in $\vec{x}$, is defined inductively as follows:

- if $t(\vec{x})$ is a variable x_i, then $\psi_t(\vec{x}, z)$ is the formula $z = x_i$;
- if $t(\vec{x})$ is a constant symbol c_q, then $\psi_t(\vec{x}, z)$ is the formula $z = c_q$;

- if $t(\vec{x})$ is $f(t_1, \ldots, t_n)$, where f is an n-ary function symbol from Ω, then $\psi_t(\vec{x}, z)$ is the formula

$$\bigvee (z = c_q \wedge \bigwedge_i \psi_{t_i}(\vec{x}, c_{q_i})),$$

  where the disjunction is taken over all tuples $(q, q_1, \ldots, q_n) \in \mathbb{Q}^{n+1}$ such that $f(q_1, \ldots, q_n) = q$;

- if $t(\vec{x})$ is $\#\vec{y}.\varphi(\vec{x}, \vec{y})$, then $\psi_t(\vec{x}, z)$ is the formula $z = \#\vec{y}.\psi_\varphi(\vec{x}, \vec{y})$;

- if $t(\vec{x})$ is $\mathsf{Aggr}_{\mathcal{F}}\vec{y}.(\varphi(\vec{x}, \vec{y}), s(\vec{x}, \vec{y}))$, where $\mathcal{F}$ is an aggregate from Θ, $\varphi(\vec{x}, \vec{y})$ is a formula, and $s(\vec{x}, \vec{y})$ is a second-sort term, then $\psi_t(\vec{x}, z)$ is the formula $\psi_1(\vec{x}, z) \vee \psi_2(\vec{x}, z)$. Here $\psi_1(\vec{x}, z)$ is

$$\bigvee_M \left((z = c_q) \wedge \nu_m[\psi_\varphi](\vec{x}) \wedge \bigwedge_{i=1}^{s} \nu_{m_i}[\psi_\varphi(\vec{x}, \vec{y}) \wedge \psi_s(\vec{x}, \vec{y}, c_{q_i})] \right)$$

  where the disjunction is taken over all finite multisets M over $\mathbb{Q}$. We use m to denote the cardinality of M, s for the number of distinct elements of M (say, $q_1, \ldots, q_s$), and m_i for the multiplicity of q_i; furthermore, q stands for $f_m(\{\!|q_1, \ldots, q_m|\!\})$, where f_m is the mth function from $\mathcal{F}$. In all occurrences, ν_m and ν_{m_i} bind $\vec{y}$.

  The formula $\psi_2(\vec{x}, z)$ is defined as

$$(z = c_\omega) \ \wedge \ \bigwedge_{m \in \mathbb{N}} \neg \nu_m[\psi_\varphi](\vec{x})$$

  where c_ω is the constant symbol corresponding to f_ω from $\mathcal{F}$, and ν_m again binds variables $\vec{y}$.

Let then $\varphi(\vec{x})$ be a formula of $\mathcal{L}_{\text{aggr}}(\mathsf{All}, \mathsf{All})$. We may assume without loss of generality that every for occurrence of a σ relation $R_i(t_1, \ldots, t_{n_i})$, each t_j is a variable of the appropriate sort. This is because the only first-sort terms are variables, and for second-sort terms $t_{j_1}, \ldots, t_{j_s}$, we can introduce fresh variables $v_{j_1}, \ldots, v_{j_s}$ of the second sort, and replace $R_i(\vec{t})$ by $\exists v_{j_1}, \ldots, v_{j_s}\ (\bigwedge_l v_{j_l} = t_{j_l} \wedge R_i(\vec{t}'))$ where $\vec{t}'$ is obtained from $\vec{t}$ by replacing each second-sort term t_{j_l} by v_{j_l}. Note that this transformation into an equivalent formula does not change the rank.

We now define the formula $\psi_\varphi(\vec{x})$ of $\mathcal{L}_{\mathbf{C}}$ (simultaneously with formulae φ_t) inductively as follows:

- if $\varphi(\vec{x})$ is a formula $R_i(\vec{x})$, where $R_i(n_i, J_i) \in \sigma$ and $\vec{t} : (n_i, J_i)$ then $\psi_\varphi(\vec{x}) = \varphi(\vec{x})$;

- if $\varphi(\vec{x})$ is $\neg\theta(\vec{x})$, then $\psi_\varphi(\vec{x})$ is $\neg\psi_\theta(\vec{x})$; and if $\varphi(\vec{x})$ is $\bigvee \varphi_i(\vec{x})$ or $\bigwedge \varphi_i(\vec{x})$, then $\psi_\varphi(\vec{x})$ is $\bigvee \psi_{\varphi_i}(\vec{x})$ or $\bigwedge \psi_{\varphi_i}(\vec{x})$, respectively;

- if $\varphi(\vec{x})$ is $\exists y \theta(\vec{x}, y)$ or $\forall y \theta(\vec{x}, y)$, then $\psi_\varphi(\vec{x})$ is $\exists y \psi_\theta(\vec{x}, y)$ or $\forall y \psi_\theta(\vec{x}, y)$, respectively;

- if $\varphi(\vec{x})$ is a formula $t(\vec{x}) = s(\vec{x})$ where t and s are first-sort terms (that is, first-sort variables), then $\psi_\varphi(\vec{x}) = \varphi(\vec{x})$;

- if $\varphi(\vec{x})$ is a formula $t(\vec{x}) = s(\vec{x})$ where t and s are second-sort terms, then $\psi_\varphi(\vec{x})$ is the formula $\exists z(\psi_t(\vec{x}, z) \wedge \psi_s(\vec{x}, z))$;

- if $\varphi(\vec{x})$ is a formula $P(t_1, \ldots, t_n)$ where each t_i is a second-sort term and P is an n-ary predicate symbol from Ω, then $\psi_\varphi(\vec{x})$ is the formula $\exists z_1 \ldots \exists z_n (\bigwedge_i \psi_{t_i}(\vec{x}, z_i) \wedge \bigvee \bigwedge_i z_i = c_{q_i})$ where the disjunction is taken over all tuples $(q_1, \ldots, q_n)$ such that $P(q_1, \ldots, q_n)$ holds.

Obviously ψ_t and ψ_φ constructed above are formulae of $\mathcal{L}_{\mathbf{C}}$. It is also straightforward to verify that $\mathsf{rk}(\psi_t) \le \mathsf{rk}(t)$ and $\mathsf{rk}(\psi_\varphi) \le \mathsf{rk}(\varphi)$, for every second-sort term t and for every formula φ of $\mathcal{L}_{\text{aggr}}(\mathsf{All},\mathsf{All})$.

It remains to verify that the constructed formulas ψ_t and ψ_φ have the desired properties. This is proved in the next lemma.

Lemma 7.2 *If $t(\vec{x})$ is a second-sort term, $\vec{a}$ is a tuple of $\mathcal{A}$ of the same type as $\vec{x}$, and $q \in \mathbb{Q}$, then $t^{\mathcal{A}}(\vec{a}) = q$ iff $\mathcal{A} \models \psi_t(\vec{a}, q)$. Similarly, if $\varphi(\vec{x})$ is a formula of $\mathcal{L}_{\text{aggr}}(\mathsf{All},\mathsf{All})$, and $\vec{a}$ is a tuple of $\mathcal{A}$ of the same type as $\vec{x}$, then $\mathcal{A} \models \varphi(\vec{a})$ iff $\mathcal{A} \models \psi_\varphi(\vec{a})$.*

Proof. Proof is by simultaneous induction. We sketch a few cases; others are straightforward. Let first $t(\vec{x})$ be a second-sort term, $\vec{a}$ a tuple of $\mathcal{A}$ of the same type as $\vec{x}$, and $q \in \mathbb{Q}$. The cases where t is a variable, constant symbol, counting term or function symbol from Ω, are obvious.

Suppose then $\varphi(\vec{x})$ is a formula of $\mathcal{L}_{\text{aggr}}(\mathsf{All},\mathsf{All})$, and $\vec{a}$ is a tuple of $\mathcal{A}$ of the same type as $\vec{x}$. Again, the first three cases above are obvious. If $\varphi(\vec{x})$ is a formula $t(\vec{x}) = s(\vec{x})$ where t and s are second-sort terms, then $\mathcal{A} \models \varphi(\vec{a})$ iff for some $q \in \mathbb{Q}$ $t^{\mathcal{A}}(\vec{a}) = q$ and $s^{\mathcal{A}}(\vec{a}) = q$. Since $t^{\mathcal{A}}(\vec{a}) = q$ and $s^{\mathcal{A}}(\vec{a}) = q$ hold iff $\mathcal{A} \models \psi_t(\vec{a}, q)$ and $\mathcal{A} \models \psi_s(\vec{a}, q)$, this is equivalent to $\mathcal{A} \models \psi_\varphi(\vec{a})$.

If $\varphi(\vec{x})$ is a formula $P(t_1, \ldots, t_n)$ where each t_i is a second-sort term and P is an n-ary predicate symbol from Ω, then $\mathcal{A} \models \varphi(\vec{a})$ iff for some $(q_1, \ldots, q_n) \in \mathbb{Q}^n$ we have $t_i^{\mathcal{A}}(\vec{a}) = q_i$ for each i and $(q_1, \ldots, q_n) \in P$. But this is exactly what ψ_φ states.

Let $t(\vec{x})$ be $\mathsf{Aggr}_{\mathcal{F}}\vec{y}.(\varphi(\vec{x},\vec{y}), s(\vec{x},\vec{y}))$. Now $t^{\mathcal{A}}(\vec{a}) = q$ iff $f_m(s(\varphi(\vec{a},\mathcal{A}))) = q$, where m is the cardinality of the multiset $M = s(\varphi(\vec{a},\mathcal{A}))$, or $q = f_\omega$ for $\varphi(\vec{a},\mathcal{A})$ is infinite. For the case of $\varphi(\vec{a},\mathcal{A})$ finite, $f_m(s(\varphi(\vec{a},\mathcal{A}))) = q$ holds iff $q = f_m(M)$, there exist distinct rationals $q_1, \ldots, q_s$ such that the multiplicity of q_i in M is $m_i > 0$, $\sum_{i=1}^{s} m_i = m$, and for each q_i, there are exactly m_i tuples $\vec{b}$ such that $\varphi(\vec{a},\vec{b})$ holds and $t^{\mathcal{A}}(\vec{a},\vec{b}) = q_i$. But this is exactly what ψ_1 states. If M is infinite, $f_m(s(\varphi(\vec{a},\mathcal{A}))) = q$ iff $q = f_\omega$. Since ψ_2 tests for M being infinite and z being f_ω, we conclude the proof of correctness for the case of aggregate terms. $\square$

Theorem 4.2 now follows immediately. $\square$

Proof of Lemma 4.5

We start with the following simple observation.

Lemma 7.3 *Every formula in $\mathcal{L}_{\mathbf{C}}$ is equivalent to a formula that does not contain any quantifiers $\exists q$ or $\forall q$ over second sort variables q.*

Proof. Clearly $\exists q \varphi(\vec{x}, q)$ is equivalent to $\bigvee_{q\in\mathbb{Q}} \varphi(\vec{x}, c_q)$, and similarly, $\forall q \varphi(\vec{x}, q)$ is equivalent to $\bigwedge_{q\in\mathbb{Q}} \varphi(\vec{x}, c_q)$. Hence the claim follows by straightforward induction. $\square$

We now prove Lemma 4.5. Let $t(\vec{x},\vec{z})$ be a second sort term, and $\varphi(\vec{x},\vec{z})$ a formula of $\mathcal{L}_{\mathbf{C}}$, where $\vec{x}$ are first sort variables and $\vec{z}$ are second sort variables. Furthermore, let $\vec{c}$ and $\vec{q}$ be tuples of elements of A and $\mathbb{Q}$ of matching lengths. We prove by simultaneous induction on $i \le r$ that the following two claims hold whenever the duplicator is playing according to his winning strategy:

(a) If $\mathsf{rk}(t) \le i$ and $\vec{c} \in (\text{dom}(p_{r-i}))^{|\vec{x}|}$, then $t^{\mathcal{A}^*}(\vec{c},\vec{q}) = t^{\mathcal{B}^*}(p_{r-i}\vec{c},\vec{q})$.

(b) If $\mathsf{rk}(\varphi) \le i$ and $\vec{c} \in (\mathrm{dom}(p_{r-i}))^{|\vec{x}|}$, then $\mathcal{A}^* \models \varphi(\vec{c}, \vec{q})$ if and only if $\mathcal{B}^* \models \varphi(p_{r-i}\vec{c}, \vec{q})$.

Assume first that $i = 0$. If $\mathsf{rk}(t) = 0$, then t is either a second sort variable or a constant symbol. In both cases (a) holds trivially. Similarly, if $\mathsf{rk}(\varphi) = 0$, then by Lemma 7.3, we can assume that φ is quantifier free. Since p_r is determined by a winning strategy of the duplicator, it is a partial isomorphism, and hence it preserves the truth of φ.

Assume then that $i > 0$ and the claims (a) and (b) hold for all $j < i$. Let $\mathsf{rk}(t) \le i$. We can assume without loss of generality that $\mathsf{rk}(t) = i$; otherwise (a) follows directly from the induction hypothesis, since $p_{r-i} \subseteq p_{r-j}$ for all $j < i$. Thus, t is of the form $\#\vec{y}.\psi(\vec{x}, \vec{y}, \vec{z})$, where $\mathsf{rk}(\psi) = i - k$ for $k = \mid \vec{y} \mid$. Consider now the following function $g : A^k \to B^k$ generated by the winning strategy of the duplicator:

$$g(d_1, \ldots, d_k) = (f_{r-i+1}(d_1), \ldots, f_{r-i+k}(d_k)).$$

As each f_{r-i+j} is a bijection $A \to B$, it is easy to see that g is a bijection $A^k \to B^k$. Moreover, by the induction hypothesis, $\mathcal{A}^* \models \psi(\vec{c}, \vec{d}, \vec{q})$ if and only if $\mathcal{B}^* \models \psi(p_{r-i+k}\vec{c}, g(\vec{d}), \vec{q})$ for all $\vec{d} = (d_1, \ldots, d_k) \in A^k$. In other words, g is a bijection between the sets $\psi(\vec{c}, \mathcal{A}^*, \vec{q})$ and $\psi(p_{r-i}\vec{c}, \mathcal{B}^*, \vec{q})$, and so $t^{\mathcal{A}^*}(\vec{c}, \vec{q}) = t^{\mathcal{B}^*}(p_{r-i}\vec{c}, \vec{q})$.

To prove (b), assume (without loss of generality) that $\mathsf{rk}(\varphi) = i$. If φ is of the form $t = s$ for second sort terms s and t, the claim follows easily from (a). The cases $\varphi = \neg\psi$, $\varphi = \bigvee \Phi$ and $\varphi = \bigwedge \Phi$ are also straightforward. Assume finally, that $\varphi = \exists y \psi(\vec{x}, y, \vec{z})$. If $\mathcal{A}^* \models \varphi(\vec{c}, \vec{q})$, then $\mathcal{A}^* \models \psi(\vec{c}, d, \vec{q})$ for some $d \in A$. The induction hypothesis implies then that $\mathcal{B}^* \models \psi(p_{r-i}\vec{c}, p_{r-i+1}(d), \vec{q})$, whence $\mathcal{B}^* \models \varphi(p_{r-i}\vec{c}, \vec{q})$. The converse implication is proved in the same way. This completes the induction.

The lemma follows now from the case $i = r$ of claim (b). $\square$

Complete definition of $\mathcal{NRL}^{\mathrm{aggr}}(\Omega, \Theta)$

The set of free variables of an expression e is defined by induction on the structure of e and we often write $e(x_1, \ldots, x_n)$ to explicitly indicate that x_1, ..., x_n are free variables of e. 0, 1, R, and $\emptyset^t$ have no free variables. The free variables of $(e_1, \ldots, e_n)$ are those of e_1, ..., e_n. The free variables of *if* e *then* e_1 *else* e_2 are those of e, e_1, and e_2. The free variables of $f(e)$, $P(e)$, $\pi_{i,n}\ e$ and $\{e\}$ are those of e. The free variables of $=(e_1, e_2)$ and $e_1 \cup e_2$ are those of e_1 and e_2. The free variable of x is the variable x itself. The free variables of $\bigcup\{e_1 \mid x \in e_2\}$, $\sum\{e_1 \mid x \in e_2\}$, and $\mathsf{Aggr}_{\mathcal{F}}\{e_1 \mid x \in e_2\}$ are the free variables of e_1, excluding x, and those of e_2. In these three constructs, we require that x is not a free variable of e_2.

Semantics For each fixed schema σ and an expression $e(x_1, \ldots, x_n)$, the value of $e(x_1, \ldots, x_n)$ is defined by induction on the structure of e and with respect to a database (finite σ-structure) $\mathcal{A}$ and a substitution $[x_1 := a_1, \ldots, x_n := a_n]$ that assigns to each variable x_i a value a_i of the appropriate type. We write $e[x_1 := a_1, \ldots, x_n := a_n](\mathcal{A})$ to denote this value; if the context is understood, we often shorten this to $e[x_1 := a_1, \ldots, x_n := a_n]$ or even just e. The values of 0 and 1 are $0, 1 \in \mathbb{Q}$. For reason of economy, we use them to code Booleans, letting 0 code "true" and 1 code "false" (any other pair of rationals can be used for that purpose). The value of $f(e)$ is the rational number obtained by applying the function $f \in \Omega$ to the value of e. The value of $P(e)$ is 0 if the predicate in Ω denoted by P holds on the tuple denoted by e; otherwise, it is 1. The value of R is the corresponding relation in $\mathcal{A}$. The value of *if* e *then* e_1 *else* e_2 is that of e_1 if the value of e is 0; otherwise, it is that of e_2. The value of $(e_1, \ldots, e_n)$ is the n-ary tuple having the values of e_1, ..., e_n at positions 1, ..., n respectively. The value of $\pi_{i,n}\ e$ is the value at the i-th position of the n-ary tuple denoted by e. The value of $=(e_1, e_2)$ is 0 if e_1 and e_2 have the same value; otherwise, it is 1. The value of the variable x is the corresponding a assigned to x in the given substitution. The value of $\{e\}$ is the singleton set containing the value of e. The value of $e_1 \cup e_2$ is the union of the two sets denoted by e_1 and e_2. The value of $\emptyset$ is the empty set.

To define the semantics of $\bigcup$, $\sum$ and $\mathsf{Aggr}_{\mathcal{F}}$, assume that the value of e_2 is the set $\{b_1, \ldots, b_m\}$. Then the value of $\bigcup\{e_1 \mid x \in e_2\}$ is defined to be

$$\bigcup_{i=1}^{m} e_1[x_1{:=}a_1, \ldots, x_n{:=}a_n, x{:=}b_i](\mathcal{A}).$$

The value of $\sum\{e_1 \mid x \in e_2\}$ is

$$\sum_{i=1}^{m} e_1[x_1{:=}a_1, \ldots, x_n{:=}a_n, x{:=}b_i](\mathcal{A}).$$

Finally, the value of $\mathsf{Aggr}_{\mathcal{F}}\{e_1 \mid x \in e_2\}$ is $f_m(\{\!|c_1, \ldots, c_m|\!\})$, where f_m is the mth function in $\mathcal{F} \in \Theta$, and each c_i is the value of $e_1[x_1{:=}a_1, \ldots, x_n{:=}a_n, x{:=}b_i]$, $i = 1, \ldots, m$.

Proof of Lemma 5.1

The inclusion of $\mathcal{NRL}^{\text{aggr}}(\mathsf{All}, \{\sum, \prod\})$ in $\mathcal{NRL}^{\text{aggr}}(\mathsf{All}, \mathsf{All})$ is trivial. For the inclusion of $\mathcal{NRL}^{\text{aggr}}(\mathsf{All}, \mathsf{All})$ in $\mathcal{NRL}^{\text{aggr}}(\mathsf{All}, \{\sum, \prod\})$, it is only necessary to show that each occurrence of $\mathsf{Aggr}_{\mathcal{F}}\{e_1 \mid x \in e_2\}$ can be expressed in $\mathcal{NRL}^{\text{aggr}}(\mathsf{All}, \{\sum, \prod\})$, for e_1 and e_2 already in $\mathcal{NRL}^{\text{aggr}}(\mathsf{All}, \{\sum, \prod\})$.

Let $\mathsf{encode} : \mathbb{Q} \to \mathbb{Q}$ be an injective function that maps the rational numbers to prime numbers. This function is in $\mathcal{NRL}^{\text{aggr}}(\mathsf{All}, \{\sum, \prod\})$, since All contains all rational number functions. Let $\{\!|e_1 \mid x \in e_2|\!\}$ be the multiset containing exactly the rational numbers that e_1 takes as x ranges over the values of a set e_2. Then $\prod\{\mathsf{encode}(e_1) \mid x \in e_2\}$ is an injection that maps multisets $\{\!|e_1 \mid x \in e_2|\!\}$ to rational numbers. Since All contains all functions on rational numbers, there is a function $\hat{\mathcal{F}}$ in All for each aggregate function $\mathcal{F}$ in All such that for each n, for each multiset B having exactly n rational numbers b_1, ..., b_n, and for each $f_n \in \mathcal{F}$, it is the case that $\hat{\mathcal{F}}(\prod_{i=1}^{n}(\mathsf{encode}(b_i))) = f_n(B)$. Therefore, $\mathsf{Aggr}_{\mathcal{F}}\{e_1 \mid x \in e_2\}$ can be replaced by the expression $\hat{\mathcal{F}}(\prod\{\mathsf{encode}(e_1) \mid x \in e_2\})$ in $\mathcal{NRL}^{\text{aggr}}(\mathsf{All}, \{\sum, \prod\})$. $\square$

Proof of Theorem 5.3

We first need a translation of $\mathcal{RL}^{\text{aggr}}$ expressions into formulae and terms of $\mathcal{L}_{\text{aggr}}$ that accounts for free variables (of record types) of $\mathcal{RL}^{\text{aggr}}$ expressions. Let Γ be a set of variable assignments, that is, pairs $(x^{rt}, \vec{y})$ where $\vec{y}$ has type rt and x is a variable of type rt. We require such a set to be *consistent*, that is, to satisfy the following. First, there are no pairs in Γ with the same first component. Second, no two components of the $\vec{y}$ tuples are the same. By $\Gamma[x^{rt}{:=}\ \vec{y}]$ we mean the result of adding the pair $(x^{rt}, \vec{y})$ to Γ, assuming that it does not violate the above consistency conditions. We write $\textsc{Cons}(\Gamma, \Gamma')$ if $\Gamma \cup \Gamma'$ is consistent[1]. Given $\Gamma = \{(x_i, \vec{y}_i) \mid i = 1, \ldots, m\}$, we define $\Gamma(x_i)$ to be $\vec{y}_i$, $\vec{y}_\Gamma$ to be $(\Gamma(x_1), \ldots, \Gamma(x_m))$, and $(\vec{y}_\Gamma, \vec{z})$ to be $(\Gamma(x_1), \ldots, \Gamma(x_m), \vec{z})$.

[1]Technically speaking, Γ is a list, so the above condition says that any list containing precisely all the elements that occur in Γ and Γ', without duplicates, is consistent.

Let an expression $e(x_1^{rt_1}, \ldots, x_m^{rt_m})$ be of type $\{rt\}$. Let Γ be defined on $x_1, \ldots, x_m$ so that $\Gamma(x_i)$ is $\vec{y}_i$ for $i = 1, \ldots, m$. Let $\varphi(\vec{y}_1, \ldots, \vec{y}_m, \vec{z})$ be a $\mathcal{L}_{\text{aggr}}$ formula where $\vec{z}$ is fresh and of type rt. We adopt the convention that every formula $\varphi(\vec{z})$ can also be considered as a formula $\varphi(\vec{y}, \vec{z})$ with any number of additional free variables, and similarly for terms. We write

$$\Gamma \vdash e \stackrel{\vec{z}}{\Longrightarrow} \varphi$$

to mean that, for any $\vec{a}_1, \ldots, \vec{a}_m$ of types $rt_1, \ldots, rt_m$ respectively, and for any σ-structure $\mathcal{A}$, it holds:

$$(\star) \qquad e[x_1{:=}\vec{a}_1, \ldots, x_m{:=}\vec{a}_m](\mathcal{A}) \quad = \quad \{\vec{a} \mid \mathcal{A} \models \varphi(\vec{a}_1, \ldots, \vec{a}_m, \vec{a})\}$$

Let an expression $e(x_1^{rt_1}, \ldots, x_m^{rt_m})$ be of type $rt = b_1 \times \ldots \times b_p$ with each b_i either b or $\mathbb{Q}$. Let Γ be defined on $x_1, \ldots, x_m$ so that $\Gamma(x_i)$ is $\vec{y}_i$ for $i = 1, \ldots, m$. Let $t_1(\vec{y}_1, \ldots, \vec{y}_m)$, ..., $t_p(\vec{y}_1, \ldots, \vec{y}_m)$ be terms of $\mathcal{L}_{\text{aggr}}$ of sorts agreeing with rt. We write

$$\Gamma \vdash e \Longrightarrow t_1 * \ldots * t_p$$

if, for any $\vec{a}_1, \ldots, \vec{a}_m$ of types $rt_1, \ldots, rt_m$ respectively, and for any σ-structure $\mathcal{A}$, it holds:

$$(\star\star) \qquad (\pi_{i,m}\ e)[x_1{:=}\vec{a}_1, \ldots, x_m{:=}\vec{a}_m](\mathcal{A}) \quad = \quad t_i^{\mathcal{A}}(\vec{a}_1, \ldots, \vec{a}_m), \qquad \text{for all } i$$

In Figure 2, we define $\Gamma \vdash e \Longrightarrow t$ and $\Gamma \vdash e \stackrel{\vec{z}}{\Longrightarrow} \varphi$ inductively on the structure of $\mathcal{RL}^{\text{aggr}}$ queries. The free variables in t and φ are in $\vec{y}_\Gamma$ and $(\vec{y}_\Gamma, \vec{z})$ respectively; we leave them implicit in the rules given in the figure, except in the last two rules. Note that we need the summation aggregate $\sum$ to code conditionals. We also use 0 instead of c_0 in those rules. Theorem 5.3 now follows from

Proposition 7.4 *The translation shown in Figure 2 satisfies conditions $(\star)$ and $(\star\star)$ for every expression e of $\mathcal{RL}^{\text{aggr}}(\Omega, \Theta)$.*

Proof is by straightforward induction on $\mathcal{RL}^{\text{aggr}}$ expressions. □

Proof of Proposition 5.6

Get G be an input graph of type $\{\{b\} \times \{b\}\}$, that is, a set $\{(X_i, Y_i) \mid i = 1, \ldots, n\}$, $X_i, Y_i \subset \mathcal{U}$. We define a query Q that on G produces the graph whose edges are (X, Y) where X, Y are among X_i, Y_i, $i = 1, \ldots, n$, and $X \subseteq Y$. Clearly, this query is definable in $\mathcal{NRL}$, and $\mathcal{NRL}$ can express the $\subseteq$ relation [3]. Assume that this query is Gaifman-local, and let $r > 0$ witness that. Let then $n \geq 4r + 4$, and let $A = \{a_1, \ldots, a_n\}$ be an n-element subset of $\mathcal{U}$. Define G_n as a graph whose nodes are sets of the form $\{a_1, \ldots, a_k\}$, $k = 1, \ldots, n$, and whose edge-relation E_n is

$$\bigcup_{k=1}^{n-1} \{(\{a_1, \ldots, a_k\}, \{a_1, \ldots, a_k, a_{k+1}\})\}$$

Let $X' = \{a_1, \ldots, a_{r+2}\}$ and $X'' = \{a_1, \ldots, a_{3r+3}$. It is straightforward to verify that the r-neighborhoods of (X', X'') and (X'', X') in G_n are isomorphic (as unions of disjoint chains), but (X', X'') belongs to the output of Q on G_n, and $X'', X')$ does not. This shows that Q is not Gaifman-local. By [15, 21] it is not Hanf-local either. □

$$\frac{\Gamma \vdash e \;\Longrightarrow\; t}{\Gamma \cup \Gamma' \vdash e \;\Longrightarrow\; t}\;\textsc{Cons}(\Gamma,\Gamma') \qquad \frac{\Gamma \vdash e \;\xRightarrow{\vec{z}}\; \varphi}{\Gamma \cup \Gamma' \vdash e \;\xRightarrow{\vec{z}}\; \varphi}\;\textsc{Cons}(\Gamma,\Gamma')$$

$$\frac{q = 0,1}{\Gamma \vdash q \;\Longrightarrow\; c_q} \qquad \frac{}{\Gamma \vdash R \;\xRightarrow{\vec{z}}\; R(\vec{z})}\;R \in \sigma$$

$$\frac{\Gamma \vdash e \;\Longrightarrow\; t_1 * \ldots * t_n}{\Gamma \vdash f(e) \;\Longrightarrow\; f(t_1,\ldots,t_n)}\;f \in \Omega \qquad \frac{\Gamma \vdash e \;\Longrightarrow\; t_1 * \ldots * t_n}{\Gamma \vdash P(e) \;\xRightarrow{\emptyset}\; P(t_1,\ldots,t_n)}\;P \in \Omega$$

$$\frac{\Gamma \vdash e_0 \;\Longrightarrow\; t_0 \qquad \Gamma \vdash e_1 \;\xRightarrow{\vec{z}}\; \varphi_1 \qquad \Gamma \vdash e_2 \;\xRightarrow{\vec{z}}\; \varphi_2}{\Gamma \vdash \mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \;\xRightarrow{\vec{z}}\; ((t_0 = 0) \wedge \varphi_1) \vee (\neg(t_0 = 0) \wedge \varphi_2)}$$

$$\frac{\Gamma \vdash e_0 \;\Longrightarrow\; t_0 \qquad \Gamma \vdash e_1 \;\Longrightarrow\; t_1 \qquad \Gamma \vdash e_2 \;\Longrightarrow\; t_2}{\Gamma \vdash \mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \;\Longrightarrow\; (\sum y.((t_0 = 0) \wedge (y = 0), t_1)) + (\sum y.(\neg(t_0 = 0) \wedge (y = 0), t_2))}$$

$$\frac{\Gamma \vdash e_1 \;\Longrightarrow\; t_1 \quad \ldots \quad \Gamma \vdash e_n \;\Longrightarrow\; t_n}{\Gamma \vdash (e_1,\ldots,e_n) \;\Longrightarrow\; t_1 * \ldots * t_n} \qquad \frac{\Gamma \vdash e \;\Longrightarrow\; t_1 * \ldots * t_n}{\Gamma \vdash \pi_{i,n}(e) \;\Longrightarrow\; t_i}$$

$$\frac{\Gamma \vdash e_1 \;\Longrightarrow\; t_1 \qquad \Gamma \vdash e_2 \;\Longrightarrow\; t_2}{\Gamma \vdash\ = (e_1, e_2) \;\Longrightarrow\; \sum w.([(t_1 = t_2) \wedge (w = 0)] \vee [\neg(t_1 = t_2) \wedge (w = 1)], w)}$$

$$\frac{}{\Gamma \vdash x^{rt} \;\Longrightarrow\; y_1 * \ldots * y_n}\;\Gamma(x^{rt}) = (y_1,\ldots,y_n) \qquad \frac{\Gamma \vdash e \;\Longrightarrow\; t_1 * \ldots * t_n}{\Gamma \vdash \{e\} \;\xRightarrow{\vec{z}}\; \bigwedge_{i=1}^{n}(z_i = t_i)}$$

$$\frac{\Gamma \vdash e_1 \;\xRightarrow{\vec{z}}\; \varphi_1 \qquad \Gamma \vdash e_2 \;\xRightarrow{\vec{z}}\; \varphi_2}{\Gamma \vdash e_1 \cup e_2 \;\xRightarrow{\vec{z}}\; \varphi_1 \vee \varphi_2} \qquad \frac{}{\Gamma \vdash \emptyset \;\xRightarrow{\vec{z}}\; \mathit{false}}$$

$$\frac{\Gamma[x{:=}\vec{v}] \vdash e_1 \;\xRightarrow{\vec{z}}\; \varphi(\vec{y}_\Gamma, \vec{v}, \vec{z}) \qquad \Gamma \vdash e_2 \;\xRightarrow{\vec{v}}\; \psi(\vec{y}_\Gamma, \vec{v})}{\Gamma \vdash \bigcup\{e_1 \mid x^{rt} \in e_2\} \;\xRightarrow{\vec{z}}\; \exists \vec{v}.\psi(\vec{y}_\Gamma, \vec{v}) \wedge \varphi(\vec{y}_\Gamma, \vec{v}, \vec{z})}$$

$$\frac{\Gamma[x{:=}\vec{v}] \vdash e_1 \;\Longrightarrow\; t(\vec{y}_\Gamma, \vec{v}) \qquad \Gamma \vdash e_2 \;\xRightarrow{\vec{v}}\; \psi(\vec{y}_\Gamma, \vec{v})}{\Gamma \vdash \mathsf{Aggr}_{\mathcal{F}}\{e_1 \mid x^{rt} \in e_2\} \;\Longrightarrow\; \mathsf{Aggr}_{\mathcal{F}}\vec{v}.(\psi(\vec{y}_\Gamma, \vec{v}), t(\vec{y}_\Gamma, \vec{v}))}\;\mathcal{F} \in \Theta$$

Figure 2: Translation

Proof of Theorem 6.4

The proof of proposition 5.2 in [23] can be adapted to show that for Θ monoidal, every expression e of $\mathcal{RL}^{\text{aggr}}(\mathsf{All}, \Theta)$ is equivalent to an expression e' in which for every subexpression of the form $\bigcup\{e_1 \mid x \in e_2\}$, $\sum\{e_1 \mid x \in e_2\}$, and $\mathsf{Aggr}_{\mathcal{F}}\{e_1 \mid x \in e_2\}$, the expression e_2 is one of $R \in \sigma$. Moreover, if $\sum$ is the only aggregate, the same is true for any expression of $\mathcal{RL}^{\text{aggr}}(\Omega \cup \{+, -, *, \div, \}, \{\sum\})$, and if $\sum$ and $\prod$ are the only aggregates, this is true for any expression of $\mathcal{RL}^{\text{aggr}}(\Omega \cup \{+, -, *, \div, \mathit{root}\}, \{\sum, \prod\})$, for any Ω. Now, to conclude the embedding of $\mathcal{RL}^{\text{aggr}}$ into L_{aggr}, we just follow the rules in Figure 2, and observe that in the resulting formulae all the quantification is over the active domain, and no infinitary connectives are present.

For the other direction, we show how to construct e_φ for every $\varphi(\vec{x})$ so that $e_\varphi[x^{rt} := \vec{a}](\mathcal{A})$ is 0 if $\mathcal{A} \models \varphi(\vec{a})$ and 1 otherwise, and how to construct at the same time, for each term $t(\vec{x})$, an expression $e_t(x^{rt})$ so that $e_t[x^{rt} := \vec{a}](\mathcal{A})$ is the value of term $t(\vec{a})$ in $\mathcal{A}$.

We first observe that for sets of type $\{\mathbb{Q}\}$ that may not contain any element other than 0 or 1, the operation MIN that selects the minimum element is definable in $\mathcal{RL}^{\text{aggr}}$: this is because $\text{MIN}(X) = 1 - (\sum\{1 \mid x \in X\} - \sum\{x \mid x \in X\})$. Also note that for each record type rt, we have a $\mathcal{RL}^{\text{aggr}}$ expression adom_{rt} that defines $\mathit{adom}_{rt}(\mathcal{A})$ for every input $\mathcal{A}$.

We now define e_φ and e_t by induction on the structure of formulae and terms. For c_0, c_1 (constant symbols for 0 and 1) and terms of the form $f(\vec{t})$ and formulae $P(\vec{t})$ with $f, P \in \Omega$, the translation is straightforward. For terms of the first sort (which must be variables), the translation is obtained by applying a projection operator to the tuple of free variables. For φ being $(t_1 = t_2)$, e_φ is $= (e_{t_1}, e_{t_2})$. We now give translations of the remaining constructs of L_{aggr}. In what follows, we use the same symbols for free variables in formulae and $\mathcal{RL}^{\text{aggr}}$ expression for better readability and to avoid repeated projections and tupling operations. For example, we write $\bigcup\{e(y, \vec{x}) \mid \vec{x} \in R\}$ instead of the official $\bigcup\{e(y, \pi_{1,n}\ x, \ldots, \pi_{n,n}\ x) \mid x \in R\}$.

- Assume that φ is $R(t_1, \ldots, t_n)$ where t_is are terms. Then e_φ is $= (0, \text{MIN}(\bigcup\{= (x, (e_{t_1}, \ldots, e_{t_n})) \mid x \in R\}))$.
- For $\psi \equiv \neg\varphi$, $e_\psi = 1 - e_\varphi$. For $\psi \equiv \varphi_1 \vee \varphi_2$, e_ψ is defined to be $\text{MIN}(\{e_{\varphi_1}\} \cup \{e_{\varphi_2}\})$.
- Let $\psi(\vec{y}) \equiv \exists z.\varphi(z, \vec{y})$. Then e_ψ is defined to be

  $$1- \quad = (0, \ \sum\{\mathit{if}\ \ e_\varphi(z, \vec{y})\ \ \mathit{then}\ 1\ \mathit{else}\ 0 \ \mid z \in \mathit{adom}_*\})$$

  where adom_* refers to adom when z is of the first sort, and to $\mathit{adom}_{\mathbb{Q}}$ when z is of the second sort.
- If $t(\vec{y}) \equiv \#\vec{x}.\varphi(\vec{x}, \vec{y})$, and $\vec{x}$ is of type rt (which is in this case $b \times \ldots \times b$), then e_t is $\sum\{\mathit{if}\ \ e_\varphi(\vec{x}, \vec{y})\ \ \mathit{then}\ 1\ \mathit{else}\ 0 \ \mid \vec{x} \in \mathit{adom}_{rt}\}$.
- If $t_0(\vec{x}) \equiv \mathsf{Aggr}_{\mathcal{F}}\vec{z}.(\varphi(\vec{x}, \vec{z}), t(\vec{x}, \vec{z}))$ and $\vec{z}$ is of type rt, then e_{t_0} is

  $$\mathsf{Aggr}_{\mathcal{F}}\{e_t(\vec{x}, \vec{z}) \mid \vec{z} \in \bigcup\{\mathit{if}\ \ e_\varphi(\vec{x}, \vec{v})\ \ \mathit{then}\ \{\vec{v}\}\ \mathit{else}\ \emptyset \ \mid \vec{v} \in \mathit{adom}_{rt}\}\ \}$$

It is straightforward to verify soundness of this translation. This completes the proof. $\square$

Proof of Corollary 6.5

It can be easily seen that the first-order logic with counting quantifiers [2] is no more expressive than $L_{\mathrm{aggr}}(\{+, *, \div, <\}, \emptyset)$; since the former captures the class of problems for which there exist constant-depth polynomial-size unbounded fan-in circuits with threshold gates [2], and connectivity is in NLOGSPACE, the result follows. $\square$