Weaker Forms of Monotonicity for Declarative Networking: A More Fine-Grained Answer to the CALM-Conjecture

DOI: 10.1145/2809784
Handle: http://hdl.handle.net/1942/21061

Weaker Forms of Monotonicity for Declarative Networking: a More Fine-grained Answer to the CALM-conjecture

Tom J. Ameloot* Bas Ketsman† Frank Neven
Daniel Zinn

February 13, 2017

Abstract

The CALM-conjecture, first stated by Hellerstein [31] and proved in its revised form by Ameloot et al. [18] within the framework of relational transducer networks, asserts that a query has a coordination-free execution strategy if and only if the query is monotone. Zinn et al. [42] extended the framework of relational transducer networks to allow for specific data distribution strategies and showed that the non-monotone win-move query is coordination-free for domain-guided data distributions. In this paper, we extend the story by equating increasingly larger classes of coordination-free computations with increasingly weaker forms of monotonicity and present explicit Datalog variants that capture each of these classes. One such fragment is based on stratified Datalog where rules are required to be connected with the exception of the last stratum. In addition, we characterize coordination-freeness as those computations that do not require knowledge about *all* other nodes in the network, and therefore, can not globally coordinate. The results in this paper can be interpreted as a more fine-grained answer to the CALM-conjecture.

1 Introduction

Declarative networking is an approach where distributed computations are modeled and programmed using declarative formalisms based on extensions of Datalog. On a logical level, programs (queries) are specified over a global

*Tom J. Ameloot is a Postdoctoral Fellow of the Research Foundation – Flanders (FWO).

†Bas Ketsman is a PhD Fellow of the Research Foundation – Flanders (FWO).

schema and are computed by multiple computing nodes over which the input database is distributed. These nodes can perform local computations and communicate asynchronously with each other via messages. The model operates under the assumption that messages can never be lost but can be arbitrarily delayed. An inherent source of inefficiency in such systems are the global barriers raised by the need for synchronization in computing the result of queries.

This source of inefficiency inspired Hellerstein [31, 12] to formulate the *CALM-principle* which suggests a link between logical monotonicity on the one hand and distributed consistency without the need for coordination on the other hand.[1] A crucial property of monotone programs is that derived facts must never be retracted when new data arrives. The latter implies a simple coordination-free execution strategy: every node sends all relevant data to every other node in the network and outputs new facts from the moment they can be derived. No coordination is needed and the output of all computing nodes is consistent. This observation motivated Hellerstein [31] to formulate the CALM-conjecture which, in its revised form[2], states

> *"A query has a coordination-free execution strategy iff the query is monotone."*

Ameloot, Neven, and Van den Bussche [18] formalized the conjecture in terms of relational transducer networks[3] and provided a proof. Zinn, Green, and Ludäscher [42] subsequently showed that there is more to this story. In particular, they obtained that when computing nodes are increasingly more knowledgeable on how facts are distributed, increasingly more queries can be computed in a coordination-free manner. Zinn et al. [42] considered two extensions of the transducer network model introduced in [18]. In the first extension, here referred to as the *policy-aware* model, every computing node is aware of the facts that should be assigned to it and can consequently evaluate negation over schema relations. In the second extension, referred to as the *domain-guided* model, data distribution is restricted as follows: each possible domain value d is assigned to at least one node; and, when an input fact contains value d, this fact is given to all nodes that d is assigned to. It was shown in [42] that the coordination-free computations within the original, policy-aware, and domain-guided models form a strict hierarchy and that the non-monotone win-move query can be computed by a coordination-free domain-guided transducer network. *The central objective of this paper is to characterize these increasingly larger classes of coordination-free computations in terms of increasingly weaker forms of monotonicity thereby obtaining a more fine-grained answer to the CALM-conjecture.*

[1] CALM stands for Consistency And Logical Monotonicity.

[2] The original conjecture replaced monotone by Datalog [18].

[3] Relational transducer networks are an extension of relational transducers as first introduced by Abiteboul in [5, 6].

Towards this goal, we introduce the set of *domain-distinct-monotone* and the set of *domain-disjoint-monotone* queries, which we denote by $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$, respectively. Recall that a query is monotone if the output does not decrease (w.r.t. set inclusion) when new facts are added. The classes of domain-distinct-monotone and domain-disjoint-monotone queries then correspond to queries with non-decreasing output (again w.r.t. set inclusion) when only facts are added that contain *at least one* and *only* new domain elements, respectively. While $\mathcal{M}_{distinct}$ is a reformulation of the class of queries preserved under extensions (c.f., Section 3.2), $\mathcal{M}_{disjoint}$ appears to be a new class.[4] We semantically characterize the coordination-free computations within the policy-aware and domain-guided model in terms of $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$, respectively, to obtain the following answer to the CALM-conjecture: for a query Q,

(*i*) Q can be computed by a coordination-free (original) transducer network iff Q is monotone [18];

(*ii*) Q can be computed by a coordination-free policy-aware transducer network iff $Q \in \mathcal{M}_{distinct}$; and,

(*iii*) Q can be computed by a coordination-free domain-guided transducer network iff $Q \in \mathcal{M}_{disjoint}$.

It is tricky to formally define coordination-freeness because ideally it should forbid communication for coordination purposes but it should allow communication to exchange data values, for instance, to compute joins. We employ the formalization of coordination-freeness as introduced in [18], where, intuitively, a query is called coordination-free if for every possible input some ideal distribution exists which allows to find the complete output already without any communication. As a transducer network has to correctly compute the query on all distributions, the latter particularly implies that for non-ideal distributions the only form of communication is for data-transfer, not for coordination. In fact, the distributed evaluation algorithms that we propose in our proofs are naive with respect to communication, in the sense that the whole database is sent to all nodes, and eventually every node computes the result of the query. The exact algorithms, in particular the moment when nodes can start producing output facts, depend on the type of monotonicity and are discussed in Section 4.4.

While we do not claim this notion of coordination-freeness to be the only possible one, the results in this paper imply that it is a sensible one. In particular, we show that coordination-free computations can not globally coordinate across *all* computing nodes. This is made precise by proving that every coordination-free transducer is equivalent to one that has no

[4]But the queries in $\mathcal{M}_{disjoint}$ are conceptually similar to the first order sentences preserved under closed extensions, studied by Compton [24].

knowledge of all other nodes in the network. We refer to Section 4.2 for a more thorough discussion.

In its original formulation [31], the CALM-conjecture did not refer to the general class of monotone queries, but rather to the monotone queries definable in Datalog. Therefore, it is interesting to investigate subclasses of Datalog with negation that remain within $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$, respectively. As mentioned before, $\mathcal{M}_{distinct}$ corresponds to the well-known class of queries which are preserved under extensions, denoted by $\mathcal{E}$. Afrati et al. [7] obtained that semi-positive datalog, denoted SP-Datalog, is included in $\mathcal{E}$, while Cabibbo [23] showed that SP-Datalog extended with value invention captures $\mathcal{E}$ and therefore $\mathcal{M}_{distinct}$. We show that *semi-connected stratified Datalog*, denoted semicon-Datalog$^\neg$, a fragment of stratified Datalog where only 'connected' rules are allowed (except for the last stratum) and which contains SP-Datalog, is included in $\mathcal{M}_{disjoint}$ and that this fragment extended with value invention captures precisely $\mathcal{M}_{disjoint}$. Furthermore, all queries definable in SP-Datalog and semicon-Datalog$^\neg$ are coordination-free within the policy-aware and domain-guided transducer network model, respectively.

The results of this paper are summarized in Figure 2.

Outline. In Section 2, we introduce the necessary definitions. In Section 3, we investigate the classes $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$. In Section 4, we semantically characterize coordination-free transducer networks in the policy-aware and domain-guided models. In Section 5, we consider Datalog fragments for $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$. Because the results are first shown in absence of nullary relations, we discuss in Section 6 how to extend the results to schemas also containing nullary relations. We discuss related work in Section 7 and conclude in Section 8.

2 Definitions

Queries and instances. We assume an infinite set **dom** of data values. A *database schema* σ is a collection of relation names R where every R has arity $ar(R)$. We call $R(\bar{d})$ a *fact* when R is a relation name and $\bar{d}$ is a tuple in **dom**. We say that a fact $R(d_1, \ldots, d_k)$ is *over* a database schema σ if $R \in \sigma$ and $ar(R) = k$. A *(database) instance* I over σ is simply a finite set of facts over σ. We denote by $adom(I)$ the set of all values that occur in facts of I. When $I = \{\mathbf{f}\}$, we simply write $adom(\mathbf{f})$ rather than $adom(\{\mathbf{f}\})$. By $|I|$ we denote the number of facts in I.

A *query* over a schema σ to a schema σ' is a generic mapping Q from instances over σ to instances over σ'. Genericity means that for every permutation π of **dom** and every instance I, $Q(\pi(I)) = \pi(Q(I))$. For a set of facts I and a schema σ, we write $I|_\sigma$ to denote the maximal subset of I that is over σ.

For convenience, we restrict our attention to schemas for which all relations have arity greater than zero. In particular, this means that queries, and therefore also Datalog programs, can not define nullary relations. We address how the addition of nullary relations changes our results in Section 6.

Datalog with negation. We recall Datalog with negation [4], abbreviated Datalog$^\neg$.

Let **var** be the universe of variables, disjoint from **dom**. An *atom* is of the form $R(u_1, \ldots, u_k)$ where R is a relation name and each $u_i \in \mathbf{var}$. We call R the *predicate*. A *literal* is an atom or a negated atom and is called *positive* in the former case and *negative* in the latter case.

A Datalog$^\neg$ *rule* φ is a quadruple $(head_\varphi, pos_\varphi, neg_\varphi, ineq_\varphi)$ where $head_\varphi$ is an atom; pos_φ and neg_φ are sets of atoms; $ineq_\varphi$ is a set of inequalities $(u \neq v)$ with $u, v \in \mathbf{var}$; and, the variables of φ all occur in pos_φ. The components $head_\varphi$, pos_φ and neg_φ are called respectively the *head*, the *positive body atoms* and the *negative body atoms*. We refer to $pos_\varphi \cup neg_\varphi$ as the *body atoms*. Note, neg_φ contains just atoms, not negative literals. Every Datalog$^\neg$ rule φ must have a head, pos_φ must be non-empty and neg_φ may be empty. If $neg_\varphi = \emptyset$ then φ is called *positive*. The set of variables of φ is denoted $vars(\varphi)$.

Of course, a rule φ may be written in the conventional syntax. For instance, if $head_\varphi = T(u, v)$, $pos_\varphi = \{R(u, v)\}$, $neg_\varphi = \{S(v)\}$, and $ineq_\varphi = \{u \neq v\}$, with $u, v \in \mathbf{var}$, then we can write φ as $T(u, v) \leftarrow R(u, v), \neg S(v), u \neq v$.

A rule φ is said to be *over schema* σ if for each atom $R(u_1, \ldots, u_k) \in \{head_\varphi\} \cup pos_\varphi \cup neg_\varphi$, the arity of R in σ is k. A Datalog$^\neg$ *program* P *over* σ is a finite set of Datalog$^\neg$ rules over σ. We write $sch(P)$ to denote the (minimal) database schema that P is over. We define $idb(P) \subseteq sch(P)$ to be the database schema consisting of all relations in rule-heads of P. We abbreviate $edb(P) = sch(P) \setminus idb(P)$. As usual, the abbreviation "idb" stands for "intensional database schema" and "edb" stands for "extensional database schema" [4].

A *valuation for* a rule φ in P w.r.t. an instance I over $edb(P)$, is a total function $V : vars(\varphi) \to \mathbf{dom}$. The *application* of V to an atom $R(u_1, \ldots, u_k)$ of φ, denoted $V(R(u_1, \ldots, u_k))$, results in the fact $R(a_1, \ldots, a_k)$ where $a_i = V(u_i)$ for each $i \in \{1, \ldots, k\}$. This is naturally extended to a set of atoms, which results in a set of facts. The valuation V is said to be *satisfying for* φ *on* I if $V(pos_\varphi) \subseteq I$, $V(neg_\varphi) \cap I = \emptyset$, and $V(u) \neq V(v)$ for each $(u \neq v) \in ineq_\varphi$. If so, φ is said to *derive* the fact $V(head_\varphi)$.

Positive and semi-positive Datalog. A Datalog$^\neg$ program P is *positive* if all rules of P are positive. We say that P is *semi-positive* if for each rule $\varphi \in P$, the atoms of neg_φ are over $edb(P)$. We now give the semantics of a semi-positive Datalog$^\neg$ program P [4]. First, let T_P be the *immediate consequence operator* that maps each instance J over $sch(P)$ to the instance $J' = J \cup A$

where A is the set of facts derived by all possible satisfying valuations for the rules of P on J. Let I be an instance over $edb(P)$. Consider the infinite sequence I_0, I_1, I_2, etc, inductively defined as follows: $I_0 = I$ and $I_i = T_P(I_{i-1})$ for each $i \geq 1$. The *output of P on input I*, denoted $P(I)$, is defined as $\bigcup_j I_j$; this is the *minimal fixpoint* of the T_P operator.

We denote by Datalog, Datalog($\neq$), and SP-Datalog the class of positive Datalog$^\neg$ programs without inequalities, the positive Datalog$^\neg$ programs, and the class of semi-positive Datalog$^\neg$ programs, respectively. Note that the last two classes may use inequalities.

Stratified semantics. We say that P is *syntactically stratifiable* if there is a function $\rho : sch(P) \to \{1, \ldots, |idb(P)|\}$ such that for each rule $\varphi \in P$, having some head predicate T, the following conditions are satisfied: (1) $\rho(R) \leq \rho(T)$ for each $R(\bar{u}) \in pos_\varphi \cap idb(P)$; and, (2) $\rho(R) < \rho(T)$ for each $R(\bar{u}) \in neg_\varphi \cap idb(P)$. For $R \in idb(P)$, we call $\rho(R)$ the *stratum number* of R. Intuitively, ρ partitions P into a sequence of semi-positive Datalog$^\neg$ programs $P_1, \ldots, P_k$ with $k \leq |idb(P)|$ such that for each $i = 1, \ldots, k$, the program P_i contains the rules of P whose head predicate has stratum number i. This sequence is called a *syntactic stratification* of P. We can now apply the *stratified semantics* to P: for an input I over $sch(P)$, we first compute the fixpoint $P_1(I)$, then the fixpoint $P_2(P_1(I))$, etc. The *output of P on input I*, denoted $P(I)$, is defined as $P_k(P_{k-1}(\ldots P_1(I)\ldots))$. It is well known that the output of P does not depend on the chosen syntactic stratification (if more than one exists). Not all Datalog$^\neg$ programs are syntactically stratifiable. By *stratified Datalog* we refer to all Datalog$^\neg$ programs which are syntactically stratifiable.

Since we only consider syntactically stratifiable programs in this paper, we denote from now on the class of stratified Datalog$^\neg$ programs simply by Datalog$^\neg$.

Computing Queries. For a query Q with input schema σ and output schema σ', and a stratifiable Datalog$^\neg$ program P, we say that P *computes* Q if $Q(I) = P(I)|_{\sigma'}$ for all instances I over σ.

We assume that for each Datalog$^\neg$ program some *idb*-relations are marked as the intended output. In our example Datalog$^\neg$ programs, we use the convention that relation 'O' denotes that output. The input relations are recognizable as the *edb*-relations.

In the sequel, we overload notation and denote both the fragment of Datalog$^\neg$ programs as well as the queries expressed by programs in that class with the same notation. For instance, we use SP-Datalog to denote both the class of semi-positive Datalog$^\neg$ programs as well as the queries which are expressible by a semi-positive Datalog$^\neg$ program.

In examples, we use a unary *idb*-relation *Adom* that contains the active domain of the input. This predicate is computed as the union of the projections of all positions of all *edb*-relations. We omit the rules to compute

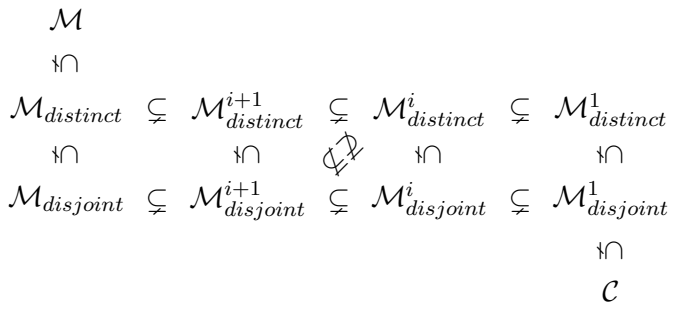

Figure 1: Monotonicity hierarchy

Adom.

3 Weaker forms of monotonicity

We introduce in Section 3.1 two weaker forms of monotonicity that will be used in Section 4 to characterize classes of coordination-free transducers. In Section 3.2, we relate these notions with the well-known classes of queries preserved under homomorphisms and extensions.

3.1 Domain distinct & disjoint monotonicity

We say that a fact $\mathbf{f}$ is *domain distinct* from instance I when $adom(\mathbf{f}) \setminus adom(I) \neq \emptyset$ (i.e., $\mathbf{f}$ should contain a new domain element not occurring in I); $\mathbf{f}$ is *domain disjoint* when $adom(\mathbf{f}) \cap adom(I) = \emptyset$. Furthermore, an instance J is domain distinct (resp., domain disjoint) from I, when every fact $\mathbf{f} \in J$ is domain distinct (resp., domain disjoint) from I. Note that J being domain-disjoint from I does not imply that J_1 is domain-disjoint from J_2 for non-empty disjoint subsets J_1 and J_2 of J.

We introduce two weaker forms of monotonicity by restricting the set of instances for which the monotonicity condition should hold:

Definition 1. Let Q be a query. Then,

- Q is *monotone* if $Q(I) \subseteq Q(I \cup J)$ for all database instances I and J;
- Q is *domain-distinct-monotone* if $Q(I) \subseteq Q(I \cup J)$ for all instances I and J for which J is domain distinct from I; and,
- Q is *domain-disjoint-monotone* if $Q(I) \subseteq Q(I \cup J)$ for all instances I and J for which J is domain disjoint from I.

We denote the class of monotone, domain-distinct-monotone, and domain-disjoint–monotone queries by $\mathcal{M}$, $\mathcal{M}_{distinct}$, and $\mathcal{M}_{disjoint}$, respectively.

Next, we restrict the monotonicity definitions to sets J of bounded size. The size restriction will allow more insight to be developed into the notions of monotonicity. One important intuition for example, is that monotonicity can be preserved by limiting the amount of new data: otherwise, the new data can sometimes grow or contain some large structure that could retract output facts when detected by the query. Formally, for $i \geq 1$, we say that Q is i-monotone, i-domain-distinct-monotone, and i-domain-disjoint-monotone when in the corresponding unrestricted definition J is now restricted to a size of at most i. We denote the respective classes by $\mathcal{M}^i$, $\mathcal{M}^i_{distinct}$, and $\mathcal{M}^i_{disjoint}$. By definition, $\mathcal{M} \subseteq \mathcal{M}_{distinct} \subseteq \mathcal{M}_{disjoint}$ and $\mathcal{M}^i \subseteq \mathcal{M}^i_{distinct} \subseteq \mathcal{M}^i_{disjoint}$.

As explained in more detail in Section 7, the classes $\mathcal{M}^1_{distinct}$ and $\mathcal{M}^1_{disjoint}$ are already introduced in [41] (albeit under a different name). The following theorem provides some insight in how the above classes are related. Separating examples can all be expressed in fragments of Datalog$^\neg$. A more visual representation is provided in Figure 1. We denote the class of all computable queries by $\mathcal{C}$.

Theorem 3.1. *For every $i, j \geq 1$ with $i < j$,*

1. $\mathcal{M} \subsetneq \mathcal{M}_{distinct} \subsetneq \mathcal{M}_{disjoint} \subsetneq \mathcal{C}$;
2. $\mathcal{M} = \mathcal{M}^i$;
3. $\mathcal{M}_{distinct} \subsetneq \mathcal{M}^{i+1}_{distinct} \subsetneq \mathcal{M}^i_{distinct}$;
4. $\mathcal{M}_{disjoint} \subsetneq \mathcal{M}^{i+1}_{disjoint} \subsetneq \mathcal{M}^i_{disjoint}$;
5. $\mathcal{M}^i_{distinct} \subsetneq \mathcal{M}^i_{disjoint}$;
6. $\mathcal{M}^j_{disjoint} \not\subseteq \mathcal{M}^i_{distinct}$; *and,*
7. $\mathcal{M}^i_{distinct} \not\subseteq \mathcal{M}^j_{disjoint}$.

Proof. We focus on the inequalities and sketch the separating examples. All queries are over directed graphs that are defined over the binary edge relation E.

For (1), first, $\mathcal{M} \subsetneq \mathcal{M}_{distinct}$ follows from SP-Datalog $\subseteq \mathcal{M}_{distinct}$. Indeed, take an SP-Datalog program P, a rule $\varphi \in P$, and a valuation V of φ on an input I. The facts $V(neg_\varphi)$ use only values from $adom(I)$. So, when adding facts that are domain-distinct from I, we can not invalidate the negation of φ. Second, we show $Q_{\overline{TC}} \in \mathcal{M}_{disjoint} \setminus \mathcal{M}_{distinct}$, where $Q_{\overline{TC}}$ is the query that computes the complement of the transitive closure of the edge relation. Towards $Q_{\overline{TC}} \in \mathcal{M}_{disjoint}$, it suffices to notice that for all instances I with a path missing from vertex a to vertex b, i.e., $(a, b) \in Q_{\overline{TC}}$, the addition of domain-disjoint subgraphs will not create this path. To argue that

$Q_{\overline{TC}} \notin \mathcal{M}_{distinct}$, it suffices to remark that the addition of domain-distinct subgraphs can create a path $E(a,c), E(c,b)$, where c is a new vertex. Third, for $\mathcal{M}_{disjoint} \subsetneq \mathcal{C}$, take the query that outputs all triangles on condition that no two disjoint triangles exist.

For (2), by definition, $\mathcal{M}^{i+1} \subseteq \mathcal{M}^i$. We show that $\mathcal{M}^i \subseteq \mathcal{M}^{i+1}$. Let $Q \in \mathcal{M}^i$ and let I and J be arbitrary instances such that $|J| \leq i+1$. Pick an arbitrary fact $\mathbf{f} \in J$ and set $J' = J \setminus \{\mathbf{f}\}$. By assumption, $Q(I) \subseteq Q(I \cup J')$ and $Q(I \cup J') \subseteq Q((I \cup J') \cup \{\mathbf{f}\})$. Hence, $Q(I) \subseteq Q(I \cup J)$.

For (3), it suffices to show that $\mathcal{M}^i_{distinct} \setminus \mathcal{M}^{i+1}_{distinct} \neq \emptyset$. Ignoring the direction of edges, let Q^k_{clique} be the query that outputs the edge relation when no clique of k vertices exists and the empty relation otherwise. We show $Q^{i+2}_{\text{clique}} \in \mathcal{M}^i_{distinct} \setminus \mathcal{M}^{i+1}_{distinct}$. Let I be an input not containing $(i+2)$-size cliques, upon which the output is thus nonempty. To expose the non-monotone behavior of Q^{i+2}_{clique}, we can try to extend any $(i+1)$-size cliques in I to $(i+2)$-size cliques by adding a domain-distinct instance J. For this to work, J needs to contain a star: one new value is the center and it points at old clique vertices of I, requiring $|J| \geq i+1$. Such instances J are not considered in the definition of $\mathcal{M}^i_{distinct}$, but they are considered in the definition of $\mathcal{M}^{i+1}_{distinct}$.

For (4), again, we only show that $\mathcal{M}^i_{disjoint} \setminus \mathcal{M}^{i+1}_{disjoint} \neq \emptyset$. Let Q^k_{star} be the query that outputs the edge relation when there is no star with k spokes in the input and the empty relation otherwise. Clearly, $Q^{i+1}_{\text{star}} \notin \mathcal{M}^{i+1}_{disjoint}$ as $i+1$ domain-disjoint edges suffice to create an entirely new star with $i+1$ spokes. On the other hand, if there is not already a star with $i+1$ spokes in the input, we can never create one by adding i domain-disjoint edges.

For (5), using similar reasoning as for (3) above, we can argue that $Q^{i+1}_{\text{clique}} \notin \mathcal{M}^i_{distinct}$ and $Q^{i+1}_{\text{clique}} \in \mathcal{M}^i_{disjoint}$.

For (6), we show $Q^{j+1}_{\text{star}} \in \mathcal{M}^j_{disjoint} \setminus \mathcal{M}^i_{distinct}$, where Q^{j+1}_{star} is the query that outputs the edge relation when there is no star with $j+1$ spokes in the input and the empty relation otherwise. To see $Q^{j+1}_{\text{star}} \in \mathcal{M}^j_{disjoint}$, if we can only add j domain-disjoint edges, we can not extend a star with less than $j+1$ spokes to a star with $j+1$ spokes, and we can also not create a completely new star with $j+1$ spokes. To see $Q^{j+1}_{\text{star}} \notin \mathcal{M}^i_{distinct}$, when the input already contains a star with j spokes, we can increase the number of spokes to $j+1$ by adding one additional edge containing the old central vertex and one new value.

Finally, for (7), we can only prove the stated result for a schema that grows with j. Therefore, let $R_1, \ldots, R_j$ be binary relations and define $Q^j_{\text{duplicate}}$ as the query that gives as output the relation R_1 when the (global) intersection of all relations is empty, and the emptyset otherwise. We first argue $Q^j_{\text{duplicate}} \in \mathcal{M}^i_{distinct}$. Let I be an arbitrary instance where the intersection of all relations is empty. Instances J that are domain-distinct with respect to I can not replicate any existing tuples of I over all relations.

Moreover, if $|J| \leq i$ with $i < j$ then J can not even replicate a completely new tuple over all relations. To see $Q^j_{\text{duplicate}} \notin \mathcal{M}^j_{disjoint}$, a domain-disjoint instance J with $|J| = j$ can replicate a new tuple over all relations. □

The following lemma shows that the proof of Theorem 3.1 item (7) can not be strengthened to a fixed schema.

Proposition 3.2. *Let σ be a fixed non-empty database schema. There are $i \in \mathbb{N}$ and $j \in \mathbb{N}$ with $i < j$, such that $Q \in \mathcal{M}^i_{distinct}$ implies $Q \in \mathcal{M}^j_{disjoint}$ for all queries Q over σ.*

Proof. Let α denote the maximum arity of the relations in σ. As we want to obtain a contradiction, we choose i and j large enough. As we see below, choosing $i = \alpha^\alpha|\sigma|$ and $j = i + 1$ is sufficient for this purpose. Let $Q \in \mathcal{M}^i_{distinct}$ be a query over σ. We show for all instances I and J where J is domain-disjoint from I and $|J| \leq j$ that $Q(I) \subseteq Q(I \cup J)$.

The proof is by contradiction. Let us assume there are instances I, J, where J is domain-disjoint from I and $|J| \leq j$, such that $Q(I) \not\subseteq Q(I \cup J)$. Clearly, $|J| \leq i$ immediately implies $Q(I) \subseteq Q(I \cup J)$ (by i-domain-distinct-monotonicity of Q). Therefore, we focus on $|J| = j$.

Clearly, $|J| \leq |adom(J)|^\alpha|\sigma|$ and $|adom(\mathbf{f})| \leq \alpha$. To finish the proof, we use an auxiliary statement (†):

$$Q(I) \not\subseteq Q(I \cup J) \text{ implies } \forall \mathbf{f}, \mathbf{f}' \in J : adom(\mathbf{f}) = adom(\mathbf{f}'). \quad (\dagger)$$

Now, (†) implies $|adom(J)| \leq \alpha$. Therefore, $|J| \leq \alpha^\alpha|\sigma| = i < j$; which is a contradiction. Hence, $Q(I) \subseteq Q(I \cup J)$.

In the remainder of the proof we show (†) by its contraposition. Assume there are two facts $\mathbf{f}, \mathbf{f}' \in J$ for which $adom(\mathbf{f}) \neq adom(\mathbf{f}')$; we show that $Q(I) \subseteq Q(I \cup J)$. Either $adom(\mathbf{f}) \setminus adom(\mathbf{f}') \neq \emptyset$ or $adom(\mathbf{f}') \setminus adom(\mathbf{f}) \neq \emptyset$. W.l.o.g. below we assume $adom(\mathbf{f}') \setminus adom(\mathbf{f}) \neq \emptyset$.

We define K as the set of facts from J that contain only constants from $adom(\mathbf{f})$, i.e.

$$K = \{\mathbf{g} \in J | adom(\mathbf{g}) \subseteq adom(\mathbf{f})\}.$$

Note that $|K| \leq i < |J|$ because $\mathbf{f}' \notin K$. Now, by i-domain-distinct-monotonicity of Q, we have $Q(I) \subseteq Q(I \cup K)$. Further, $J \setminus K$ is domain-distinct from $I \cup K$ and $|J \setminus K| \leq i < |J|$ because $\mathbf{f} \in K$. So, again by i-domain-distinct-monotonicity of Q, we have $Q(I \cup K) \subseteq Q(I \cup K \cup (J \setminus K)) = Q(I \cup J)$. Hence, $Q(I) \subseteq Q(I \cup J)$. This completes the proof. □

Finally, as is to be expected, deciding whether a query class belongs to one of the monotonicity classes quickly turns undecidable:

Proposition 3.3. *Let $\mathcal{A}$ be a class among $\mathcal{M}$, $\mathcal{M}^i_{distinct}$, and $\mathcal{M}^i_{disjoint}$ for $i \in \mathbb{N}^+ \cup \{\infty\}$; and let P be a Datalog$^\neg$ program with two strata. It is undecidable whether $P \in \mathcal{A}$.*

Proof. The proof is by reduction from testing containment of datalog programs, which is shown to be undecidable in [39].

To this end, let P_1 and P_2 be two datalog programs over the same schema σ and with output predicates O_1 and O_2, respectively, of the same arity. Let U be a unary relation symbol not occurring in σ. Consider the program P

$$
\begin{array}{lcl}
P_1 & & \\
P_2 & & \\
T(\bar{x}) & \leftarrow & O_1(\bar{x}), \neg O_2(\bar{x}) \\
\mathit{Switch}(z) & \leftarrow & U(x), \mathit{Adom}(z) \\
O(\bar{x}) & \leftarrow & T(\bar{x}), \neg \mathit{Switch}(z), \mathit{Adom}(z)
\end{array}
$$

which computes $P_1(I) \setminus P_2(I)$ for every instance I where U is empty, and returns the empty relation otherwise. Now, if $P_1(J) \subseteq P_2(J)$ for all J, then the output of P is always empty and consequently P is in $\mathcal{A}$. When there is an instance J such that $P_1(J) \not\subseteq P_2(J)$ then define I as the extension of J with U interpreted as the empty relation. Then, $P(I) \neq \emptyset$ but $P(I \cup \{U(a)\}) = \emptyset$ for a novel domain element a, and $P \notin \mathcal{A}$. The result follows. □

3.2 Correspondence with other classes

We relate the above classes to those defined in terms of preservation of properties. Let I and J be two instances over σ. A *homomorphism* from I to J is a mapping h from $\mathit{adom}(I)$ to $\mathit{adom}(J)$ such that $\{R(h(\bar{d})) \mid R(\bar{d}) \in I\} \subseteq J$. We say that h is *injective* if $h(d) \neq h(d')$ whenever $d \neq d'$. An instance J is called an *induced subinstance* of I if $J = \{\mathbf{f} \in I \mid \mathit{adom}(\mathbf{f}) \subseteq \mathit{adom}(J)\}$.

Definition 2. Let Q be a query. Then,

- Q is *preserved under (injective) homomorphisms* if for all instances I and J, and for every (injective) homomorphism $h : \mathit{adom}(I) \to \mathit{adom}(J)$, we have $\{R(h(\bar{d})) \mid R(\bar{d}) \in Q(I)\} \subseteq Q(J)$.
- Q is *preserved under extensions* if for all instances I and J for which J is an induced subinstance of I, we have $Q(J) \subseteq Q(I)$.

We denote by $\mathcal{H}$, $\mathcal{H}_{\text{inj}}$, and $\mathcal{E}$ the class of queries preserved under homomorphisms, injective homomorphisms, and extensions, respectively. Sometimes $\mathcal{H}$ is also referred to as the class of *strongly monotonic queries* (e.g., [7, 35]).

Proposition 3.4. $\mathcal{H} \subsetneq \mathcal{H}_{inj} = \mathcal{M} \subsetneq \mathcal{E} = \mathcal{M}_{distinct}$.

Proof. We only argue that $\mathcal{E} = \mathcal{M}_{distinct}$ because the rest is folklore (see, e.g., [7, 35, 38]). The equality follows immediately as J is an induced subinstance of I iff $I \setminus J$ is domain distinct from J. □

4 Coordination-freeness

A *relational transducer* is essentially a collection of queries that transforms a sequence of input facts to a sequence of output facts while maintaining a relational state [6, 27, 28]. In the distributed context, the functionality at each node of a network can be described via a relational transducer, giving rise to so-called relational transducer networks [18]. Subsequently, relational transducer networks have been extended in two ways to give nodes restricted access to *distribution policies* [42]. Such policies model deterministic input data distributions based on hashing. In this section, we review these two extensions, here referred to as *policy-aware* and *domain-guided* transducer networks, and characterize the corresponding classes of coordination-free queries. In particular, we show the following: the queries that are *coordination-free* in policy-aware transducer networks and domain-guided transducer networks correspond to the query classes $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$, respectively. It was shown in [18] that coordination-free queries in the original transducer networks correspond precisely to the set $\mathcal{M}$ of all monotone queries. The results in this section therefore provide a refinement of the CALM-conjecture in terms of weaker forms of monotonicity.

We formalize transducer networks in Section 4.1, and the notion of coordination-freeness in Section 4.2. We present the results in Section 4.3. We provide some additional discussion in Section 4.4.

4.1 Transducer models

We review the two extensions of Zinn et al. [42] to the original transducer network model [18]. Here, we refer to these extensions as *policy-aware* and *domain-guided* transducer networks, respectively.

4.1.1 Networks, data distribution, and policies

A *network* $\mathcal{N}$ is a nonempty finite set of values from **dom**, which we call *nodes*. Let σ be a database schema. A *distributed database instance* H over σ and $\mathcal{N}$ is a function that maps each $x \in \mathcal{N}$ to an instance over σ. A distributed database instance H over σ models a distribution with potential replication of data over the schema σ.

For a set X, let $\mathcal{P}^+(X) = \mathcal{P}(X) \setminus \{\emptyset\}$ denote the set of all non-empty subsets of X. We write $facts(\sigma)$ to denote the set of all possible facts over σ, i.e., using all possible values from **dom**. A *distribution policy* $\boldsymbol{P}$ *for* σ *and* $\mathcal{N}$ is a total function from $facts(\sigma)$ to $\mathcal{P}^+(\mathcal{N})$. Intuitively, $\boldsymbol{P}$ says how to distribute any instance I over σ to the nodes of $\mathcal{N}$, possibly with replication. Concretely, we define $dist_{\boldsymbol{P}}(I)$ to be the distributed database instance H over σ and $\mathcal{N}$ that satisfies $H(x) = \{\mathbf{f} \in I \mid x \in \boldsymbol{P}(\mathbf{f})\}$ for each $x \in \mathcal{N}$.

A *domain assignment* $\boldsymbol{\alpha}$ *for* $\mathcal{N}$ is a total function from **dom** to $\mathcal{P}^+(\mathcal{N})$. A distribution policy $\boldsymbol{P}$ for σ and $\mathcal{N}$ is called *domain-guided* if there exists a domain assignment $\boldsymbol{\alpha}$ for $\mathcal{N}$ such that, for each $R(a_1, \ldots, a_k) \in \mathit{facts}(\sigma)$, we have $\boldsymbol{P}(R(a_1, \ldots, a_k)) = \bigcup_{i=1}^{k} \boldsymbol{\alpha}(a_i)$.[5] In this case, we also say that $\boldsymbol{P}$ is *induced* by domain-assignment $\boldsymbol{\alpha}$. Intuitively, for each value $a \in$ **dom**, function $\boldsymbol{\alpha}$ says which nodes get input facts containing a.

Example 4.1. *Suppose* **dom** *is the set* $\mathbb{N}$ *of natural numbers. Let* $\mathcal{N} = \{1, 2\}$ *be a two-node network and let the schema* σ *contain the single relation symbol* $\mathtt{E}$ *of arity* 2*. Consider the following distribution policy* $\boldsymbol{P}_1$ *for* σ *and* $\mathcal{N}$*:*

$$\boldsymbol{P}_1(\,\mathtt{E}(a, b)\,) = \begin{cases} \{1\} & \text{if } a \text{ is odd} \\ \{2\} & \text{otherwise} \end{cases}$$

for each $a, b \in \mathbb{N}$*. Note that* $\boldsymbol{P}_1$ *partitions any input* I *over* σ *based on its first attribute. If* $I = \{\mathtt{E}(1,3), \mathtt{E}(3,4), \mathtt{E}(4,6)\}$*, the distributed database instance* $\mathit{dist}_{\boldsymbol{P}_1}(I)$ *is*

$$\{1 \mapsto \{\mathtt{E}(1,3), \mathtt{E}(3,4)\}, 2 \mapsto \{\mathtt{E}(4,6)\}\}.$$

This input I *demonstrates that* $\boldsymbol{P}_1$ *is not a domain-guided policy: neither node is assigned all facts containing domain value* 4*.*

A domain-guided policy assigns domain values (rather than facts) to nodes in the network. Consider the domain assignment $\boldsymbol{\alpha}$ *for* $\mathcal{N}$ *that maps odd numbers to* $\{1\}$ *and even numbers to* $\{2\}$*. The corresponding domain-guided distribution policy* $\boldsymbol{P}_2$ *for* σ *and* $\mathcal{N}$ *assigns a fact* $\mathbf{f}$ *to node* 1 *if* $\mathit{adom}(\mathbf{f})$ *contains an odd value, and to node* 2 *if* $\mathit{adom}(\mathbf{f})$ *contains an even value. For the specific input* I *from above, the instance* $\mathit{dist}_{\boldsymbol{P}_2}(I)$ *is*

$$\{1 \mapsto \{\mathtt{E}(1,3), \mathtt{E}(3,4)\}, 2 \mapsto \{\mathtt{E}(3,4), \mathtt{E}(4,6)\}\}.$$

Note that node identifiers may occur in input facts. □

4.1.2 Policy-aware relational transducers

In the following, we write $R^{(k)}$ to denote a relation symbol R of arity k. A *(policy-aware) transducer schema* Υ is a tuple $(\Upsilon_{\text{in}}, \Upsilon_{\text{out}}, \Upsilon_{\text{msg}}, \Upsilon_{\text{mem}}, \Upsilon_{\text{sys}})$ of database schemas with disjoint relation names, with the additional restriction that

$$\Upsilon_{\text{sys}} = \{\mathtt{Id}^{(1)}, \mathtt{All}^{(1)}, \mathtt{MyAdom}^{(1)}\} \cup \{\mathtt{policy}_R^{(k)} \mid R^{(k)} \in \Upsilon_{\text{in}}\}.$$

These schemas are called respectively "input", "output", "message", "memory", and "system".

A *(policy-aware relational) transducer* Π over Υ is a quadruple $(Q_{\text{out}}, Q_{\text{ins}}, Q_{\text{del}}, Q_{\text{snd}})$ of queries having the input schema $\Upsilon_{\text{in}} \cup \Upsilon_{\text{out}} \cup \Upsilon_{\text{msg}} \cup \Upsilon_{\text{mem}} \cup \Upsilon_{\text{sys}}$ and such that

[5] Recall that we have assumed $k \geq 1$ for now; but see also Section 6.

- query Q_{out} has target schema Υ_{out};
- queries Q_{ins} and Q_{del} both have target schema Υ_{mem};
- query Q_{snd} has target schema Υ_{msg}.

These queries form the mechanism by which a node on a network produces output, updates its memory (through insertions and deletions), and sends messages.

We note that the transducers in [18] do not have the relations `MyAdom` and $\texttt{policy}_R$ (with R in Υ_{in}).

4.1.3 Policy-aware transducer networks

A *(policy-aware) transducer network* $\boldsymbol{\Pi}$ is a quadruple $(\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P})$ where $\mathcal{N}$ is a network, Υ is a transducer schema, Π is a transducer over Υ, and $\boldsymbol{P}$ is a distribution policy for Υ_{in} and $\mathcal{N}$. We say that $\boldsymbol{\Pi}$ is *domain-guided* if the distribution policy $\boldsymbol{P}$ is domain-guided. So, domain-guided transducer networks are a special kind of policy-aware transducer networks. We will next give the semantics of $\boldsymbol{\Pi}$ on an input I over Υ_{in}. We start with the underlying intuition.

Intuition We put a copy of transducer Π on each node of $\mathcal{N}$, and policy $\boldsymbol{P}$ initializes each node with a fragment of I. Then we choose an arbitrary node $x \in \mathcal{N}$ and make it "active": we execute the queries of transducer Π to update the output and memory at x, and to generate new messages for the other nodes. Such an active moment of a node is called a *transition* of $\boldsymbol{\Pi}$. The semantics of $\boldsymbol{\Pi}$ is described by so-called *runs* which are infinite sequences of transitions.

Now, the queries of Π may concretely read the following facts at x: local input facts, output and memory facts, any received message facts (over Υ_{msg}), and also some facts over Υ_{sys}. In more detail, the facts over Υ_{sys} consist of the following:

- relation `Id` provides the identifier of x (i.e., just value 'x');
- relation `All` provides the identifiers of all nodes in the network;
- relation `MyAdom` provides for convenience the local active domain of x (based on local facts and received messages); and,
- the relations $\texttt{policy}_R$ provide the facts assigned to x by policy $\boldsymbol{P}$, but restricted to the local active domain of x.[6]

[6]The relations $\texttt{policy}_R$ were previously called '$\texttt{local}_R$' [42]. Here, we have chosen a new predicate name to avoid confusion with local input facts at a node.

Intuitively, by considering only $\texttt{policy}_R$-facts over the local active domain of x, we provide "safe" access to the distribution policy, i.e., we prevent x from using values outside $adom(I) \cup \mathcal{N}$.[7]

Example 4.2. *Recall* **dom**, $\mathcal{N}$, σ, *and* $\boldsymbol{P}_1$ *from Example 4.1. Consider a policy-aware transducer network* $\boldsymbol{\Pi} = (\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P}_1)$ *with* $\Upsilon_{\text{in}} = \sigma$. *We leave* Υ_{out}, Υ_{msg}, *and* Υ_{mem} *unspecified. Let us focus on the node* 1. *On input* $I = \{\texttt{E}(1,3), \texttt{E}(3,4), \texttt{E}(4,6)\}$, *at least the following facts will be exposed to node* 1 *during each transition: the local input facts* $\texttt{E}(1,3)$ *and* $\texttt{E}(3,4)$; *the system facts* $\texttt{Id}(1)$, $\texttt{All}(1)$, $\texttt{All}(2)$, $\texttt{MyAdom}(a)$ *for each* $a \in \{1,2,3,4\}$, *and* $\texttt{policy}_{\texttt{E}}(a,b)$ *with* $a \in \{1,3\}$ *and* $b \in \{1,2,3,4\}$. *If node* 1 *would later receive (and store) the value* 6, *then also* $\texttt{MyAdom}(6)$ *will be exposed, and the* $\texttt{policy}_{\texttt{E}}(a,b)$*-facts with* $a \in \{1,3\}$ *and* $b = 6$. *Also, note that node* 1 *can in principle deduce that* $\texttt{E}(3,2)$ *is not part of* I *since* $\texttt{policy}_{\texttt{E}}(3,2)$ *is present at node* 1 *but not* $\texttt{E}(3,2)$. □

Formal semantics Let $\boldsymbol{\Pi} = (\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P})$ be a policy-aware transducer network. A *configuration* of $\boldsymbol{\Pi}$ is a pair $\rho = (s, b)$ of functions s and b such that:

- s maps each $x \in \mathcal{N}$ to a set of facts over $\Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}}$;
- b maps each $x \in \mathcal{N}$ to a multiset of facts over Υ_{msg}.

We call s and b respectively the *state* and *(message) buffer*. Intuitively, s specifies for each node what output and memory facts it has locally available, and b specifies for each node what messages have been sent to it but that are not yet delivered. The reason for having multisets for the buffers, is that the same message can be sent multiple times to the same recipient and thus multiple copies can be floating around in the network simultaneously. The *start configuration* of $\boldsymbol{\Pi}$ is the unique configuration $\rho = (s, b)$ that satisfies $s(x) = \emptyset$ and $b(x) = \emptyset$ for each $x \in \mathcal{N}$.

Denote $\Pi = (Q_{\text{out}}, Q_{\text{ins}}, Q_{\text{del}}, Q_{\text{snd}})$. A *transition of* $\boldsymbol{\Pi}$ *on an input* I *over* Υ_{in} is a quadruple (ρ_1, x, m, ρ_2) where $\rho_1 = (s_1, b_1)$ and $\rho_2 = (s_2, b_2)$

[7] This safety restriction also makes our model more realistic: in general, a node still needs to communicate with other nodes before it can draw global conclusions about the input or network.

are configurations of $\mathbf{\Pi}$, $x \in \mathcal{N}$, m is a submultiset of $b_1(x)$, and, letting

$$
\begin{aligned}
H &= \mathit{dist}_{\boldsymbol{P}}(I),\\
M &= m \text{ collapsed to a set},\\
J &= H(x) \cup s_1(x) \cup M,\\
A &= \mathit{adom}(J) \cup \mathcal{N},\\
S &= \{\mathtt{Id}(x)\} \cup \{\mathtt{All}(y) \mid y \in \mathcal{N}\} \cup \{\mathtt{MyAdom}(a) \mid a \in A\} \cup\\
&\qquad \{\mathtt{policy}_R(a_1,\ldots,a_k) \mid R^{(k)} \in \Upsilon_{\text{in}},\ \{a_1,\ldots,a_k\} \subseteq A,\\
&\qquad x \in \boldsymbol{P}(R(a_1,\ldots,a_k))\},\\
D &= J \cup S,
\end{aligned}
$$

for the state s_2, we have,

$$
\begin{aligned}
s_2(x)|_{\Upsilon_{\text{out}}} &= s_1(x)|_{\Upsilon_{\text{out}}} \cup Q_{\text{out}}(D),\\
s_2(x)|_{\Upsilon_{\text{mem}}} &= \big[s_1(x)|_{\Upsilon_{\text{mem}}} \cup (Q_{\text{ins}}(D) \setminus Q_{\text{del}}(D))\big]\\
&\qquad \setminus (Q_{\text{del}}(D) \setminus Q_{\text{ins}}(D)),\\
s_2(y) &= s_1(y) \text{ for each } y \in \mathcal{N} \setminus \{x\},
\end{aligned}
$$

and for the buffer b_2, we have (using multiset difference and union),

$$
\begin{aligned}
b_2(x) &= b_1(x) \setminus m,\\
b_2(y) &= b_1(y) \cup Q_{\text{snd}}(D) \text{ for each } y \in \mathcal{N} \setminus \{x\},
\end{aligned}
$$

where we use multiset difference and union. We call ρ_1 and ρ_2 respectively the *source* and *target* configuration of the transition and we refer to x as the active node. If $m = \emptyset$, then we call the transition a *heartbeat*.

For an input I over Υ_{in}, a *run* $\mathcal{R}$ *of* $\mathbf{\Pi}$ *on* I is an infinite sequence of transitions of $\mathbf{\Pi}$ on I, such that the start configuration of $\mathbf{\Pi}$ is used as the source configuration of the first transition, and the target configuration of each transition is the source configuration of the next transition. Note that runs represent nondeterminism: each transition can choose what node becomes active and what messages to deliver from the buffer of that node. We consider only *fair* runs. These are the runs that satisfy the following additional conditions: *(i)* each node is the active node in an infinite number of transitions; and, *(ii)* if a fact occurs infinitely often in the message buffer of a node then this fact is infinitely often delivered to that node. Intuitively, the last condition demands that no sent messages are infinitely delayed.

4.1.4 Computing queries

We are interested in transducers that produce the same facts over Υ_{out} regardless of the network, the distribution policy, and the order of transitions. These transducers are said to compute a query.

Formally, let Q be a query with input schema σ_1 and output schema σ_2. Further, let $\mathbf{\Pi} = (\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P})$ be a policy-aware transducer network. We define the output of a run $\mathcal{R}$ of $\mathbf{\Pi}$, denoted $out(\mathcal{R})$, to be the union of all output facts jointly produced by the nodes of $\mathcal{N}$ during $\mathcal{R}$, i.e., all facts over Υ_{out}. Note that once a fact is added to Υ_{out}, it can never be retracted. We say that $\mathbf{\Pi}$ *computes* Q if *(i)* $\Upsilon_{\text{in}} = \sigma_1$ and $\Upsilon_{\text{out}} = \sigma_2$; and, *(ii)* for each input I over σ_1, every (fair) run $\mathcal{R}$ of $\mathbf{\Pi}$ on I satisfies $out(\mathcal{R}) = Q(I)$.

Now, letting Π be a policy-aware transducer over transducer schema Υ, we say that

- Π *(distributedly) computes* Q *(for all policies)* if for all networks $\mathcal{N}$ and all distribution policies $\boldsymbol{P}$ for Υ_{in} and $\mathcal{N}$, the policy-aware transducer network $(\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P})$ computes Q;
- Π *(distributedly) computes* Q *under domain-guidance* if for all networks $\mathcal{N}$ and all domain-guided distribution policies $\boldsymbol{P}$ for Υ_{in} and $\mathcal{N}$, the transducer network $(\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P})$ computes Q.

4.2 Defining Coordination-freeness

We define coordination-freeness for policy-aware transducers similarly as for the original transducer model [18]. Let Π be a policy-aware transducer over a schema Υ.

- We say that Π is *coordination-free* if (1) Π distributedly computes a query Q, and (2) for all networks $\mathcal{N}$, for all inputs I for Q, there is a distribution policy $\boldsymbol{P}$ for Υ_{in} and $\mathcal{N}$ such that the policy-aware transducer network $(\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P})$ has a run on input I in which $Q(I)$ is already computed in a prefix consisting of only heartbeat transitions.[8]

  Let $\mathcal{F}_1$ denote the set of queries distributedly computed by coordination-free policy-aware transducers.

- Similarly, we say that Π is *coordination-free under domain-guidance* if (1) Π distributedly computes a query Q under domain-guidance, and (2) if for all networks $\mathcal{N}$, for all inputs I for Q, there is a domain-guided policy $\boldsymbol{P}$ for $\mathcal{N}$, such that the transducer network $(\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P})$ has a run on input I in which $Q(I)$ is already computed in a prefix consisting of only heartbeat transitions.

  Let $\mathcal{F}_2$ denote the set of queries distributedly computed by policy-aware transducers that are coordination-free under domain-guidance.

[8] Technically, heartbeat transitions allow to send messages but not to read them. So, 'only heartbeat' transitions effectively means 'no communication'.

4.2.1 Discussion

It is useful to reflect on what it means to be coordination-free. Of course, one could prohibit any form of communication but that would be too drastic and unworkable when data is distributed and communication is already needed for the simple purpose of exchanging data (for instance, to compute joins). Therefore, the present formalization tries to separate the 'data'-communication (that can never be eliminated in a distributed setting) from the 'coordination'-communication by requiring that there is some 'ideal' distribution on which the query can be computed without any communication.[9]

The intuition is as follows: because on the ideal distribution there is no coordination needed (as even communication is not needed there), and the transducer network has to correctly compute the query on *all* distributions, communication is only used to transfer data on non-ideal distributions and is not used to coordinate.

While we do not claim our notion of coordination-freeness to be the only possible one, the results in Section 4.4 confirm that the just described intuition is not too far off. Indeed, it follows that coordination-freeness corresponds precisely to those computations that do not require the knowledge about *all* other nodes in the network, and hence, can not globally coordinate. Specifically, we show that when a node has no complete overview of *all* the nodes in the network, which can be achieved by removing the relation `All`, then every transducer is coordination-free. More importantly, the converse holds true as well. That is, we show that every coordination-free transducer is equivalent to one that does not use the relation `All`.

4.3 Characterization

We characterize the classes $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$ by coordination-free transducers:

Theorem 4.3. $\mathcal{M}_{distinct} = \mathcal{F}_1$.

Theorem 4.4. $\mathcal{M}_{disjoint} = \mathcal{F}_2$.

The rest of Section 4.3 is devoted to the proofs of the just mentioned theorems. In particular, the proof of Theorem 4.3 is divided into Propositions 4.5 and 4.6. Similarly, the proof of Theorem 4.4 is divided into Propositions 4.7 and 4.8. We provide a discussion of these results in Section 4.4.

Proposition 4.5. $\mathcal{F}_1 \subseteq \mathcal{M}_{distinct}$.

[9] We remark that in this ideal distribution it is not always sufficient to give the full input to all nodes (see, e.g., [18]). Furthermore, the network is not necessarily aware that the data is ideally distributed as it can not communicate.

Proof. Let Q be a query distributedly computed by a coordination-free transducer Π. Let Υ denote the schema of Π. Let I and J be two instances over the input schema of Q, such that J is domain-distinct from I. Let $\mathbf{f} \in Q(I)$. We show $\mathbf{f} \in Q(I \cup J)$.

By assumption, Π computes Q on a network $\mathcal{N}$ with at least two nodes. By coordination-freeness, there is a distribution policy $\boldsymbol{P}_1$ for Υ_{in} and $\mathcal{N}$ such that the transducer network $\boldsymbol{\Pi}_1 = (\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P}_1)$ when given input I, has a run $\mathcal{R}_1$ in which $Q(I)$ is already computed in a prefix consisting of only heartbeat transitions. Let $x \in \mathcal{N}$ be a node that outputs $\mathbf{f}$ in this prefix. Intuitively, we now show that x can be made to output $\mathbf{f}$ on input $I \cup J$, so that $\mathbf{f} \in Q(I \cup J)$.

Fix an arbitrary node $y \in \mathcal{N} \setminus \{x\}$. Consider the following distribution policy $\boldsymbol{P}_2$ for Υ_{in} and $\mathcal{N}$: $\boldsymbol{P}_2(\mathbf{g}) = \{y\}$ for all $\mathbf{g} \in J$, and $\boldsymbol{P}_2(\mathbf{g}) = \boldsymbol{P}_1(\mathbf{g})$ for all $\mathbf{g} \in \mathit{facts}(\Upsilon_{\text{in}}) \setminus J$. Denote $\boldsymbol{\Pi}_2 = (\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P}_2)$. Now, $\boldsymbol{\Pi}_2$ on input $I \cup J$ gives to x the same local input facts as $\boldsymbol{\Pi}_1$ on input I:

- Let $\mathbf{g} \in I$ be a fact given to x by $\boldsymbol{\Pi}_1$ on input I. This implies $x \in \boldsymbol{P}_1(\mathbf{g})$. We have $\mathbf{g} \notin J$ because J is domain-distinct from I. Then $\boldsymbol{P}_2(\mathbf{g}) = \boldsymbol{P}_1(\mathbf{g})$ by definition of $\boldsymbol{P}_2$. So, $x \in \boldsymbol{P}_2(\mathbf{g})$, and thus $\mathbf{g}$ is given to x by $\boldsymbol{\Pi}_2$ on input $I \cup J$.

- Let $\mathbf{g} \in I \cup J$ be a fact given to x by $\boldsymbol{\Pi}_2$ on input $I \cup J$. This implies $x \in \boldsymbol{P}_2(\mathbf{g})$. But we have designed $\boldsymbol{P}_2$ to give facts of J only to y, and not to x. Hence, $\mathbf{g} \in I$. So, $\boldsymbol{P}_2(\mathbf{g}) = \boldsymbol{P}_1(\mathbf{g})$ by definition of $\boldsymbol{P}_2$. Thus $x \in \boldsymbol{P}_1(\mathbf{g})$, and $\mathbf{g}$ is given to x by $\boldsymbol{\Pi}_1$ on input I.

Now, when running $\boldsymbol{\Pi}_2$ on input $I \cup J$, if we initially do only heartbeats with active node x, the node x goes through the same state changes as in the heartbeat-prefix of $\mathcal{R}_1$: the local input at x is the same as in $\mathcal{R}_1$, and the locally known active domain does not change since no messages are witnessed during the heartbeats. This causes x to repeatedly observe the same facts in $\texttt{policy}_R$ and `MyAdom`, which are also the same as in $\mathcal{R}_1$.[10] Relations `Id` and `All` also contain the same values at x as in $\mathcal{R}_1$. So, after a while, say after k heartbeats, node x outputs $\mathbf{f}$ in $\boldsymbol{\Pi}_2$. This finite heartbeat prefix can be extended to a full fair run $\mathcal{R}_2$ of $\boldsymbol{\Pi}_2$ on input $I \cup J$, for which $\mathit{out}(\mathcal{R}_2) = Q(I \cup J)$ holds by assumption on Π. Hence, $\mathbf{f} \in Q(I \cup J)$. □

Proposition 4.6. $\mathcal{M}_{\mathit{distinct}} \subseteq \mathcal{F}_1$.

Proof. Let $Q \in \mathcal{M}_{\mathit{distinct}}$. Let σ_1 and σ_2 denote respectively the input and output schema of Q. We construct a coordination-free transducer Π that computes Q.[11]

[10] The relations $\texttt{policy}_R$ appear the same to x during the heartbeat runs of $\boldsymbol{\Pi}_1$ and $\boldsymbol{\Pi}_2$ because $\boldsymbol{P}_2(\mathbf{g}) = \boldsymbol{P}_1(\mathbf{g})$ for all $\mathbf{g} \in \mathit{facts}(\Upsilon_{\text{in}})$ with $\mathit{adom}(\mathbf{g}) \subseteq \mathit{adom}(I)$; this is because J is domain-distinct from I.

[11] We use auxiliary nullary relations, but this is allowed because we impose no restrictions on the languages used to implement transducer queries.

We start with the intuition.

Intuition Let I be an input for Q that is distributed over a network $\mathcal{N}$. We say that a subset $C \subseteq \mathit{adom}(I) \cup \mathcal{N}$ is *complete* at a node $x \in \mathcal{N}$, when x knows for every fact $\mathbf{f}$ over σ_1 with $\mathit{adom}(\mathbf{f}) \subseteq C$ whether $\mathbf{f} \in I$ or $\mathbf{f} \notin I$. If C is indeed complete at x, node x will output $Q(I')$ where $I' = \{\mathbf{f} \in I \mid \mathit{adom}(\mathbf{f}) \subseteq C\}$. Note that $Q(I') \subseteq Q(I)$ by domain-distinct-monotonicity of Q.[12] We construct a policy-aware transducer that executes Q at a node x whenever the unary system relation `MyAdom` is complete at x.

To start, the nodes broadcast their locally given input facts. Each node x stores every received input fact. This way, the system relation `MyAdom` grows at x. During each transition of x, the following happens:

- Node x checks for each input relation $R^{(k)}$ and each k-tuple $(a_1, \ldots, a_k)$ over $\texttt{MyAdom}^k$ whether the fact $\texttt{policy}_R(a_1, \ldots, a_k)$ is shown to x. If so, then x is reponsible for the fact $R(a_1, \ldots, a_k)$ under the distribution policy. In that case, if $R(a_1, \ldots, a_k)$ is absent from the local input at x, node x can conclude that $R(a_1, \ldots, a_k)$ is actually globally absent from the entire input. Then x broadcasts the absence of this fact. These absences are accumulated at all nodes.

- Node x checks whether `MyAdom` is complete at x. Relation `MyAdom` is complete if for each fact $\mathbf{f}$ over σ_1 with $\mathit{adom}(\mathbf{f}) \subseteq \texttt{MyAdom}$, either *(i)* $\mathbf{f}$ is available in the accumulated input facts at x, giving $\mathbf{f} \in I$; or *(ii)* x has locally concluded or received the explicit absence of $\mathbf{f}$ from I. Whenever `MyAdom` is complete, node x computes Q on the locally accumulated input facts.

With the above strategy, no wrong outputs are produced. To show that at least $Q(I)$ is produced, we note that at some point, each node x has received all available input facts and all absences of facts over $\mathit{adom}(I) \cup \mathcal{N}$. At that moment, x computes Q on I, causing at least $Q(I)$ to be output in each run.

Transducer Π is indeed coordination-free according to the formal definition: for all networks $\mathcal{N}$, and for all inputs I, the full output will be computed at some node x with only heartbeats when x is made responsible (under the distribution policy) for all facts over σ_1 made with the values $\mathit{adom}(I) \cup \mathcal{N}$; then x will immediately detect that relation `MyAdom` is complete.

[12] Indeed, since $\mathit{adom}(I') \subseteq C$ and each fact $\mathbf{f} \in I \setminus I'$ has a value outside C, we know that $I \setminus I'$ is domain-distinct from I'. So, $Q(I') \subseteq Q(I' \cup (I \setminus I')) = Q(I)$.

Construction We define the transducer schema Υ of Π as follows:

$$\begin{aligned}\Upsilon_{\text{in}} &= \sigma_1,\\ \Upsilon_{\text{out}} &= \sigma_2,\\ \Upsilon_{\text{msg}} &= \{\texttt{R\_msg}^{(k)}, \texttt{R\_notMsg}^{(k)} \mid R^{(k)} \in \sigma_1\},\\ \Upsilon_{\text{mem}} &= \{\texttt{R\_mem}^{(k)}, \texttt{R\_notMem}^{(k)} \mid R^{(k)} \in \sigma_1\},\end{aligned}$$

and the contents of Υ_{sys} is uniquely determined by the contents of Υ_{in} using the definition of the transducer schema.

Now we specify the transducer $\Pi = (Q_{\text{out}}, Q_{\text{ins}}, Q_{\text{del}}, Q_{\text{snd}})$. For easier readability, each query is specified by a set of rules that are executed in the order they are written. We will go step-by-step through the protocol sketched above, and we specify what rules should be added to Q_{out}, Q_{ins}, and Q_{snd}. First, we define Q_{del} to always return $\emptyset$, causing nodes to only accumulate facts.

To broadcast the input, we add the following rule to Q_{snd} for each $R^{(k)} \in \sigma_1$:

$$\texttt{R\_msg}(\texttt{u}_1, \ldots, \texttt{u}_k) \leftarrow \texttt{R}(\texttt{u}_1, \ldots, \texttt{u}_k).$$

We store any received input facts in memory by adding the following rule to Q_{ins} for each $R^{(k)} \in \sigma_1$:

$$\texttt{R\_mem}(\texttt{u}_1, \ldots, \texttt{u}_k) \leftarrow \texttt{R\_msg}(\texttt{u}_1, \ldots, \texttt{u}_k).$$

At each node, this mechanism eventually collects all input domain values in `MyAdom`.

To broadcast the absence of input facts, we add the following rule to Q_{ins} for each $R^{(k)} \in \sigma_1$:

$$\texttt{R\_notMsg}(\texttt{u}_1, \ldots, \texttt{u}_k) \leftarrow \texttt{policy}_\texttt{R}(\texttt{u}_1, \ldots, \texttt{u}_k), \neg\texttt{R}(\texttt{u}_1, \ldots, \texttt{u}_k).$$

Recall that $\texttt{policy}_R$ uses only values from `MyAdom`. We store any received absence-facts in memory by adding the following rule to Q_{ins} for each $R^{(k)} \in \sigma_1$:

$$\texttt{R\_notMem}(\texttt{u}_1, \ldots, \texttt{u}_k) \leftarrow \texttt{R\_notMsg}(\texttt{u}_1, \ldots, \texttt{u}_k).$$

To produce output, we use the following functionality for Q_{out}.[13] First, we compute inside Q_{out} for each $R^{(k)} \in \sigma_1$ an auxiliary relation $\texttt{R\_known}^{(k)}$ to contain all tuples for which we either know they certainly exist or certainly not exist in input relation R:

$$\begin{aligned}&\texttt{R\_known}(\texttt{u}_1, \ldots, \texttt{u}_k) \leftarrow \texttt{R}(\texttt{u}_1, \ldots, \texttt{u}_k).\\ &\texttt{R\_known}(\texttt{u}_1, \ldots, \texttt{u}_k) \leftarrow \texttt{R\_mem}(\texttt{u}_1, \ldots, \texttt{u}_k).\\ &\texttt{R\_known}(\texttt{u}_1, \ldots, \texttt{u}_k) \leftarrow \texttt{policy}_\texttt{R}(\texttt{u}_1, \ldots, \texttt{u}_k), \neg\texttt{R}(\texttt{u}_1, \ldots, \texttt{u}_k).\\ &\texttt{R\_known}(\texttt{u}_1, \ldots, \texttt{u}_k) \leftarrow \texttt{R\_notMem}(\texttt{u}_1, \ldots, \texttt{u}_k).\end{aligned}$$

[13] Note that the query language for Q_{out} needs to be at least as powerful as the query language for Q.

Next, we check inside Q_{out} for each $R^{(k)} \in \sigma_1$ whether MyAdom is complete for relation R:

$$\texttt{R\_missing}() \leftarrow \texttt{MyAdom}(\texttt{u}_1), \ldots, \texttt{MyAdom}(\texttt{u}_k), \neg\texttt{R\_known}(\texttt{u}_1, \ldots, \texttt{u}_k).$$

Finally, we allow Q_{out} to execute Q when for all input relations $R_1, \ldots, R_n$ the corresponding relations $\texttt{R}_1\texttt{\_missing}, \ldots, \texttt{R}_n\texttt{\_missing}$ are empty:

$\texttt{ready}() \leftarrow \neg\texttt{R}_1\texttt{\_missing}(), \ldots, \neg\texttt{R}_n\texttt{\_missing}().$

```
if ready() then
   compute Q on the local input UNION the R_mem relations.
```

Output lower bound Let Π be as constructed above. Let $\mathcal{N}$ be a network and let $\boldsymbol{P}$ be a distribution policy for σ_1 and $\mathcal{N}$. Let $\boldsymbol{\Pi} = (\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P})$ be the corresponding transducer network. Let I be an input instance over σ_1. Let $\mathcal{R}$ be a run of $\boldsymbol{\Pi}$ on I. We show $Q(I) \subseteq \mathit{out}(\mathcal{R})$.

Fix some node $x \in \mathcal{N}$. Because input facts are sent as messages, there is a transition index i of $\mathcal{R}$ after which x has accumulated I, giving $\texttt{MyAdom} = \mathit{adom}(I) \cup \mathcal{N}$. We now argue there is a transition j of $\mathcal{R}$ with $i < j$ after which MyAdom is complete at x, causing the ready-flag to become true and making x subsequently compute $Q(I)$. If we can argue for each fact $R(a_1, \ldots, a_k)$ over σ_1 with $\{a_1, \ldots, a_k\} \subseteq \texttt{MyAdom}$ that $\texttt{R\_known}(a_1, \ldots, a_k)$ eventually appears in the evaluation of Q_{out} at x, the sought transition index j can be defined as the smallest transition index with active node x, with $i < j$, during which all these R_known-facts appear.[14] So, let $R(a_1, \ldots, a_k)$ be a fact over σ_1 with $\{a_1, \ldots, a_k\} \subseteq \texttt{MyAdom}$.

- Suppose $R(a_1, \ldots, a_k) \in I$. The fact $\texttt{R\_known}(a_1, \ldots, a_k)$ appears on x no later than when all of I is available on x.
- Suppose $R(a_1, \ldots, a_k) \notin I$. By definition of distribution policy, there is some $y \in \mathcal{N}$ such that $y \in \boldsymbol{P}(R(a_1, \ldots, a_k))$. Using the same reasoning as above, after a while, y will also locally have $\texttt{MyAdom} = \mathit{adom}(I) \cup \mathcal{N}$. At that point, the fact $\texttt{policy}_R(a_1, \ldots, a_k)$ will be shown to y. Now $R(a_1, \ldots, a_k) \notin I$ implies that y is not given $R(a_1, \ldots, a_k)$. If $x = y$ then $\texttt{R\_known}(a_1, \ldots, a_k)$ appears directly at x in Q_{out}. If $x \neq y$ then y will first send the message $\texttt{R\_notMsg}(a_1, \ldots, a_k)$. By fairness, this message arrives at x, and x stores $\texttt{R\_notMem}(a_1, \ldots, a_k)$; and thus $\texttt{R\_known}(a_1, \ldots, a_k)$ appears in Q_{out}.

Output upper bound Let $\mathcal{N}$, $\boldsymbol{P}$, $\boldsymbol{\Pi}$, I, and $\mathcal{R}$ be as above. We now show $\mathit{out}(\mathcal{R}) \subseteq Q(I)$.

[14] Note that once an R_known-fact appears in Q_{out}, it will keep doing so by the accumulating nature of the transducer.

Let $\mathbf{f} \in \mathit{out}(\mathcal{R})$. Let $x \in \mathcal{N}$ be a node that outputs $\mathbf{f}$ during some transition i of $\mathcal{R}$. This implies that during transition i, the `ready`-flag is true, and Q is computed over a locally gathered subset $I' \subseteq I$ at x. So, $\mathbf{f} \in Q(I')$. Denote $J = I \setminus I'$. We will now argue that J is domain-distinct from I', so that $\mathbf{f} \in Q(I') \subseteq Q(I' \cup J) = Q(I)$ by domain-distinct-monotonicity of Q.

Towards a contradiction, suppose there is a fact $\mathbf{g} \in J$ with $\mathit{adom}(\mathbf{g}) \subseteq \mathit{adom}(I')$. So, in transition i, we have $\mathit{adom}(\mathbf{g}) \subseteq \texttt{MyAdom}$ since $\mathit{adom}(I') \subseteq \texttt{MyAdom}$. Denote $\mathbf{g} = R(a_1, \ldots, a_k)$. Now, because the `ready`-flag is true, the `R_missing`-flag is false, which implies that the fact $\texttt{R\_known}(a_1, \ldots, a_k)$ exists during transition i. Looking at how relation `R_known` is computed, we distinguish between the following cases, each leading to a contradiction:

- Rule 1 & 2: If the local input fact $R(a_1, \ldots, a_k)$ or the memory fact $\texttt{R\_mem}(a_1, \ldots, a_k)$ is available at x during transition i then $R(a_1, \ldots, a_k) \in I'$, which is false.

- Rule 3: Suppose $\texttt{policy}_R(a_1, \ldots, a_k)$ is shown to x but the input fact $R(a_1, \ldots, a_k)$ is missing at x. The existence of the $\texttt{policy}_R$-fact implies $x \in \boldsymbol{P}(R(a_1, \ldots, a_k))$. Now, since $R(a_1, \ldots, a_k) \in I$ by definition of J, the fact $R(a_1, \ldots, a_k)$ should have been given to x, resulting in $R(a_1, \ldots, a_k) \in I'$, which is false.

- Rule 4: This is similar to Rule 3. If $\texttt{R\_notMem}(a_1, \ldots, a_k)$ is available at x during transition i, some node y with $y \in \boldsymbol{P}(R(a_1, \ldots, a_k))$ had previously sent $\texttt{R\_notMsg}(a_1, \ldots, a_k)$. This means that y does not locally have input fact $R(a_1, \ldots, a_k)$. But $R(a_1, \ldots, a_k) \in I$ (by definition of J) and $y \in \boldsymbol{P}(R(a_1, \ldots, a_k))$ together imply that y is actually given the input fact $R(a_1, \ldots, a_k)$. Since y sends all its input facts to x, we would get $R(a_1, \ldots, a_k) \in I'$, which is false.

Coordination-freeness We argue that Π is coordination-free. Let $\mathcal{N}$ be a network and let I be an input over σ_1. We define a distribution policy $\boldsymbol{P}$ for σ_1 and $\mathcal{N}$ so that $Q(I)$ can be computed with only heartbeat transitions. Concretely, we define $\boldsymbol{P}(\mathbf{f}) = \mathcal{N}$ for each $\mathbf{f} \in \mathit{facts}(\sigma_1)$. Denote $\boldsymbol{\Pi} = (\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P})$.

Fix some arbitrary node $x \in \mathcal{N}$. Transducer network $\boldsymbol{\Pi}$ gives the entire instance I as input to x. Suppose we initially do only heartbeat transitions at x. We show that the `ready`-flag becomes true at x already in the first heartbeat transition, causing x to compute $Q(I)$. Note that $\texttt{MyAdom} = \mathit{adom}(I) \cup \mathcal{N}$ in the first heartbeat transition. Next, for any fact $R(a_1, \ldots, a_k)$ over σ_1 with $\{a_1, \ldots, a_k\} \subseteq \texttt{MyAdom}$, either this fact is in I or it is not in I. In the first case, because x is given all of I as local input, the first rule of relation `R_known` makes the fact $\texttt{R\_known}(a_1, \ldots, a_k)$ available at x during the first heartbeat transition. In the second case, because $x \in \boldsymbol{P}(R(a_1, \ldots, a_k))$ by design of $\boldsymbol{P}$, the third rule of relation `R_known`

makes the fact $\texttt{R\_known}(a_1, \ldots, a_k)$ also available at x during the first heartbeat transition.

Overall, each $\texttt{R\_missing}$-flag is false in the first heartbeat transition of x, making the $\texttt{ready}$-flag true. This heartbeat prefix can be extended to a full fair run of $\mathbf{\Pi}$ on input I. □

Proposition 4.7. $\mathcal{F}_2 \subseteq \mathcal{M}_{disjoint}$.

Proof. The proof strategy is similar to the proof strategy of Proposition 4.5.

Let Q be a query distributedly computed by a policy-aware transducer Π that is coordination-free under domain-guidance. Let Υ denote the transducer schema of Π. Let I and J be two instances over the input schema of Q, such that J is domain-disjoint from I. Let $\mathbf{f} \in Q(I)$. We show $\mathbf{f} \in Q(I \cup J)$.

By coordination-freeness, Π distributedly computes Q on a network $\mathcal{N}$ with at least two nodes. Also by coordination-freeness, there is a domain-guided distribution policy $\boldsymbol{P}_1$ for Υ_{in} and $\mathcal{N}$ such that the transducer network $\mathbf{\Pi}_1 = (\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P}_1)$ when given input I, has a run $\mathcal{R}_1$ in which $Q(I)$ is already computed in a prefix consisting of only heartbeat transitions. Let $x \in \mathcal{N}$ be a node that outputs $\mathbf{f}$ in this prefix. Intuitively, we now show that x can be made to output $\mathbf{f}$ on input $I \cup J$, so that $\mathbf{f} \in Q(I \cup J)$.

Since $\boldsymbol{P}_1$ is domain-guided, there exists a domain-assignment $\boldsymbol{\alpha}_1$ for $\mathcal{N}$ inducing $\boldsymbol{P}_1$. Fix an arbitrary node $y \in \mathcal{N} \setminus \{x\}$. We define the following domain-assignment $\boldsymbol{\alpha}_2$: $\boldsymbol{\alpha}_2(a) = \{y\}$ for all $a \in \mathit{adom}(J)$, and $\boldsymbol{\alpha}_2(a) = \boldsymbol{\alpha}_1(a)$ for all $a \in \mathbf{dom} \setminus \mathit{adom}(J)$. Let $\boldsymbol{P}_2$ be the domain-guided distribution policy for Υ_{in} and $\mathcal{N}$ induced by $\boldsymbol{\alpha}_2$. Denote $\mathbf{\Pi}_2 = (\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P}_2)$. We argue that $\mathbf{\Pi}_2$ on input $I \cup J$ gives node x the same local input facts as $\mathbf{\Pi}_1$ on input I:

- Let $\mathbf{g} \in I$ be a fact given to x by $\mathbf{\Pi}_1$ on input I. There must be a value $a \in \mathit{adom}(\mathbf{g})$ such that $x \in \boldsymbol{\alpha}_1(a)$. But since $a \in \mathit{adom}(I)$, we have $a \notin \mathit{adom}(J)$. We obtain $x \in \boldsymbol{\alpha}_2(a)$ since $\boldsymbol{\alpha}_2(a) = \boldsymbol{\alpha}_1(a)$ by definition of $\boldsymbol{\alpha}_2$. So, $\mathbf{\Pi}_2$ gives $\mathbf{g}$ to x on input $I \cup J$.

- Let $\mathbf{g} \in I \cup J$ be a fact given to x by $\mathbf{\Pi}_2$ on input $I \cup J$. There must be a value $a \in \mathit{adom}(\mathbf{g})$ such that $x \in \boldsymbol{\alpha}_2(a)$. If $a \in \mathit{adom}(J)$ then $x = y$, which is false. So, $a \in \mathit{adom}(I)$. We obtain $x \in \boldsymbol{\alpha}_1(a)$ since $\boldsymbol{\alpha}_2(a) = \boldsymbol{\alpha}_1(a)$ by definition of $\boldsymbol{\alpha}_2$. Moreover, since $a \in \mathit{adom}(I)$ and $\mathit{adom}(I) \cap \mathit{adom}(J) = \emptyset$, we have $\mathbf{g} \in I$. So, $\mathbf{\Pi}_1$ gives $\mathbf{g}$ to x on input I.

Now, similarly as in the proof for Proposition 4.5, when running $\mathbf{\Pi}_2$ on input $I \cup J$, if we initially do only heartbeats with active node x, the node x goes through the same state changes as in the heartbeat-prefix of $\mathcal{R}_1$. So after a while, say after k heartbeats, node x outputs $\mathbf{f}$. This finite prefix can be extended to a full fair run $\mathcal{R}_2$ of $\mathbf{\Pi}_2$ on input $I \cup J$, for which $\mathit{out}(\mathcal{R}_2) = Q(I \cup J)$ holds by assumption on Π. Hence, $\mathbf{f} \in Q(I \cup J)$. □

Proposition 4.8. $\mathcal{M}_{disjoint} \subseteq \mathcal{F}_2$.

Proof. Let $Q \in \mathcal{M}_{disjoint}$. Let σ_1 and σ_2 denote respectively the input and output schema of Q. We construct a transducer Π that computes Q and that is coordination-free under domain-guidance. We start with the intuition.

Intuition Let I be an input over σ_1 that is distributed over a network $\mathcal{N}$, by means of a domain-guided distribution policy. We call a subset $I' \subseteq I$ *safe* when $I' = \{\mathbf{f} \in I \mid adom(\mathbf{f}) \cap adom(I') \neq \emptyset\}$. If a node x has obtained such a safe subset I' then node x may compute $Q(I')$ because $Q(I') \subseteq Q(I)$ by domain-disjoint-monotonicity of Q.

We construct a policy-aware transducer Π that postpones executing Q at a node x until a safe subset of I is collected at x. To start, the nodes broadcast the active domain of their local input fragment. These values are accumulated at each node. Note that when a node x has some value a in relation `MyAdom`, node x is responsible for a under the domain assignment if and only if $\texttt{policy}_R(a, \ldots, a)$ is shown to x for at least one input relation R. If x is indeed responsible for a then x is already locally given all facts of I containing a (because the distribution policy is domain-guided). Now, when x is not responsible for a, node x will send out a request pair (x, a). Any node y responsible for a under the domain assignment will then send all input facts containing a to x. When x has acknowledged all these facts to y, node y will send "OK(x, a)". Now, consider a transition of x, and let $I' \subseteq I$ denote the set of collected input facts at x so far. Node x checks for each value a in `MyAdom` whether x is responsible for a or that x has "OK" for a. If this is so, since always $adom(I') \subseteq \texttt{MyAdom}$, node x has obtained all input facts containing values from $adom(I')$, i.e., I' is safe. Subsequently, x computes $Q(I')$.

We have already argued that no wrong outputs are produced. To see that at least $Q(I)$ is computed, we note that each node x eventually knows of the entire active domain (because the broadcasted domain will eventually arrive), and will thus at some point compute Q on the entire set of collected input facts.

The transducer Π is coordination-free according to the formal definition: for all networks $\mathcal{N}$, and all inputs I, the output will be computed with only heartbeats at some node x when x is made responsible (under the domain assignment) for all values $adom(I) \cup \mathcal{N}$; then x will immediately detect that its local input fragment is safe.

Note that there is no "global" coordination between all nodes. Indeed, the acknowledgments sent from a node x to a node y before obtaining an "OK" message from y can be viewed as the exchange of a big message between just x and y.

Construction We first define a transducer schema Υ as follows:

$$\begin{aligned}
\Upsilon_{\text{in}} &= \sigma_1, \\
\Upsilon_{\text{out}} &= \sigma_2, \\
\Upsilon_{\text{msg}} &= \{\texttt{valueMsg}^{(1)}, \texttt{requestMsg}^{(2)}, \texttt{okMsg}^{(2)}\} \cup \\
&\quad \{\texttt{R_msg}^{(k+2)}, \texttt{R_ackMsg}^{(k+2)} \mid R^{(k)} \in \sigma_1\}, \\
\Upsilon_{\text{mem}} &= \{\texttt{valueMem}^{(1)}, \texttt{requestMem}^{(2)}, \texttt{okMem}^{(1)}\} \cup \\
&\quad \{\texttt{R_mem}^{(k)}, \texttt{R_ackMem}^{(k+2)} \mid R^{(k)} \in \sigma_1\},
\end{aligned}$$

and the contents of Υ_{sys} is uniquely determined by the contents of Υ_{in} using the definition of the transducer schema.

Now we give the transducer $\Pi = (Q_{\text{out}}, Q_{\text{ins}}, Q_{\text{del}}, Q_{\text{snd}})$. For specifying these transducer queries, we use the same syntax flavor as in the proof of Proposition 4.6. We will implement the protocol sketched above step by step. We define Q_{del} to always return $\emptyset$.

In each query, we partition `MyAdom` into auxiliary relations `respAdom` and `otherAdom`. Relation `respAdom` contains the values that the current node is responsible for under the domain assignment. These relations are only visible inside the queries (and are thus not in Υ). For each $v \in \mathbf{var}$ and $k \in \mathbb{N}$, let $(v^k) = (v, \ldots, v)$ denote the tuple in which v is repeated k times. Now, we add to Q_{out}, Q_{ins}, and Q_{snd} for each input relation $R^{(k)}$ the rule

$$\texttt{respAdom}(\texttt{v}) \leftarrow \texttt{MyAdom}(\texttt{v}), \texttt{policy}_{\texttt{R}}(\texttt{v}^k)$$

and at the end we add the single rule

$$\texttt{otherAdom}(\texttt{v}) \leftarrow \texttt{MyAdom}(\texttt{v}), \neg\texttt{respAdom}(\texttt{v}).$$

For any transition of a node x, for any $a \in \texttt{MyAdom}$, one can verify that a fact $\texttt{policy}_R(a^k)$ is exposed to x if and only if x is assigned a by the domain assignment. Relations `respAdom` and `otherAdom` are recomputed in each transition.

To inform all nodes about the existence of all active domain values, we add the following rule to Q_{snd} for each relation $R^{(k)} \in \sigma_1$ and each $i \in \{1, \ldots, k\}$:

$$\texttt{valueMsg}(\texttt{v}) \leftarrow \texttt{R}(\texttt{u}_1, \ldots, \texttt{u}_{\texttt{i}-1}, \texttt{v}, \texttt{u}_{\texttt{i}+1}, \ldots, \texttt{u}_{\texttt{k}}).$$

To remember the received values, we add this rule to Q_{ins}:

$$\texttt{valueMem}(\texttt{v}) \leftarrow \texttt{valueMsg}(\texttt{v}).$$

On each node, this mechanism will cause relation `MyAdom` to eventually contain all active domain values available on the network. This way, nodes will

eventually collect the entire input (see below). To request input facts, we add this rule to Q_{snd}:

$$\texttt{requestMsg}(\texttt{x}, \texttt{v}) \leftarrow \texttt{Id}(\texttt{x}), \texttt{otherAdom}(\texttt{v}).$$

To send out the requested input facts, we add the following rule to Q_{snd} for each $R^{(k)} \in \sigma_1$ and each $i \in \{1, \ldots, k\}$:

$$\texttt{R\_msg}(\texttt{x}, \texttt{v}, \texttt{u}_1, \ldots, \texttt{u}_{i-1}, \texttt{v}, \texttt{u}_{i+1}, \ldots, \texttt{u}_k) \leftarrow \texttt{requestMsg}(\texttt{x}, \texttt{v}), \texttt{respAdom}(\texttt{v}), \texttt{R}(\texttt{u}_1, \ldots, \texttt{u}_{i-1}, \texttt{v}, \texttt{u}_{i+1}, \ldots, \texttt{u}_k).$$

We remember the request by adding the following rule to Q_{ins}:

$$\texttt{requestMem}(\texttt{x}, \texttt{v}) \leftarrow \texttt{requestMsg}(\texttt{x}, \texttt{v}), \texttt{respAdom}(\texttt{v}).$$

To handle received input facts, we add the following rule to Q_{ins} for every $R^{(k)} \in \sigma_1$:

$$\texttt{R\_mem}(\texttt{u}_1, \ldots, \texttt{u}_k) \leftarrow \texttt{R\_msg}(\texttt{x}, \texttt{v}, \texttt{u}_1, \ldots, \texttt{u}_k), \texttt{Id}(\texttt{x}).$$

We acknowledge the receipt by adding the following rule to Q_{snd}:

$$\texttt{R\_ackMsg}(\texttt{x}, \texttt{v}, \texttt{u}_1, \ldots, \texttt{u}_k) \leftarrow \texttt{R\_msg}(\texttt{x}, \texttt{v}, \texttt{u}_1, .., \texttt{u}_k), \texttt{Id}(\texttt{x}).$$

To handle received acknowledgments, we add the following rule to Q_{ins}:

$$\texttt{R\_ackMem}(\texttt{x}, \texttt{v}, \texttt{u}_1, \ldots, \texttt{u}_k) \leftarrow \texttt{R\_ackMsg}(\texttt{x}, \texttt{v}, \texttt{u}_1, \ldots, \texttt{u}_k), \texttt{requestMem}(\texttt{x}, \texttt{v}).$$

To check whether a requesting node has acknowledged all input facts containing the value from the request, we first compute an auxiliary relation `R_missing` for each relation R in σ_1. Concretely, we add the following rule to Q_{snd} for each $R^{(k)} \in \sigma_1$ and each $i \in \{1, \ldots, k\}$:

$$\texttt{R\_missing}(\texttt{x}, \texttt{v}) \leftarrow \texttt{requestMem}(\texttt{x}, \texttt{v}), \texttt{R}(\texttt{u}_1, \ldots, \texttt{u}_{i-1}, \texttt{v}, \texttt{u}_{i+1}, \ldots, \texttt{u}_k), \neg\texttt{R\_ackMem}(\texttt{x}, \texttt{v}, \texttt{u}_1, \ldots, \texttt{u}_{i-1}, \texttt{v}, \texttt{u}_{i+1}, \ldots, \texttt{u}_k).$$

These relations are recomputed in each transition. Now, we add the following rule to Q_{snd} where $R_1, \ldots, R_n$ are the relations from σ_1:

$$\texttt{okMsg}(\texttt{x}, \texttt{v}) \leftarrow \texttt{requestMem}(\texttt{x}, \texttt{v}), \neg\texttt{R}_1\texttt{\_missing}(\texttt{x}, \texttt{v}), \ldots, \neg\texttt{R}_n\texttt{\_missing}(\texttt{x}, \texttt{v}).$$

To receive these `okMsg`-facts, we add the following rule to Q_{ins}:

$$\texttt{okMem}(\texttt{v}) \leftarrow \texttt{okMsg}(\texttt{x}, \texttt{v}), \texttt{Id}(\texttt{x}).$$

Finally, we use the following functionality for Q_{out}:[15]

$\texttt{missing}() \leftarrow \texttt{otherAdom}(\texttt{v}), \neg\texttt{okMem}(\texttt{v}).$

$\texttt{ready}() \leftarrow \neg\texttt{missing}().$

`if ready() then compute` Q `on the local input UNION the R_mem relations.`

Next we will show that Π correctly computes Q, and is coordination-free.

[15] Recall that `otherAdom` is an auxiliary relation computed inside Q_{out}; see previously.

Output lower bound Let Π be as constructed above. Let $\mathcal{N}$ be a network and let $\boldsymbol{P}$ be a domain-guided distribution policy for σ_1 and $\mathcal{N}$. Let $\boldsymbol{\Pi} = (\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P})$ be the corresponding domain-guided transducer network. Let I be an input instance over σ_1. Let $\mathcal{R}$ be a run of $\boldsymbol{\Pi}$ on I. We show $Q(I) \subseteq \mathit{out}(\mathcal{R})$.

Fix some $x \in \mathcal{N}$. We show that x outputs $Q(I)$ at some point during $\mathcal{R}$. First, because the nodes broadcast all values from their local input, and these values are accumulated at x, there is a transition of $\mathcal{R}$ after which continuously $\texttt{MyAdom} = \mathit{adom}(I) \cup \mathcal{N}$ at x.

Now, recall that `MyAdom` is partitioned into `respAdom` and `otherAdom`. Because $\boldsymbol{P}$ is domain-guided, we have already given all $\mathbf{f} \in I$ for which $\mathit{adom}(f) \cap \texttt{respAdom} \neq \emptyset$ to x as local input. We argue that x also obtains all facts $\mathbf{f} \in I$ for which $\mathit{adom}(\mathbf{f}) \cap \texttt{otherAdom} \neq \emptyset$. Indeed, for at least one value $b \in \mathit{adom}(\mathbf{f}) \cap \texttt{otherAdom}$, node x will send $\texttt{requestMsg}(x, b)$. There surely exists some node y that is responsible for b under the domain assignment behind $\boldsymbol{P}$. Upon receiving the request, node y will send all facts of I that contain b to x, including $\mathbf{f}$. Node x will store these facts in its memory, acknowledges them, and eventually receives $\texttt{okMsg}(x, b)$ from y that is also stored.[16] So, there eventually is a transition of x during which x has all of I (either in local relations or in memory relations), and during which x also has $\texttt{okMem}(b)$ for all $b \in \texttt{otherAdom}$. During such a transition, the `ready`-flag is true and Q is computed on I.

Output upper bound Let $\mathcal{N}$, $\boldsymbol{P}$, $\boldsymbol{\Pi}$, I, and $\mathcal{R}$ be as above. We now show $\mathit{out}(\mathcal{R}) \subseteq Q(I)$.

Let $\mathbf{f} \in \mathit{out}(\mathcal{R})$. Let $x \in \mathcal{N}$ be a node that outputs $\mathbf{f}$ during some transition i of $\mathcal{R}$. This means that during i the `ready`-flag was true and that Q was subsequently applied to some locally available set $I' \subseteq I$ (i.e., the local input united with any received input facts). So, $\mathbf{f} \in Q(I')$. Denote $J = I \setminus I'$. If we can show that $\mathit{adom}(J) \cap \mathit{adom}(I') = \emptyset$ then we obtain $\mathbf{f} \in Q(I') \subseteq Q(I' \cup J) = Q(I)$ by domain-disjoint-monotonicity of Q, as desired.

Towards a proof by contradiction, suppose there is some $\mathbf{g} \in J$ and a value $a \in \mathit{adom}(\mathbf{g}) \cap \mathit{adom}(I')$. Because $a \in \mathit{adom}(I')$, we have $a \in \texttt{MyAdom}$ during transition i. Recall that `MyAdom` is partitioned into `respAdom` and `otherAdom`:

- If $a \in \texttt{respAdom}$, then $\mathbf{g}$ would have been given as local input to x, resulting in $\mathbf{g} \in I'$, which is false.
- Suppose $a \in \texttt{otherAdom}$. Because the `ready`-flag is true during i, node x has $\texttt{okMem}(a)$. This means that some node y has sent $\texttt{okMsg}(x, a)$,

[16] In general, a node identifier $b \in \mathcal{N}$ will not occur in the input. In that case, some node is still responsible for b under the domain assignment; this node will send $\texttt{okMsg}(x, b)$ after receiving $\texttt{requestMsg}(x, b)$ without the need for acknowledgements.

where y is responsible for a under the domain assignment behind $\boldsymbol{P}$. This in turn means that x has received and acknowledged all input facts containing value a, including $\mathbf{g}$. So, again we obtain $\mathbf{g} \in I'$, which is false.[17]

Coordination-freeness We argue that Π is coordination-free under domain-guidance. Let $\mathcal{N}$ be a network. Let I be an input for Q. Consider the domain assignment $\boldsymbol{\alpha}$ for $\mathcal{N}$ that assigns **dom** to each node. Note that the corresponding domain-guided distribution policy $\boldsymbol{P}$ for σ_1 and $\mathcal{N}$ always assigns the full input to each node. Denote $\boldsymbol{\Pi} = (\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P})$. We give I as input to $\boldsymbol{\Pi}$.

Now, fix some node $x \in \mathcal{N}$. Suppose we initially do only heartbeats at x. During the first heartbeat transition, we have `respAdom` = `MyAdom` by design of $\boldsymbol{\alpha}$. So, the `ready`-flag immediately becomes true because `otherAdom` = $\emptyset$. Then Q is computed over all local input facts, which are all facts of I by design of $\boldsymbol{\alpha}$. Hence, $Q(I)$ is output during the first heartbeat transition of x. We can extend this heartbeat prefix to an infinite fair run $\mathcal{R}$ of $\boldsymbol{\Pi}$ on input I. □

4.4 Discussion

We contrast the distributed evaluation algorithms in the proofs of Proposition 4.6 (for $\mathcal{M}_{distinct}$) and Proposition 4.8 (for $\mathcal{M}_{disjoint}$) with a distributed evaluation algorithm for the class $\mathcal{M}$ of purely monotone queries. It is important to realize that the algorithms for $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$ are naive in the sense that the whole database is sent to all nodes and every node computes the result of the query. It is the type of monotonicity that determines when a node can start producing output:

- $\mathcal{M}$: every node broadcasts all local input facts; output is generated for every newly received fact;
- $\mathcal{M}_{distinct}$: every node broadcasts all local input facts as well as absences of facts; output is generated at a node when the missing facts are surely domain-distinct from the already collected facts; and,
- $\mathcal{M}_{disjoint}$: every node broadcasts the values in the local input; whenever a new value is received that the node is not responsible for, a coordination protocol is initiated with nodes responsible for this value; output is generated at a node when the missing facts are surely domain-disjoint from the already collected facts.

[17] Note that during the protocol, the fact $\mathbf{g}$ is indeed stored at x strictly before `okMem`(a) is stored at x, even if x is talking in parallel to two nodes y and z that are both responsible for a.

While it might seem contradictory that the coordination-free evaluation of queries in $\mathcal{M}_{disjoint}$ requires the use of a coordination protocol, it is important to realize that this coordination is only determined by the way data is distributed and does not require *global* coordination between *all* nodes. Indeed, we show below that transducers that do not access the system relation `All`, containing the names of all the nodes in the network, are immediately coordination-free. Moreover, the classes of queries that are distributedly computed by (policy-aware) transducers without relation `All` still capture $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$. That is, in absence of relation `All`, we do not need the notion of coordination-freeness to characterize $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$.

Formally, to prevent usage of relation `All`, at a node x we modify the semantics of transitions from Section 4.1.3 as follows: we define the set A now as $A = adom(J) \cup \{x\}$; and, the set S is defined as before, but now without the `All`-facts. For this resulting model, let $\mathcal{A}_1$ denote the set of queries distributedly computed by policy-aware transducers, and let $\mathcal{A}_2$ denote the set of queries distributedly computed by policy-aware transducers under domain-guidance.

Theorem 4.9. *$\mathcal{A}_1 = \mathcal{M}_{distinct}$ and $\mathcal{A}_2 = \mathcal{M}_{disjoint}$*

Proof. First, the transducers constructed for the proofs of Proposition 4.6 ($\mathcal{M}_{distinct} \subseteq \mathcal{F}_1$) and Proposition 4.8 ($\mathcal{M}_{disjoint} \subseteq \mathcal{F}_2$) can be used as is to respectively show that $\mathcal{M}_{distinct} \subseteq \mathcal{A}_1$ and $\mathcal{M}_{disjoint} \subseteq \mathcal{A}_2$, because they do not use `All`.

We sketch the proof of $\mathcal{A}_1 \subseteq \mathcal{M}_{distinct}$. Let Q be a query that is distributedly computed by a policy-aware transducer Π without `All`. By assumption, transducer Π also computes Q on a single-node network $\{x\}$. So, on input I, the single-node transducer network will produce $Q(I)$ with just heartbeats (no messages can be sent). Let J be an input instance for Q that is domain-distinct from I. To show $Q(I) \subseteq Q(I \cup J)$, we consider a two-node network $\{x, y\}$ on which we run Π, and consider the distribution policy that assigns J to just y and the other facts to x. Then x is still given precisely I when the two-node transducer network is given $I \cup J$ as input. Now, if we do only heartbeats at x, then x will behave the same as on the single-node network because it can not detect the difference with the two-node network in absence of relation `All`. So, x will produce $Q(I)$ after a finite number of heartbeats. This prefix can be extended to a full fair run of the two-node transducer network on input $I \cup J$, as follows: we systematically deliver all messages available in the last configuration of the prefix; we activate both nodes x and y infinitely often; and, we also keep delivering any newly sent messages. By assumption, the transducer network consisting of Π and node set $\{x, y\}$ will eventually compute all of $Q(I \cup J)$ in this fair run.

The inclusion $\mathcal{A}_2 \subseteq \mathcal{M}_{disjoint}$ can be shown similarly, except that now J is chosen to be domain-disjoint from I, and we assign all values of $adom(J)$ to y and all other domain values to x. □

It is interesting to note what happens when transducers are not aware of the distribution policies, i.e., when we do not provide the $\texttt{policy}_R$-relations. In the resulting model, the set $\mathcal{F}_0$ of queries distributedly computed by coordination-free transducers is precisely the set $\mathcal{M}$ of monotone queries; and, relating to the above, the set $\mathcal{A}_0$ of queries distributedly computed by transducers without relation `All` is also $\mathcal{M}$ [18].

For completeness, we also mention the existence of so-called *oblivious* transducers in the original transducer model of Ameloot et al. [18]: these transducers may use neither relation `Id` nor relation `All`. The set of queries distributedly computed by oblivious transducers is again the set $\mathcal{M}$.

Corollary 4.10. $\mathcal{F}_0 = \mathcal{A}_0 = \mathcal{M}$.

5 Datalog fragments

5.1 Semi-connected Datalog$^\neg$

As mentioned in the introduction, the original formulation of the CALM-conjecture links coordination-free computation to Datalog. It is therefore interesting to investigate subclasses of Datalog with negation that remain within $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$. While SP-Datalog $\subseteq \mathcal{M}_{distinct} (= \mathcal{E})$ [7], it is not known whether SP-Datalog can be further extended while remaining within $\mathcal{E}$.

We next identify a fragment of Datalog$^\neg$ which is domain-disjoint-monotone. Let φ be a Datalog$^\neg$ rule. We define $graph^+(\varphi)$ as the graph where nodes are the variables in positive body atoms of φ, and there is an edge between two variables if they occur together in a positive body atom of φ. We say φ is *connected* if $graph^+(\varphi)$ is connected. We say that an SP-Datalog program is *connected* when all rules are connected.

Definition 3. Let P be a program in Datalog$^\neg$. Then P is a *connected stratified datalog program*, when there is a stratification for P such that all strata are connected SP-Datalog programs. We say that P is *semi-connected* when there is a stratification such that all strata except possibly the last one are connected SP-Datalog programs.

In the following, we denote the class of connected and semi-connected stratified Datalog$^\neg$ programs by con-Datalog$^\neg$ and semicon-Datalog$^\neg$, respectively. Then *(i)* SP-Datalog $\subsetneq$ semicon-Datalog$^\neg$, *(ii)* SP-Datalog $\not\subseteq$ con-Datalog$^\neg$, *(iii)* con-Datalog$^\neg$ $\subsetneq$ semicon-Datalog$^\neg$, and *(iv)* semicon-Datalog$^\neg$ $\subsetneq$ Datalog$^\neg$.

Example 5.1. *Consider the following Datalog$^\neg$ program P_1, which expects a graph as its input and asks for all the nodes that are not part of a (directed)*

triangle:

$$\begin{array}{rcl} T(x) & \leftarrow & E(x,y), E(y,z), E(z,x), y \neq x, y \neq z, x \neq z \\ O(x) & \leftarrow & \neg T(x), Adom(x) \end{array}$$

Then, P_1 is in con-Datalog$^\neg$, but $P_1 \notin \mathcal{M}_{distinct}$. Indeed, $P_1(\{E(a,b)\}) \neq \emptyset$, while $P_1(\{E(a,b)\} \cup \{E(b,c), E(c,a)\}) = \emptyset$. Therefore, $P_1 \notin$ SP-Datalog.

Consider the program P_2 which is not a semicon-Datalog$^\neg$ program:

$$\begin{array}{rcl} T(x,y,z) & \leftarrow & E(x,y), E(y,z), E(z,x), y \neq x, y \neq z, x \neq z \\ D(x_1) & \leftarrow & T(x_1,x_2,x_3), T(y_1,y_2,y_3), \bigwedge_{1 \leq i,j \leq 3} x_i \neq y_j \\ O(x) & \leftarrow & \neg D(x), Adom(x) \end{array}$$

Note that the expressed query is not in $\mathcal{M}_{disjoint}$. Indeed, for domain disjoint instances $I = \{E(a,b), E(b,c), E(c,a)\}$ and $J = \{E(d,e), E(e,f), E(f,d)\}$, $P_2(I) \neq \emptyset$, while $P_2(I \cup J) = \emptyset$. As we will see shortly, this implies that the query itself can not be defined by any semicon-Datalog$^\neg$ program. □

We next relate connected programs to distributed evaluation over components. An instance J is a *component* of an instance I when $J \subseteq I$, $J \neq \emptyset$, $adom(J) \cap adom(I \setminus J) = \emptyset$ and J is minimal with this property. That is, there is no strict nonempty subset J' of J for which $adom(J') \cap adom(I \setminus J') = \emptyset$. Intuitively, a component is complete w.r.t. its active domain as it either contains all the facts in I in which a certain domain element occurs, or it contains none of these facts. Further notice that for component J from I: $I \setminus J$ is domain disjoint from J and vice versa. Denote by $co(I)$ the components of I.

Definition 4. A query Q *distributes over components* if for all instances I:

$$Q(I) = \bigcup_{C \in co(I)} Q(C).$$

Lemma 5.3 highlights an important property of con-Datalog$^\neg$ and forms a crucial part of the proof of Theorem 5.4. Towards a proof for Lemma 5.3 we first show the following preliminary result:

Lemma 5.2. *Every connected SP-Datalog program distributes over components.*

Proof. Let P be a connected SP-Datalog program and I an instance. We need to show that $P(I) = \bigcup_{C \in co(I)} P(C)$.

Let $C \in co(I)$. Then $I \setminus C$ is domain disjoint from C. By domain-disjoint-monotonicity of P, it thus follows that $P(C) \subseteq P(I)$. Therefore, $\bigcup_{C \in co(I)} P(C) \subseteq P(I)$.

We next show that for every fact $\mathbf{f}$,

$$\text{if } \mathbf{f} \in P(I) \text{ then there is a } C_{\mathbf{f}} \in co(I) \text{ such that } \mathbf{f} \in P(C_{\mathbf{f}}). \qquad (\dagger)$$

The latter implies $P(I) \subseteq \bigcup_{C \in co(I)} P(C)$. We prove (†) by induction on the number of derivation steps. Define $\mathbf{T}_P^i(I)$ as the output of the immediate consequence operator of P on I after i iterations, and $\mathbf{T}_P^*(I)$ as its fixpoint on I. Then, we show that for all $i \geq 1$,

$$\text{if } \mathbf{f} \in \mathbf{T}_P^i(I) \text{ then there is a } C_{\mathbf{f}} \in co(I) \text{ such that } \mathbf{f} \in \mathbf{T}_P^i(C_{\mathbf{f}}). \qquad (\ddagger)$$

Clearly, for $i = 1$, (‡) follows directly from the connectedness of P. Assume $\mathbf{f} \in \mathbf{T}_P^{i+1}(I)$. Then there is a rule φ in P and a valuation V such that $V(pos_\varphi) \subseteq \mathbf{T}_P^i(I)$ and $V(neg_\varphi) \cap \mathbf{T}_P^i(I) = \emptyset$. By induction, for every fact $\mathbf{g} \in V(pos_\varphi)$ there is a $C_{\mathbf{g}} \in co(I)$ such that $\mathbf{g} \in \mathbf{T}_P^i(C_{\mathbf{g}})$. As φ is connected and $adom(\mathbf{T}_P^i(C')) \cap adom(\mathbf{T}_P^i(C'')) = \emptyset$ for every two distinct components $C', C'' \in co(I)$, there must be a $C \in co(I)$, such that $C_{\mathbf{g}} = C_{\mathbf{g}'} = C$ for all $\mathbf{g}, \mathbf{g}' \in V(pos_\varphi)$. So, $\mathbf{f} \in \mathbf{T}_P^{i+1}(C)$. This completes the proof. □

Lemma 5.3. *Every query in con-Datalog$^\neg$ distributes over components.*

Proof. Let P be a con-Datalog$^\neg$ program with stratification $P_1, \ldots, P_m$ where for every $i \leq m$, P_i is a connected SP-Datalog program.

We show $P(I) = \bigcup_{C \in co(I)} P(C)$, for all I, by induction on the number of strata for P.

Thereto, define $P_{\leq i}$ as the composition $P_i \circ \cdots \circ P_1$ of the first i strata. We show by induction that for every instance I,

$$P_{\leq i}(I) = \bigcup_{C \in co(I)} P_{\leq i}(C).$$

Case $i = 1$. Follows directly from Lemma 5.2.

Case $i > 1$. Let $C \in co(I)$. Then, by induction, $P_{\leq i-1}(I \setminus C)$ is domain disjoint from $P_{\leq i-1}(C)$. Indeed, $C \in co(I)$ implies $adom(C) \cap adom(I \setminus C) = \emptyset$ and therewith it follows that $adom(P_{\leq i-1}(C)) \cap adom(P_{\leq i-1}(I \setminus C)) = \emptyset$.

Now, $P_i(P_{\leq i-1}(C)) \subseteq P_i(P_{\leq i-1}(I))$ (by domain-disjoint-monotonicity of SP-Datalog.). So, $\bigcup_{C \in co(I)} P_{\leq i}(C) \subseteq P_{\leq i}(I)$.

We next show $P_{\leq i}(I) \subseteq \bigcup_{C \in co(I)} P_{\leq i}(C)$. From Lemma 5.2, it follows that

$$P_{\leq i}(I) = P_i(P_{\leq i-1}(I)) = \bigcup_{D \in co(P_{\leq i-1}(I))} P_i(D).$$

Let $D \in co(P_{\leq i-1}(I))$. By minimality, there is no strict non-empty subset D' of D such that $adom(D') \cap adom(D \setminus D') = \emptyset$. Therefore, D cannot be dispersed over the components of $P_{\leq i-1}(I)$. Indeed, it follows from the definition of component that $adom(C) \cap adom(C') = \emptyset$ for every $C, C' \in co(I)$, implying $adom(P_{\leq i-1}(C)) \cap adom(P_{\leq i-1}(C')) = \emptyset$. So, there is a $C \in co(I)$ such that $D \subseteq P_{\leq i-1}(C)$.

Although $P_{\leq i-1}(C)$ itself may have multiple components, D must be one of them, and thus $P_{\leq i-1}(C) \setminus D$ is domain-disjoint from D. Indeed, by definition of component, $adom(D) \cap adom(P_{\leq i-1}(I) \setminus D) = \emptyset$. As $P_{\leq i-1}(C) \subseteq P_{\leq i-1}(I)$ (by the induction hypothesis), $adom(D) \cap adom(P_{\leq i-1}(C) \setminus D) = \emptyset$ follows. The domain-distinct-montonicity of P_i together with $D \subseteq P_{\leq i-1}(C)$ and $P_{\leq i-1}(C) \setminus D$ being domain-disjoint from D imply $P_i(D) \subseteq P_i(P_{\leq i-1}(C))$. Therefore,

$$P_{\leq i}(I) = \bigcup_{D \in co(P_{\leq i-1}(I))} P_i(D) \subseteq \bigcup_{C \in co(I)} P_{\leq i}(C).$$

□

We are now armed to prove the following theorem:

Theorem 5.4. *semicon-Datalog*$^\neg \subseteq \mathcal{M}_{disjoint}$

Proof. Let P be a semicon-Datalog$^\neg$ program with stratification $P_1, \ldots, P_s$ where P_i is a connected SP-Datalog program for $i < s$. Define $P_{\leq i}$ as the composition $P_i \circ \cdots \circ P_1$ of the first i strata. Clearly, $P = P_{\leq s} = P_s \circ P_{\leq s-1}$. Let I and J be two instances, where J is domain disjoint from I, i.e., $adom(I) \cap adom(J) = \emptyset$. Then, it follows that $adom(P_{\leq s-1}(I)) \cap adom(P_{\leq s-1}(J)) = \emptyset$ implying $P_{\leq s-1}(J)$ to be domain disjoint from $P_{\leq s-1}(I)$. By domain-disjoint-monotonicity of SP-Datalog programs, it follows that

$$P_s(P_{\leq s-1}(I)) \subseteq P_s(P_{\leq s-1}(I) \cup P_{\leq s-1}(J)). \qquad (\dagger)$$

We next show that

$$P_{\leq s-1}(I) \cup P_{\leq s-1}(J) = P_{\leq s-1}(I \cup J), \qquad (\ddagger)$$

which together with (†) implies that $P(I) \subseteq P(I \cup J)$ and therefore $P \in \mathcal{M}_{disjoint}$.

It remains to prove (‡). Lemma 5.3 implies that $P_{\leq s-1}(I) = \bigcup_{C \in co(I)} P_{\leq s-1}(C)$, $P_{\leq s-1}(J) = \bigcup_{D \in co(J)} P_{\leq s-1}(D)$ and $P_{\leq s-1}(I \cup J) = \bigcup_{K \in co(I \cup J)} P_{\leq s-1}(K)$. Since I and J are domain disjoint, $co(I) \cup co(J) = co(I \cup J)$. Therefore,

$$\begin{aligned} & P_{\leq s-1}(I) \cup P_{\leq s-1}(J) \\ &= \bigcup_{C \in co(I)} P_{\leq s-1}(C) \cup \bigcup_{D \in co(J)} P_{\leq s-1}(D) \\ &= \bigcup_{K \in co(I \cup J)} P_{\leq s-1}(K) \\ &= P_{\leq s-1}(I \cup J). \end{aligned}$$

This completes the proof. □

5.2 Datalog$^\neg$ with value invention

While Datalog and SP-Datalog are included the classes $\mathcal{M}$ and $\mathcal{E}$, the inclusion is strict in both cases. One way to capture these classes is to add a mechanism to invent new values. In particular, it is known that the classes $\mathcal{M}$ and $\mathcal{E}$ (and therefore $\mathcal{M}_{distinct}$) are captured by fragments of ILOG$^\neg$, a declarative formalism in the style of stratified Datalog$^\neg$ originally introduced in the context of object databases by Hull and Yoshikawa [32]. ILOG$^\neg$ introduces a restricted form of value invention. In this section, we show that adding value invention to semi-connected Datalog$^\neg$ suffices to capture $\mathcal{M}_{disjoint}$.

We follow Cabibbo [23] for the definition of ILOG$^\neg$ and assume the existence of *invention relations*. These are relations with a distinguished first position, called the *invention position* or the *invention attribute*. An *invention atom* is an atom of the form $R(*, u_1, \ldots, u_k)$, where R is an invention relation and each $u_i \in \mathbf{var}$. The symbol $*$ is called the *invention symbol*. An ILOG$^\neg$ program P over σ is a Datalog program with negation over σ where we allow head atoms to be either (ordinary) relation atoms or invention atoms.

Before defining the semantics of ILOG$^\neg$, we associate to each invention relation R a distinct *Skolem functor* f_R of arity $ar(R)-1$. The *Skolemization* of P, denoted by $Skol(P)$, is the set of rules in P where every occurrence of the invention symbol is replaced by appropriate Skolem functor terms. For example, the Skolemization of the rule

$$R(*, x_1, x_2) \leftarrow E(x_1, x_2)$$

is

$$R(f_R(x_1, x_2), x_1, x_2) \leftarrow E(x_1, x_2).$$

The semantics of ILOG$^\neg$ is defined similar to the semantics of Datalog with negation, but valuations are applied on the Skolemized rules instead of the rules itself and are w.r.t. the Herbrand universe $\mathcal{H}_P$ for P which is the set of all ground terms built using elements from **dom** and Skolem functors for invention relations in P. When the repeated application of the immediate consequence operator eventually gives rise to relations of infinite size, the output of the program is undefined, otherwise the output consists of the facts in the output relations.

An ILOG$^\neg$ program P is *safe* when the output contains no invented values. Although safety is an undecidable property, there are syntactical restrictions that ensure safe programs. One of these is called weakly safe which we define next. We consider the set of *unsafe positions* in atoms of P. This is the smallest set S containing pairs of the form (R, i), where R is a relation name and $i \leq ar(R)$, such that

- $(R, 1) \in S$ for every invention relation R; and,

- if $(R, i) \in S$ and there is a rule φ in P such that $R(x_1, \ldots, x_k) \in pos_\varphi$, $T(y_1, \ldots, y_\ell) = head_\varphi$, and x_i and y_j refer to the same variable, then $(T, j) \in S$.

An ILOG$^\neg$ program P is *weakly safe* when the output relations of P do not contain unsafe positions. Note that a weakly safe program is always safe. We denote weakly safe ILOG$^\neg$ by wILOG$^\neg$. Furthermore, wILOG$^\neg$ where negation is restricted to predicates in $edb(P)$ is denoted by SP-wILOG for semi-positive wILOG$^\neg$. Stratification can be defined for wILOG$^\neg$ in the same way as for Datalog$^\neg$. As before, we only consider stratified negation in this paper.

Example 5.5. *As an example consider the program P_{path} over a binary relation* `E`*:*

$$\begin{array}{lcl}
\texttt{path}^{nil}(*) & \leftarrow & \\
\texttt{path}(*, x, y, nil) & \leftarrow & \texttt{E}(x, y), \texttt{path}^{nil}(nil) \\
\texttt{path}(*, x, z, s) & \leftarrow & \texttt{E}(x, y), \texttt{path}(s, y, z, t) \\
\texttt{output}(x, y) & \leftarrow & \texttt{path}(s, x, y, t)
\end{array}$$

First notice that P_{path} is weakly safe. Indeed, the unsafe position for `path` *are 1 and 4. Consequently, no positions in* `output` *are unsafe. The program outputs the transitive closure over relation* `E` *in a left-linear fashion. But, during the computation it also assigns to every path a unique (invented) value, and a reference to the preceding path where it was derived from.*

Specifically, the relation $\texttt{path}^{nil}$ *contains an invented value used as dummy reference or id in the representation for paths of length one (i.e., where there is no preceding path to refer to) while the relation* `path` *stores the derived paths. In particular, in a fact of the form* $\texttt{path}(s, x, y, t)$*, x and y are nodes and there is a path in* `E` *from x to y, s is an id for this path from x to y, and t is an id for a path starting from some node z to node y, where there is an edge between x and z. Using invented values to create linked lists as in the relation* `path` *will be exploited below to generate enumerations of database instances.*

Let f_p and f_n be the Skolem functors associated to `path` *and* $\texttt{path}^{nil}$*, respectively. When the input graph is acyclic there are only a finite number of distinct paths. So, computing the transitive closure in a left-linear fashion, eventually there are no more values to be added. The output of the immediate consequence operator for P_{path} on $I = \{E(a, b), E(b, c)\}$ is given below (we only show newly derived facts):*

i	*output of* $\boldsymbol{T}$ *in the* i*th iteration step on* I
1:	$\{\texttt{path}^{nil}(f_n())\}$
2:	$\{\texttt{path}(f_p(b,c,f_n()),b,c,f_n()),\texttt{path}(f_p(a,b,f_n()),a,b,f_n())\}$
3:	$\{\texttt{path}(f_p(a,c,f_p(b,c,f_n())),a,c,f_p(b,c,f_n())),\texttt{output}(b,c),\texttt{output}(a,b)\}$
4:	$\{\texttt{output}(a,c)\}$
5:	$\emptyset$

However, when the input contains a cycle, the immediate consequence operator will never reach a fixpoint. Take for example $J = \{E(a,b), E(b,a)\}$*:*

i	*output of* $\boldsymbol{T}$ *in the* i*th iteration step on* J
1:	$\{\texttt{path}^{nil}(f_n())\}$
2:	$\{\texttt{path}(f_p(b,a,f_n()),b,a,f_n()),\texttt{path}(f_p(a,b,f_n()),a,b,f_n())\}$
3:	$\{\texttt{path}(f_p(a,a,f_p(b,a,f_n())),a,a,f_p(b,a,f_n())),$ $\texttt{path}(f_p(b,b,f_p(a,b,f_n())),b,b,f_p(a,b,f_n())),\texttt{output}(a,b),\texttt{output}(b,a)\}$
4:	$\{\texttt{path}(f_p(b,a,f_p(a,a,f_p(b,a,f_n()))),b,a,f_p(a,a,f_p(b,a,f_n()))),$ $\texttt{path}(f_p(a,b,f_p(b,b,f_p(a,b,f_n()))),a,b,f_p(b,b,f_p(a,b,f_n()))),$ $\texttt{output}(a,a),\texttt{output}(b,b)\}$
5:	$\{\texttt{path}(f_p(a,a,f_p(b,a,f_p(a,a,f_p(b,a,f_n())))),a,a,f_p(b,a,f_p(a,a,f_p(b,a,f_n())))),$ $\texttt{path}(f_p(b,b,f_p(a,b,f_p(b,b,f_p(a,b,f_n())))),b,b,f_p(a,b,f_p(b,b,f_p(a,b,f_n()))))\}$
...	...

Cabibbo investigated the expressiveness of ILOG$^\neg$ over relational databases [23] and obtained the following results: stratified wILOG$^\neg$ programs with two strata capture the class of all computable queries; wILOG$^\neg$ programs where negation is restricted to inequalities and extensional database predicates captures $\mathcal{M}$ and $\mathcal{E}$, respectively.

We introduce the class of *semi-connected wILOG*$^\neg$ programs and show that it captures precisely the domain-disjoint-monotone queries. Analogous to the definitions for Datalog$^\neg$, we say that a SP-wILOG program is *connected* when all rules are connected. A wILOG$^\neg$ program P is *semi-connected* when there is a stratification for P such that all strata, except possibly the last one, are connected SP-wILOG programs. We denote the set of all semi-connected wILOG$^\neg$ programs with semicon-wILOG$^\neg$. Notice that semicon-wILOG$^\neg$ is a conservative extension of semicon-Datalog$^\neg$ that strictly subsumes SP-wILOG.

We now give some definitions that generalize notions from datalog queries to wILOG$^\neg$. The only difference is that now queries can be undefined. A query Q over σ is *domain-disjoint-defined* if for each pair of instances I, J over σ, J is domain-disjoint from I and Q is defined over $I \cup J$ imply that Q is defined over I. A (partial) query is *domain-disjoint-monotone* if it is domain-disjoint-defined and for each instance I and J, J is domain-disjoint from I and Q is defined over $I \cup J$ implies that $Q(I) \subseteq Q(I \cup J)$.

We show in this section that every query defined by a semicon-wILOG$^\neg$ program is domain-disjoint-monotone. Thereto, we first show that, similar to connected SP-Datalog and con-Datalog$^\neg$ programs, connected SP-wILOG and connected wILOG$^\neg$ programs distribute over components.

Lemma 5.6. *Every connected SP-wILOG program distributes over components.*

Proof. Let P be a connected SP-wILOG program and I an instance. The proof is analogous to the proof of Lemma 5.2. Therefore, we only show that

$$\text{if } \mathbf{f} \in \mathbf{T}_P^i(I) \text{ then there is a } C_\mathbf{f} \in co(I) \text{ such that } \mathbf{f} \in \mathbf{T}_P^i(C_\mathbf{f}). \qquad (\dagger)$$

Clearly, for $i = 1$, ($\dagger$) follows from the connectedness of P. Assume $\mathbf{f} \in \mathbf{T}_P^{i+1}(I)$. Then, there is a Skolemized rule γ in the Skolemization of P and a valuation V such that $V(pos_\gamma) \subseteq \mathbf{T}_P^i(I)$ and $V(neg_\gamma) \cap \mathbf{T}_P^i(I) = \emptyset$. By induction, for every fact $\mathbf{g} \in V(pos_\gamma)$ there is a $C_\mathbf{g} \in co(I)$ such that $\mathbf{g} \in \mathbf{T}_P^i(C_\mathbf{g})$. As γ is connected there is a $C \in co(I)$, such that $C_\mathbf{g} = C_\mathbf{g}$, $= C$ for all $\mathbf{g}, \mathbf{g'} \in V(pos_\gamma)$. So, $\mathbf{f} \in \mathbf{T}_P^{i+1}(C)$. This completes the proof. □

Lemma 5.7. *Every connected wILOG$^\neg$ program distributes over components.*

Proof. The proof is analogous to the proof of Lemma 5.3. Lemma 5.6 serves as the base case, and we rely on the domain-distinct-monotonicity for SP-wILOG programs, which is shown in [23] (albeit under the name "semimonotone"). □

Notice that, by Lemma 5.6 and Lemma 5.7, when the output of a connected SP-wILOG or wILOG$^\neg$ program is finite over I, it is finite over the components of I, and vice versa.

We are now ready to show the following theorem:

Theorem 5.8. *semicon-wILOG$^\neg$ $\subseteq \mathcal{M}_{disjoint}$.*

Proof. Let $P = P_2 \circ P_1$ such that P_1 is a connected wILOG$^\neg$ program and P_2 is a SP-wILOG program. Then, we show that $P(I) \subseteq P(I \cup J)$, for all I, J where J is domain-disjoint from I and the output of P over $(I \cup J)$ is defined.

From Lemma 5.7 it follows that

$$\begin{aligned} P_1(I \cup J) &= \bigcup_{D \in co(I \cup J)} P_1(D) \\ &= \left(\bigcup_{D \in co(I)} P_1(D) \right) \cup \left(\bigcup_{D \in co(J)} P_1(D) \right) \\ &= P_1(I) \cup P_1(J). \end{aligned}$$

So, by finiteness of $P_1(I \cup J)$: $P_1(I)$ and $P_1(J)$ are finite.

By definition, P_2 is in SP-wILOG, so the query expressed by P_2 is domain-distinct-monotone ([23]). Now, by domain-distinctness of $P_1(I \cup J) \setminus P_1(I)$ from $P_1(I)$ and finiteness of P_2 over $P_1(I \cup J)$: P_2 is finite over $P_1(I)$ and $P(I) \subseteq P(I \cup J)$. □

Next, we show the reverse direction:

Theorem 5.9. $\mathcal{M}_{disjoint} \subseteq$ *semicon-wILOG*$^\neg$.

Proof. The proof is rather tedious and can be found in Appendix A. □

Finally, we obtain the following corollary:

Corollary 5.10. *Semi-connected wILOG*$^\neg$ *computes precisely all queries in* $\mathcal{M}_{disjoint}$.

6 Nullary Relations

A nullary relation is a relation having arity zero which can only serve as a boolean flag: either the relation is empty, or it is nonempty. So far, we have avoided a discussion of nullary relations to simplify the presentation. For completeness, we discuss here how the results and proofs can be extended to incorporate nullary relations. Specifically, we treat nullary facts in a way that does not interfere with non-nullary facts. For instance, nullary facts are never domain-disjoint from any instance and distribution policies that are domain-guided always assign nullary facts to all nodes.

6.1 Domain-distinct-monotone and Domain-disjoint-monotone

A fact $\mathbf{f}$ is domain distinct from an instance I when $adom(\mathbf{f}) \setminus adom(I) \neq \emptyset$. That is, $\mathbf{f}$ contains at least one new value not yet occurring in I. Therefore, nullary facts are never domain distinct from any instance. However, a non-nullary fact can still be domain-distinct from an instance containing nullary-facts. The following Datalog program, for instance, reads a unary input relation R and a nullary input relation S and is thus domain-distinct-monotone:

$$T(x) \leftarrow R(x), \neg S(\,).$$

Indeed, let I, J be instances with J domain-distinct from I, then either $S() \in I$ implying the output to be empty on both I and $I \cup J$, or $S() \notin I$ implying that the output on I is a subset of the output on $I \cup J$, because J cannot contain $S()$ by domain-distinct monotonicity.

Regarding domain-disjoint-monotonicity, we expand the definition of domain disjointness to include nullary facts by simply saying that a nullary

fact is never domain disjoint from any instance. Notice that now the property of an instance J being domain disjoint from an instance I is no longer symmetric: for example, $\{R(a)\}$ is domain disjoint from $\{S(\,)\}$, but not vice versa.

Overall, the intuition of nullary facts is that they are present at the beginning of the computation, and the monotonicity of a query or program is only judged when we grow the input with non-nullary facts.

6.2 Transducers and Distribution Policies

First, note that the operational semantics of transducer networks should not be modified to deal with nullary input relations. Indeed, automatically, for each nullary input relation R, we expose the fact $\texttt{policy}_R(\,)$ at a node x if x is responsible for the fact $R(\,)$ under the distribution policy. We now discuss the proofs of the results.

6.2.1 Regarding domain-distinct-monotone queries

The proof of Proposition 4.5 remains unaltered. Note that the instance J considered there does not contain nullary facts by definition of domain-distinct-monotonicity. Any nullary facts contained in $I \cup J$ are actually in I, and the node x is given again I by the second transducer network $\mathbf{\Pi}_2$.

For the proof of Proposition 4.6, we should add the following rule for each nullary input relation R, to avoid reading relation `MyAdom` for nullary facts:

$$\texttt{R\_missing}(\,) \leftarrow \neg \texttt{R\_known}(\,).$$

The discussion regarding the completeness of `MyAdom` uniformly encompasses any nullary input facts.

6.2.2 Regarding domain-disjoint-monotone queries

We redefine when a distribution policy is domain-guided. Formally, we say that a distribution policy $\boldsymbol{P}$ over a schema σ and network $\mathcal{N}$ is *domain-guided* if there is a domain assignment $\boldsymbol{\alpha}$ for $\mathcal{N}$ such that *(i)* $\boldsymbol{P}(\mathbf{f}) = \mathcal{N}$ for each nullary fact $\mathbf{f} \in \mathit{facts}(\sigma)$ and *(ii)* $\boldsymbol{P}(\mathbf{f}) = \bigcup_{a \in \mathit{adom}(\mathbf{f})} \boldsymbol{\alpha}(a)$ for each non-nullary fact $\mathbf{f} \in \mathit{facts}(\sigma)$. So, all nullary input facts are always assigned to all nodes.

In the proof of Proposition 4.7, the considered instance J contains no nullary facts under the new definition of domain disjointness. Where we argue that node x is given instance I by transducer network $\mathbf{\Pi}_2$ on input $I \cup J$, we now additionally incorporate the following reasoning steps for nullary facts:

- Let $\mathbf{g} \in I$ be a nullary fact given to x by $\mathbf{\Pi}_1$ on input I. Under the new definition of domain-guided distribution policy, the constructed

distribution policy $\boldsymbol{P}_2$ always satisfies $x \in \boldsymbol{P}_2(\mathbf{g})$. So, $\mathbf{g}$ is given to x by $\boldsymbol{\Pi}_2$ on input $I \cup J$.

- Let $\mathbf{g} \in I \cup J$ be a nullary fact given to x by $\boldsymbol{\Pi}_2$ on input $I \cup J$. Because J contains no nullary facts, we concretely know $\mathbf{g} \in I$. Again, the new definition of domain-guided distribution policy implies that always $x \in \boldsymbol{P}_1(\mathbf{g})$. So, $\mathbf{g}$ is given to x by $\boldsymbol{\Pi}_1$ on input I.

The proof of Proposition 4.8 works as is for nullary facts: the new definition of domain-guided distribution policy assigns all nullary facts to all nodes, so the transducer should not be equipped with additional rules to collect all nullary facts.

6.3 Datalog Results

We first generalize the notion of component. In the presence of nullary facts, a component must always contain all the nullary facts of the instance. For example, $I = \{R(a), R(b), S(), T()\}$ has two components, namely $\{R(a), S(), T()\}$ and $\{R(b), S(), T()\}$. Further, an instance J containing only nullary facts can be a component of I only if $I = J$.

Definition 5 generalizes the syntactic restrictions on con-Datalog$^\neg$ and semicon-Datalog$^\neg$ programs to incorporate nullary relations. For this, we borrow from [17] the following terminology. We say that nullary relations of $edb(P)$ are *global* (for all components) if the nullary input facts are given to all components by definition. Similarly, we say a nullary relation of $idb(P)$ is global if all its rules, and the rules of the *idb*-relations it depends on, do not use variables. So, the term "global" means that these nullary relations will have the same contents on every component.

Definition 5. *Let P be a program in Datalog$^\neg$. Then P is a* connected stratified datalog program*, when all rules in P are connected and the only nullary-relations in the body of a rule are global. We say that P is* semi-connected *when P has a stratification such that all strata except possibly the last one are connected SP-Datalog programs with only global nullary relations.*

Particularly notice that the definition of connected-datalog allows to derive non-global nullary-relations for output purposes (i.e., relations that do not occur in the body of any rule), while the definition of semi-connected datalog forbids non-global nullary relations as a whole in all but the last stratum. Nevertheless, every connected-datalog program is a semi-connected datalog program according to these definitions, as all the none-global nullary relations can be accumulated in the last stratum.

Before arguing the correctness of Lemmas 5.2, 5.3 and Theorem 5.4, over these generalized definitions, we make the following observation: Denote by $glob(P)$ the set of global relations in $\sigma(P)$. For a connected datalog program

P, it follows that $P(I)_{|glob(P)} = P(C)_{|glob(P)}$ for every $C \in co(I)$. Indeed, every component of an instance I contains all the global relations in I; and consequently P computes the same global relations on each component.

By the above observation, Lemma 5.2 and Lemma 5.3 now generalize immediately. Particularly notice that to show "$\mathbf{f} \in \mathbf{T}_P^{i+1}(I)$ implies $\mathbf{f} \in \mathbf{T}_P^{i+1}(C)$, for some $C \in \mathbf{T}_P^{i+1}(I)$", the choice of C depends on non-nullary facts only. Indeed, by the above observation, global facts are in $\mathbf{T}_P^{i+1}(C)$ by construction of P. When $V(pos_\varphi)$ contains only nullary-facts, C is chosen arbitrarily.

In the proof for Lemma 5.3 we choose a stratification for P where every rule deriving a non-global, nullary relation is placed in the last stratum. Notice that such a stratification exists, as these relations cannot occur in the body of any rule (by definition of con-Datalog$^\neg$). Now, the proof proceeds as before.

Finally, the proof of Theorem 5.4 needs no adjustments, as it follows straightforwardly from the updated Lemmas 5.2 and 5.3; and the new definition of semi-connected datalog.

Remark 6.1. *We recall from [17] the notion of 'value-detecting' relations: A nullary relation $S^{(0)} \in idb(P)$ is called* value-detecting *when* (i) *for each non-nullary relation $R^{(k)} \in edb(P)$, program P contains a rule isomorphic to '$S() \leftarrow R(u1, ..., uk)$', using pairwise distinct variables; and,* (ii) *there are no other rules for S in P. In [17] the following result is shown:*

Theorem 6.2. *Every query in Datalog$^\neg$ that distributes over components can be expressed with connected datalog with value-detecting relations.*

Intuitively, a value-detecting relation allows to detect whether there is at least one non-nullary fact in the input; or, negated, that the input contains only nullary-facts. So, value-detecting relations are just a special type of global relations. Indeed, for an instance I, either every component of I has at least one non-nullary relation; or none of them have, indicating that only one component exists (namely, I itself). By the above intuition together with Lemma 5.3 the following result follows immediately:

Theorem 6.3. *Every query expressible with connected datalog with value-detecting relations distributes over components.*

Corollary 6.4. *A query is in Datalog$^\neg$ and distributes over components if, and only if, it can be expressed with connected datalog with value-detecting relations.*

7 Related work

The present article expands upon the conference version [16] by providing all proofs; only proof sketches were present in [16]. We also added Section 6 that explains how to incorporate nullary relations.

Declarative networking & CALM. The approach in this paper is motivated by the work on the CALM-conjecture. Hellerstein [31] formulated the CALM-principle which suggests a link between logical monotonicity and distributed consistency without the need for coordination. The latter principle is, for instance, embedded in BLOOM, a declarative language for distributed programming, for which practical program analysis techniques have been developed for detecting potential consistency anomalies [12, 13, 25]. Hellerstein [31] furthermore argues that the theory of declarative database query languages can provide a foundation for the next generation of parallel and distributed programming languages and formulates a number of related conjectures for the PODS community. Datalog has previously been considered for distributed systems, see e.g. [33, 1, 37].

Ameloot et al. [18] have introduced the framework of relational transducer networks to formalize and prove the CALM-conjecture. The formalism was then parameterized by Zinn et al. [42] to allow for specific data distribution strategies. These authors showed that the classes of coordination-free queries in the original transducer network model ($\mathcal{F}_0$), in the policy-aware transducer network model ($\mathcal{F}_1$), and in the domain-guided transducer network model ($\mathcal{F}_2$) form a strict hierarchy. In particular, they showed that the non-monotone win-move query is in $\mathcal{F}_2$. Some further work on relational transducer networks was done by Ameloot and Van den Bussche who studied decidability of confluence [19] and consistency [15]. The CRON-conjecture, which relates the causal delivery of messages to the nature of the computations that those messages participate in (like monotone versus non-monotone) is treated by Ameloot and Van den Bussche [20].

Zinn introduced in [41] the idea of domain-distinct- and domain-disjoint-monotone queries (albeit under a different name) and related these query classes to the coordination-free queries. However, by wrongly assuming that $\mathcal{M}^1_{distinct}$ and $\mathcal{M}^1_{disjoint}$ equal $\mathcal{M}_{distinct}$and $\mathcal{M}_{disjoint}$, respectively, the corresponding versions of Theorem 4.3 and Theorem 4.4 in [41] are incorrect[18]. Although the proof and statements of the results are incorrect, the proposed approach did already contain the ideas on which the proofs presented in this paper are based. In fact, it was the enthusiasm of the first three authors over the approach in [41] that in a collaborative effort with the fourth author led to the present paper. In this way, Section 4.3 of the present paper can be seen as an extension and correction of [41]. The results in Section 3, Section 4.4, and Section 5 are not discussed in [41].

The networked relational transducer model is just one paradigm for studying distributed query evaluation. The main intuition of the results and the proofs can likely be transferred to other languages for describing distributed programs. When the classes of queries would be restricted to

[18]The definitions for domain-distinct- and domain-disjoint-monotone query classes in [41] were restricted to $\mathcal{M}^1_{distinct}$ and $\mathcal{M}^1_{disjoint}$.

the fixpoint or "while" queries [4], it suffices to consider operational models that provide computing nodes with first-order logic as their local query language, combined with some state relations to which facts can be added or from which facts can be deleted (as in transducers); the intuition is that repeated transitions involving a node can simulate the iterations of an implicit while loop [18]. These features are provided by some other models studied in the literature. For example, another notable model (or language) is WebdamLog [3], which is a Datalog-variant for declarative networking. This language additionally supports *delegation*, where a node can send rules to another node instead of just facts; transmitted rules can then be locally evaluated at the addressee. In general, the distributed computations expressed by WebdamLog do not assume a fixed set of nodes, but they allow previously unseen nodes to participate. Other rule-based languages are, e.g., Netlog [30] and Dedalus [14].

The evaluation of queries has been considered in the MapReduce framework. Afrati and Ullman [10] study the evaluation of join queries and take the amount of communication, calculated as the sum of the sizes of the input to reducers, as a complexity measure. Evaluation of transitive closure and datalog queries in MapReduce has been investigated in [8, 11].

Another model, called the massively parallel (MP) model, is introduced by Koutris and Suciu [36]. Here, computation proceeds in a sequence of parallel steps, each followed by global synchronization of all servers. In this model, evaluation of conjunctive queries [36, 22] as well as skyline queries [9] have been considered.

Lastly, in Active XML [2], a distributed system is represented as a collection of XML documents in which some vertices contain calls to remote webservices. Any data returned by the calls is incorporated into the calling document.

Finite model theory. The expressiveness of (extensions of) Datalog and its relation to monotonicity have been previously investigated. We discuss some of the results relevant to the present paper. It is known that Datalog and Datalog($\neq$) are strictly included in $\mathcal{H}$ and $\mathcal{M}$, respectively (c.f., e.g., [7, 35]). As mentioned before, Afrati et al. [7] obtained that SP-Datalog $\subseteq$ $\mathcal{E}$. Feder and Vardi [29] showed that all queries in SP-Datalog that are preserved under homomorphisms can already be expressed in Datalog. That is, SP-Datalog $\cap\, \mathcal{H}$ = Datalog. Dawar and Kreutzer [26] showed that the latter result can not be extended to least fixed-point logic (LFP): LFP$\cap\mathcal{H} \not\subseteq$ Datalog. The status of SP-Datalog w.r.t. $\mathcal{E}$ is less clear. It is, for instance, not known whether SP-Datalog = $\mathcal{E} \cap$ Datalog$^\neg$. A related result here is the one by Rosen [38], who showed that FO$[\exists^*\forall\exists] \cap \mathcal{E} \subseteq$ SP-Datalog, where FO$[\exists^*\forall\exists]$ denotes first-order logic formulas of the form $\exists\bar{x}\forall y\exists z\, \psi$ where ψ is quantifier-free. We note that Tait's counterexample [40] separating $FO[\exists]$ from $\mathcal{E} \cap FO$ is definable in SP-Datalog [38]. Atserias et al. [21] study $\mathcal{E}$ in

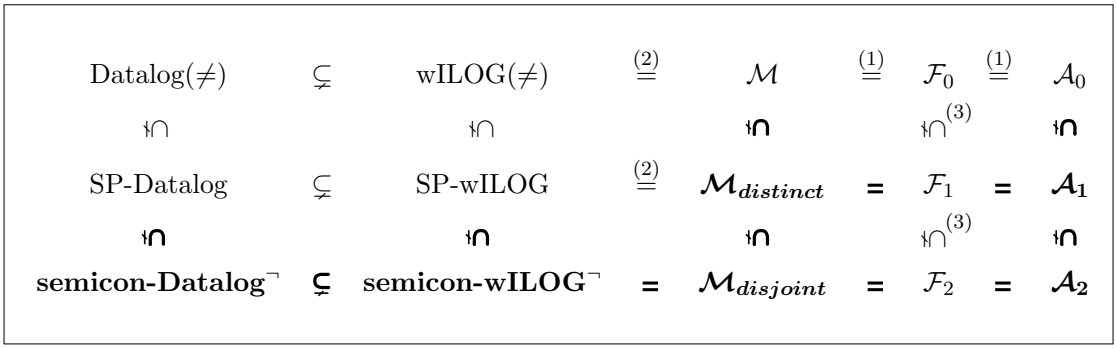

Figure 2: Main results of this work: query classes introduced and relationships obtained in this paper are shown in bold-face; related work is given with annotations. Non-bold-face relationships without annotation are part of database folklore.

relation to FO over restricted classes of structures.

8 Conclusion

Figure 2 summarizes the main findings of this paper. At the same time the figure formulates a more fine-grained answer to the CALM-conjecture which stipulates that a program has a coordination-free execution strategy if and only if the program is monotone. In particular, our results equate increasingly larger classes of coordination-free computations with increasingly weaker forms of monotonicity. We also present explicit Datalog variants for each of these classes. Furthermore, the last two columns, as already explained in Section 4.2 and Section 4.4, confirm that the notion of coordination-freeness as proposed in [18] is a sensible one. Indeed, the notion corresponds to the intended semantics in that coordination-freeness avoids global synchronization barriers through the absence of knowledge about *all* the nodes in the network. That said, we do not claim that our notion is the only possible one. Indeed, one could argue that, especially within $\mathcal{F}_1$ and $\mathcal{F}_2$, even though there is no global synchronization barrier, computing nodes are still prone to wait until complete subsets of the input data have been accumulated (cf. Section 4.3). Of course, the semantic characterizations in terms of weaker forms of monotonicity make precise that this waiting is determined by the way data is distributed. The query evaluation algorithms in the proofs in Section 4.3 are inefficient in that they require all data to be sent to all nodes, it remains to investigate how the insights obtained in this paper can lead to more practical algorithms.

In particular, as an initial step, [34] investigate more economical broadcasting strategies for full conjunctive queries without self-joins that only transmit a part of the local data necessary to evaluate the query at hand.

Another contribution of this paper is the identification of (semi-)connected Datalog variants which to the best of our knowledge have not been considered before. It is shown in [17] that the connected variant of Datalog under the well-founded semantics, making use of the well-known "doubled program" approach, remains within $\mathcal{M}_{disjoint}$. The latter implies a simpler proof of the fact that win-move is in $\mathcal{M}_{disjoint}$ (one of the main results in [42]). In addition it is shown in [17] that semi-connected Datalog under the well-founded semantics is in $\mathcal{M}_{disjoint}$.

As is to be expected, deciding whether a query belongs to one of the monotonicity classes quickly turns undecidable. Still, it would be interesting to find decidable subclasses or identify sufficient conditions as this would provide insight on the way queries can be distributedly computed. In Section 3, we introduced the bounded classes $\mathcal{M}^i_{distinct}$ and $\mathcal{M}^i_{disjoint}$ mainly because of the mismatch with [41] as explained in Section 7. It remains to investigate their relationship with distributed evaluation of queries.

Acknowledgements We thank Georg Gottlob, Thomas Eiter, and Phokion Kolaitis for answering our questions on Datalog. We also thank Phokion Kolaitis for bringing [38] to our attention.

References

[1] S. Abiteboul, Z. Abrams, S. Haar, and T. Milo. Diagnosis of asynchronous discrete event systems: Datalog to the rescue! In *PODS*, pages 358–367. ACM, 2005.

[2] S. Abiteboul, O. Benjelloun, and T. Milo. The active xml project: An overview. *The VLDB Journal*, 17(5):1019–1040, 2008.

[3] S. Abiteboul, M. Bienvenu, A. Galland, and E. Antoine. A rule-based language for Web data management. In *PODS*, pages 293–304. ACM Press, 2011.

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

[5] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. In *PODS*, pages 179–187. ACM, 1998.

[6] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. *J. Comput. Syst. Sci.*, 61(2):236–269, 2000.

[7] F. Afrati, S. S. Cosmadakis, and M. Yannakakis. On Datalog vs polynomial time. *Journal of computer and system sciences*, 51:177–196, 1995.

[8] F. N. Afrati, V. R. Borkar, M. J. Carey, N. Polyzotis, and J. D. Ullman. Map-reduce extensions and recursive queries. In *ICDE*, pages 1–8, 2011.

[9] F. N. Afrati, P. Koutris, D. Suciu, and J. D. Ullman. Parallel skyline queries. In *ICDT*, pages 274–284, 2012.

[10] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.

[11] F. N. Afrati and J. D. Ullman. Transitive closure and recursive Datalog implemented on clusters. In *EDBT*, pages 132–143, 2012.

[12] P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260, 2011.

[13] P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier. Blazes: Coordination analysis for distributed programs. In *International Conference on Data Engineering (ICDE)*, pages 52–63. IEEE, 2014.

[14] P. Alvaro, W. Marczak, et al. Dedalus: Datalog in time and space. Technical Report UCB/EECS-2009-173, EECS Department, University of California, Berkeley, Dec 2009.

[15] T. J. Ameloot. Deciding correctness with fairness for simple transducer networks. In *ICDT*, 2014.

[16] T. J. Ameloot, B. Ketsman, F. Neven, and D. Zinn. Weaker forms of monotonicity for declarative networking: a more fine-grained answer to the calm-conjecture. In *PODS*, pages 64–75. ACM, 2014.

[17] T. J. Ameloot, B. Ketsman, F. Neven, and D. Zinn. Datalog queries distributing over components. In *Proceedings of the 18th International Conference on Database Theory (ICDT)*, pages 308–323, 2015.

[18] T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. *J. ACM*, 60(2):15, 2013.

[19] T. J. Ameloot and J. Van den Bussche. Deciding eventual consistency for a simple class of relational transducer networks. In *ICDT*, pages 86–98, 2012.

[20] T. J. Ameloot and J. Van den Bussche. On the CRON conjecture. In *Datalog*, pages 44–55, 2012.

[21] A. Atserias, A. Dawar, and M. Grohe. Preservation under extensions on well-behaved finite structures. *SIAM J. Comput.*, 38(4):1364–1381, 2008.

[22] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. In *PODS*, pages 273–284, 2013.

[23] L. Cabibbo. The expressive power of stratified logic programs with value invention. *Information and Computation*, 147(1):22–56, 1998.

[24] K. Compton. Some useful preservation theorems. *Journal of Symbolic Logic*, 48:427–440, 1983.

[25] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *Symposium on Cloud Computing (SoCC)*, page 1. ACM, 2012.

[26] A. Dawar and S. Kreutzer. On Datalog vs. LFP. In *ICALP*, pages 160–171, 2008.

[27] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven Web applications. *J. Comput. Syst. Sci.*, 73(3):442–474, 2007.

[28] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. Verification of communicating data-driven Web services. In *PODS*, pages 90–99. ACM Press, 2006.

[29] T. Feder and M. Y. Vardi. Homomorphism closed vs. existential positive. In *LICS*, pages 311–320, 2003.

[30] S. Grumbach and F. Wang. Netlog, a rule-based language for distributed programming. In *Proceedings of the 12th International Conference on Practical Aspects of Declarative Languages*, pages 88–103. Springer-Verlag, 2010.

[31] J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Record*, 39(1):5–19, 2010.

[32] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *VLDB*, pages 455–468, 1990.

[33] T. Jim and D. Suciu. Dynamically distributed query evaluation. In *PODS*, pages 28–39. ACM, 2001.

[34] B. Ketsman and F. Neven. Optimal broadcasting strategies for conjunctive queries over distributed data. In *Proceedings of the 18th International Conference on Database Theory (ICDT)*, pages 291–307, 2015.

[35] P. G. Kolaitis and M. Y. Vardi. On the expressive power of Datalog: Tools and a case study. In *PODS*, pages 61–71, 1990.

[36] P. Koutris and D. Suciu. Parallel evaluation of conjunctive queries. In *PODS*, pages 223–234, 2011.

[37] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: Language, execution and optimization. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 97–108. ACM, 2006.

[38] E. Rosen. *Finite Model Theory and finite Variable Logics*. PhD thesis, University of Pennsylvania, 1995.

[39] O. Shmueli. Equivalence of Datalog queries is undecidable. *The Journal of Logic Programming*, 15(3):231 – 241, 1993.

[40] W. W. Tait. A counterexample to a conjecture of Scott and Suppes. *The journal of Symbolic Logic*, 24(1):15–16, 1959.

[41] D. Zinn. Weak forms of monotonicity and coordination-freeness. *CoRR*, abs/1202.0242, 2012.

[42] D. Zinn, T. J. Green, and B. Ludäscher. Win-move is coordination-free (sometimes). In *ICDT*, pages 99–113, 2012.

APPENDIX

A Proof of Theorem 5.9

Cabibbo [23] showed that wILOG$^\neg$ captures the class of all computable queries. In particular, he showed that for every computable query Q there is a two-stratum wILOG$^\neg$ program P for which $Q(J) = P(J)$ for every instance J. We provide a sketch of the construction, as it forms the basis for the construction that shows $\mathcal{M}_{disjoint} \subseteq$ semicon-wILOG$^\neg$.

But, we first introduce the notion of enumeration, which plays a crucial role throughout the construction. Let σ be a database schema and I an instance over σ. By $R(I)$ we denote the set of tuples representing facts in I with relation symbol R. An *enumeration of* $R(I)$ is simply a sequence of all the tuples in $R(I)$ encapsulated between square brackets, where tuples are represented as sequences of values between parentheses. By $enum_R(I)$ we denote the set of all enumerations for $R(I)$. For example, $enum_R(I) = \{[(ab)(bc)], [(bc)(ab)]\}$ for $R(I) = \{(a, b), (b, c)\}$. Assuming a total ordering over σ, by an *enumeration of* I we mean a concatenation

of $enum_R(I)$ for each $R \in \sigma$ following the assumed ordering. For example, $[(ab)(bc)][(a)]$ is an enumeration for $I = \{R(a,b), R(b,c), S(a)\}$ over $\sigma = \{R, S\}$, where R precedes S in the considered ordering over σ. By $enum(I)$ we denote the set of all enumerations for I. To highlight the difference between enumerations in $enum_R(I)$ and $enum(I)$ we sometimes refer to these as *relation-enumerations* and *instance-enumerations*, respectively. Further, we refer to "[", "]", "(", and ")" as delimiters, which are special values outside **dom**.

A.1 $\mathcal{C} \subseteq$ wILOG$^\neg$

We are now ready to sketch P. [23] Intuitively, P can be divided in three functional parts:

1. **Encoding:** A two-stratum wILOG$^\neg$ program P_{encode} that encodes the input instance as a set of input enumerations;
2. **Simulation:** A positive wILOG$^\neg$ program P_{M} that simulates Q over the generated input enumerations by means of a domain Turing machine M (as defined later), resulting in a set of output enumerations; and
3. **Decoding:** A positive wILOG$^\neg$ program P_{decode} that decodes the generated output enumerations into an output instance.

Next, we explain these parts in more detail. Let Q be a query over input schema $\sigma = \{R_1, \ldots, R_n\}$ and output schema σ'. Henceforth, we assume the natural ordering $R_1, \ldots, R_n$ on σ.

A.1.1 Encoding

Program P_{encode} can be seen as a two-stratum wILOG$^\neg$ program defined over input schema σ and output schema σ^{in}_{enum},

$$\sigma^{in}_{enum} \stackrel{\text{def}}{=} \{\texttt{ENC}^{(1)}, \texttt{ENC}^{cons(3)}, \texttt{ENC}^{nil(1)}\},$$

which generates for a given input instance I, in parallel, all enumerations in $enum(I)$. Here, the relation $\texttt{ENC}^{cons}$ represents the consecutive values of the enumerations in $enum(I)$, stored as facts $\texttt{ENC}^{cons}(s, v, t)$, where s is an id for the given fact, v is a value from $adom(I)$ or a working symbol, and t is an id referring to the next fact in the enumeration, if there is such a fact. Relation $\texttt{ENC}^{nil}$ identifies a special invented value used to denote the end of an enumeration. Furthermore, the relation ENC contains exactly those ids referring to facts in $\texttt{ENC}^{cons}$ forming the initial element in an enumeration.

Example A.1. *Recall our running example, i.e.,* $I = \{R(a,b), R(b,c), S(a)\}$, *with* $enum(I) = \{[(ab)(bc)][(a)], [(bc)(ab)][(a)]\}$. *Then,*

$$
\begin{aligned}
P_{encode}(I) = \{ & \\
& \mathit{ENC}(v_1), \mathit{ENC}^{cons}(v_1, \text{`[', } v_2), \mathit{ENC}^{cons}(v_2, \text{`(', } v_3), \mathit{ENC}^{cons}(v_3, a, v_4), \ldots, \\
& \mathit{ENC}^{cons}(v_{15}, \text{`]', } v_{nil}), \mathit{ENC}^{nil}(v_{nil}), \\
& \mathit{ENC}(v_{16}), \mathit{ENC}^{cons}(v_{16}, \text{`[', } v_{17}), \mathit{ENC}^{cons}(v_{17}, \text{`(', } v_{18}), \ldots, \\
& \mathit{ENC}^{cons}(v_{30}, \text{`]', } v_{nil})\},
\end{aligned}
$$

where each v_i *represents an invented value and thus represents an id. Notice that,* $\mathit{ENC} = \{v_1, v_{14}\}$ *as these values refer to the initial elements of the two enumerations.*

Program P_{encode} generates enumerations by performing the following tasks:

1. **Generation of Relation-Enumerations:** For every relation name $R \in \sigma$, every enumeration in $enum_R(I)$ is generated; and
2. **Concatenation of Relation-Enumerations:** The enumerations in $enum_R(I)$ are generated by concatenating every sequence of relation-enumerations $e_1 e_2 \ldots e_n$, where $e_1 \in enum_{R_1}(I), \ldots, e_n \in enum_{R_n}(I)$.

We recall the constructions for (1) and (2) in more detail below.

Generation of Relation-Enumerations Enumerations in $enum_{R_i}(I)$ are generated and stored in auxiliary relations $\texttt{ENC}_i$, $\texttt{ENC}_i^{cons}$, and $\texttt{ENC}_i^{nil}$. These have a similar meaning as their instance-enumeration equivalents $\texttt{ENC}$, $\texttt{ENC}^{cons}$, and $\texttt{ENC}^{nil}$ which we described earlier. The only difference is that $\texttt{ENC}_i$ is a ternary relation containing facts of the form $\texttt{ENC}_i(t, \text{“[”}, s)$ where t is the id of the initial element in the enumeration.

Enumerations are constructed starting from the end value, and build up by adding values in front of it. For the construction, $\texttt{ENC}_i^{cons}$ is divided in multiple relations, each with its own meaning: $\texttt{ENC}_i^{*}$ contains facts referring to a partial enumeration for R_i without opening bracket, whereas $\texttt{ENC}_i^{)}$, $\texttt{ENC}_i^{1}$, and $\texttt{ENC}_i^{2}$ contain facts referring to a partial enumeration with an incomplete tuple in front. Further, the relation $\texttt{misses}_i$ associates partial enumerations with tuples from R_i that are not yet in the enumeration.

The construction is given for a binary relation R_i, but can be easily extended for a relation with arity k. To reduce clutter, variables that occur only once are replaced by anonymous variables, which are denoted by underscores. Constants are always between double quotes:

$$
\begin{array}{lcl}
\mathtt{ENC}_i^{nil}(*) & \leftarrow & \\
\mathtt{ENC}_i^{*}(*, \text{“]”}, nil) & \leftarrow & \mathtt{ENC}_i^{nil}(nil) \\
\mathtt{misses}_i(s, x_1, x_2) & \leftarrow & \mathtt{ENC}_i^{*}(s, \text{“]”}, nil), \mathtt{ENC}_i^{nil}(nil), \mathtt{R}_i(x_1, x_2) \\
\mathtt{ENC}_i^{)}(*, \text{“)”}, s) & \leftarrow & \mathtt{ENC}_i^{*}(s, \_, \_), \mathtt{misses}_i(s, \_, \_) \\
\mathtt{ENC}_i^{2}(*, x_2, s_1) & \leftarrow & \mathtt{ENC}_i^{)}(s_1, \text{“)”}, s_0), \mathtt{ENC}_i^{*}(s_0, \_, \_), \mathtt{misses}_i(s_0, \_, x_2) \\
\mathtt{ENC}_i^{1}(*, x_1, s_2) & \leftarrow & \mathtt{ENC}_i^{2}(s_2, x_2, s_1), \mathtt{ENC}_i^{)}(s_1, \text{“)”}, s_0), \\
 & & \mathtt{ENC}_i^{*}(s_0, \_, \_), \mathtt{misses}_i(s_0, x_1, x_2) \\
\mathtt{ENC}_i^{*}(*, \text{“(”}, s_3) & \leftarrow & \mathtt{ENC}_i^{1}(s_3, x_1, s_2), \mathtt{ENC}_i^{2}(s_2, x_2, s_1), \\
 & & \mathtt{ENC}_i^{)}(s_1, \text{“)”}, s_0), \mathtt{ENC}_i^{*}(s_0, \_, \_), \mathtt{misses}_i(s_0, x_1, x_2) \\
\mathtt{misses}_i(s_4, y_1, y_2) & \leftarrow & \mathtt{ENC}_i^{*}(s_4, \text{“(”}, s_3), \mathtt{ENC}_i^{1}(s_3, x_1, s_2), \\
 & & \mathtt{ENC}_i^{2}(s_2, \_, s_1), \mathtt{ENC}_i^{)}(s_1, \text{“)”}, s_0) \\
 & & \mathtt{ENC}_i^{*}(s_0, \_, \_), \mathtt{misses}_i(s_0, y_1, y_2), x_1 \neq y_1
\end{array}
$$

$$
\begin{array}{lcl}
\mathtt{misses}_i(s_4, y_1, y_2) & \leftarrow & \mathtt{ENC}_i^{*}(s_4, \text{“(”}, s_3), \mathtt{ENC}_i^{1}(s_3, \_, s_2), \\
 & & \mathtt{ENC}_i^{2}(s_2, x_2, s_1), \mathtt{ENC}_i^{)}(s_1, \text{“)”}, s_0) \\
 & & \mathtt{ENC}_i^{*}(s_0, \_, \_), \mathtt{misses}_i(s_0, y_1, y_2), x_2 \neq y_2 \\
\mathtt{misses}_i^{proj}(s) & \leftarrow & \mathtt{misses}_i(s, \_, \_) \\
\mathtt{ENC}_i(*, \text{“[”}, s) & \leftarrow & \mathtt{ENC}_i^{*}(s, \_, \_), \neg \mathtt{misses}_i^{proj}(s)
\end{array}
$$

Notice in particular the use of negation in the last rule above to separates completeenumerations from partial enumerations for R_i, by checking that no more facts aremissing in the enumeration. This is the only place in the construction where negation is used (except for negation over equalities).

Eventually, the relation $\mathtt{ENC}_i^{cons}$ is constructed to provide a uniform interface to the generated enumerations, which can be used in the consecutive parts of the construction:

$$
\begin{array}{lcl}
\mathtt{ENC}_i^{cons}(s, v, t) & \leftarrow & \mathtt{ENC}_i(s, v, t) \\
\mathtt{ENC}_i^{cons}(s, v, t) & \leftarrow & \mathtt{ENC}_i^{)}(s, v, t) \\
\mathtt{ENC}_i^{cons}(s, v, t) & \leftarrow & \mathtt{ENC}_i^{*}(s, v, t) \\
\mathtt{ENC}_i^{cons}(s, v, t) & \leftarrow & \mathtt{ENC}_i^{1}(s, v, t) \\
\mathtt{ENC}_i^{cons}(s, v, t) & \leftarrow & \mathtt{ENC}_i^{2}(s, v, t)
\end{array}
$$

Concatenation of Relation-Enumerations Enumerations for the input instance are build by transcribing complete enumerations for the relations in σ by following the order on σ, i.e., new enumerations are created by first starting from an enumeration $e_1 \in enum_{R_1}(I)$, followed by an enumeration $e_2 \in enum_{R_2}(I)$, and so on. More specifically, the enumerations in $enum(I)$ are generated by concatenating every sequence of relation-enumerations $e_1 e_2 \ldots e_n$, where $e_1 \in enum_{R_1}(I), \ldots, e_n \in enum_{R_n}(I)$.

Below, the relation $\texttt{represents}_i$ associates the id of every fact of a newly created enumeration for I with its current position in a complete enumeration of one of its relations:

$$
\begin{array}{lcl}
\texttt{ENC}^{nil}(*) & \leftarrow & \\
\texttt{represents}(nil, nil') & \leftarrow & \texttt{ENC}^{nil}(nil), \texttt{ENC}_n^{nil}(nil') \\
\texttt{ENC}^{cons}(*, v, s_1) & \leftarrow & \texttt{ENC}_n^{cons}(\_, v, s_0), \texttt{represents}(s_1, s_0) \\
\texttt{represents}(s_1, s_0) & \leftarrow & \texttt{ENC}^{cons}(s_1, v, t_1), \texttt{ENC}_n^{cons}(s_0, v, t_0), \texttt{represents}(t_1, t_0) \\
\texttt{represents}(s_1, nil) & \leftarrow & \texttt{ENC}_{n-1}^{nil}(nil), \texttt{ENC}_n(s_0, \_, \_), \texttt{represents}(s_1, s_0) \\
\texttt{ENC}^{cons}(*, v, s_1) & \leftarrow & \texttt{ENC}_{n-1}^{cons}(l, v, s_0), \texttt{represents}(s_1, s_0) \\
\texttt{represents}(s_1, s_0) & \leftarrow & \texttt{ENC}^{cons}(s_1, v, t_1), \texttt{ENC}_{n-1}^{cons}(s_0, v, t_0), \texttt{represents}(t_1, t_0) \\
 & \ldots & \\
\texttt{ENC}(s_1) & \leftarrow & \texttt{ENC}^{cons}(s_1, \_, \_), \texttt{ENC}_1(s_0, \_, \_), \texttt{represents}(s_1, s_0)
\end{array}
$$

A.1.2 Simulation

It is a result by Hull and Su [**?**] that every computable query can be represented by an order independent domain Turing machine (TM). A *domain TM* (as introduced in [**?**]) is an adjusted TM designed for generic computations, which especially differs from traditional TMs in that it works directly over an infinite alphabet instead of a finite alphabet thereby eliminating the need to introduce (possibly complex) encodings. A domain TM M is said to be *order independent* if for every instance I, either for every enumeration of I TM M does not halt, or there is an instance J such that for every enumeration of I, M halts and its output tape contains only an enumeration of J.

Cabibbo showed that every order independent domain TM can be simulated with a positive wILOG$^\neg$ program with inequalities. In particular, for a set of enumerations, P_{M} computes M over all these enumerations by materializing the consecutive instantaneous descriptions (ID) of M, by following the transitions of M, running over each enumeration separately.

Notice that order independence ensure that, even though P_{encode} may generates several distinct enumerations for the given input instance, implying simulated runs of M over all these enumerations, either none of the runs halt, or all of them halt and result in (possibly multiple distinct) enumerations of the same output instance.

Program P_M can be seen as defined over input schema σ_{enum}^{in} and output schema σ_{enum}^{out},

$$\sigma_{enum}^{out} \stackrel{\text{def}}{=} \{\texttt{ENC}^{out(1)}, \texttt{ENC}^{cons(3)}, \texttt{ENC}^{nil(1)}\}.$$

The output instance is encoded in the same way as the input instance. Here, $\texttt{ENC}^{out}$ contains facts that represent the start of an output enumeration.

A.1.3 Decoding

Finally, a positive wILOG$^\neg$ program P_{decode} is constructed that transforms the output enumerations generated by P_{M} into an output instance by eliminating the delimiters and projecting the values into the desired output relations. Program P_{decode} expects enumerations over input schema σ^{out}_{enum} and results in an instance over output schema σ' containing only active domain values from instance I.

A.1.4 $\mathcal{M} \subseteq$ pos-wILOG and $\mathcal{M}_{distinct} \subseteq$ SP-wILOG

Cabibbo adapted the above technique to show that $\mathcal{M} \subseteq$ pos-wILOG and $\mathcal{E} \subseteq$ SP-wILOG. [23] The only change in the construction concerns the encoding phase which requires a major modification: stratified negation can no longer be used to detect whether a partial enumeration is complete.

Let $p\text{-}enum(I) \stackrel{\text{def}}{=} \bigcup_{J \subseteq I} enum(J)$, i.e., $p\text{-}enum(I)$ is the set of all partial enumerations for I. Obviously, if we only consider monotonic queries, and because I is a subset of itself, $Q(I) = \bigcup_{J \subseteq I} Q(J)$ for all I. Therefore, to show $\mathcal{M} \subseteq$ pos-wILOG it suffices to generate $p\text{-}enum(I)$ rather than $enum(I)$, which can be achieved without using negation. Indeed, we only need to replace the rule $\texttt{ENC}_i(*, \text{“[”}, s) \leftarrow \texttt{ENC}^*_i(s, \_, \_), \neg \texttt{misses}^{proj}_i(s)$ by $\texttt{ENC}_i(*, \text{“[”}, s) \leftarrow \texttt{ENC}^*_i(s, \_, \_)$. in the construction of P_{encode}.

To obtain a proof for $\mathcal{M}_{distinct} \subseteq$ SP-wILOG, a solution is to construct in parallel all enumerations of every induced subinstance of I. These subinstances can be obtained by considering for every subset D of $adom(I)$ the instance $I_{|D} = \{\mathbf{f} \in I \mid adom(\mathbf{f}) \subseteq D\}$. Indeed, as only queries Q are considered which are domain-distinct-monotone (or, equivalently, preserved under extensions):

$$Q(I) = \bigcup_{D \subseteq adom(I)} Q(I_{|D}).$$

The difficult part of the proof then is to construct a SP-wILOG program that can generate all enumerations of every I_D.

A.2 $\mathcal{M}_{disjoint} \subseteq$ semicon-wILOG$^\neg$

We now explain how to generalize the above ideas to prove that semi-connected wILOG$^\neg$ captures $\mathcal{M}_{disjoint}$. Again, the difficult part of the proof is to construct a semicon-wILOG$^\neg$ program that generates enumerations which allow to compute queries in $\mathcal{M}_{disjoint}$. Intuitively, the desired property for enumerations is to be complete with respect to the components that they contain. More formally, we say that D is a *full-component subset* of I, denoted $D \subseteq_c I$, when for all $C \in co(I)$ and $\mathbf{f} \in C$, we have that $\mathbf{f} \in D$ implies $C \subseteq D$. So, D contains either all facts of a component or none of

them. Indeed, as only queries Q are considered which are domain-disjoint-monotone and every instance is a full-component subset of itself:

$$Q(I) = \bigcup_{D \subseteq_c I} Q(D), \qquad (\dagger)$$

for all I.

Towards a semicon-wILOG$^\neg$ construction for P, we define $p\text{-}enum_{cseq}(I)$ as the set containing all (partial) enumerations of I that have the form $e = e_1 e_2 \dots e_m$ (recall that there are m relations), where $e_i = [c_i^1 c_i^2 \dots c_i^k]$ is a component wise enumeration of R_i. That is, there are distinct components $C_1, \dots, C_k \in co(I)$ such that $[c_i^j] \in enum_{R_i}(C_j)$. Further, we denote $enum_{cseq}(I)$ as the subset of $p\text{-}enum_{cseq}(I)$, where $k = |co(I)|$. We sometimes refer to enumerations in $enum_{R_i}(C_j)$ as full-component enumerations.

Now, we are ready to sketch the construction:

1. **Encoding:** A semi-connected wILOG$^\neg$ program $P^{\text{cseq}}_{\text{encode}}$ that generates $p\text{-}enum_{cseq}(I)$;

2. **Simulation:** A positive wILOG$^\neg$ program P_{M} that simulates Q over the generated input enumerations by means of a domain Turing machine M, resulting in a set of output enumerations; and

3. **Decoding:** A positive wILOG$^\neg$ program P_{decode} that decodes the generated output enumerations into an output instance.

Now, $P \stackrel{\text{def}}{=} P^{\text{cseq}}_{\text{encode}} \cup P_{\text{M}} \cup P_{\text{decode}}$ is the desired semicon-wILOG$^\neg$ program. Indeed, as for every domain-disjoint-monotonic query Q there is an order independent domain TM M that computes Q and a positive wILOG$^\neg$ program P_{M} that simulates M on enumerations; there is a semicon-wILOG$^\neg$program $P^{\text{cseq}}_{\text{encode}}$ which encodes the given input instance as a set of enumerations seeding M; and there is a positive wILOG$^\neg$ program P_{decode} that decodes the output of M.

In particular, for a domain-disjoint-monotone query Q and an order independent domain TM M that computes Q, we have,

$$\begin{aligned} P(I) &= \bigcup_{e \in p\text{-}enum_{cseq}(I)} P_{decode}(P_M(e)) \\ &= \bigcup_{D \subseteq_c I} \bigcup_{e \in enum_{cseq}(D)} P_{decode}(P_M(e)) \\ &= \bigcup_{D \subseteq_c I} Q(D) \\ &= Q(I). \end{aligned}$$

The third equality follows from order-independence of M. In particular, notice that $enum_{cseq}(D)$ is nonempty for every $D \subseteq_c I$. The fourth equality follows from $(\dagger)$.

As the expressibility of both the simulation (2) and decoding (3) follow directly from [23], we focus on the construction of $P^{\text{cseq}}_{\text{encode}}$.

Remark A.2. *Notice that (analogous to [23]) we use constants in the construction, like "[", "]", "(", and ")". However, these are only for encoding purposes and are (by construction) removed from the output by P_{decode}. Only in connected fragments we cannot use constants, as these could introduce unwanted interference between components. Therefore, we use invented values rather than constants where connectedness takes priority. For example, rather than writing:*

$$B(a, \text{“[”}) \quad \leftarrow \quad A(a),$$

we write:

$$\begin{array}{lll} \mathit{OpenBracket}(*, a) & \leftarrow & A(a). \\ B(a, b) & \leftarrow & A(a), \mathit{OpenBracket}(b, a). \end{array}$$

where `OpenBracket` *then encodes alternatives for the constant "[".*

A.2.1 Construction of $P^{\text{cseq}}_{\text{encode}}$

For ease of readability, we subdivide the construction of $P^{\text{cseq}}_{\text{encode}}$ in the following functional parts:

1. **Generation of Component-Enumerations:** For every relation symbol $R \in \sigma$ and every component C of the input, $enum_R(C)$ is generated;

2. **Generation of Partial-Enumerations:** For every relation symbol $R \in \sigma$, $p\text{-}enum_R(I)$ is generated;

3. **Selection of Full-Component Enumerations:** Partial enumerations $e_1, \ldots, e_n$, where $e_1 \in p\text{-}enum_{R_1}(I)$, $\ldots$, $e_n \in p\text{-}enum_{R_n}(I)$, are selected for which $e_1 \ldots e_n \in p\text{-}enum_{cseq}(I)$;

4. **Generation of Full-Component Enumerations:** $p\text{-}enum_{cseq}(I)$ is generated;

Next, we illustrate the above mentioned constructions in more detail.

Generation of Component-Enumerations First, a relation `C` is constructed, which represents the reflexive transitive closure of the Gaifman graph for the input, i.e., the graph having as nodes the active domain values for the input, and where two nodes a and b are connected by an edge iff $a = b$ or there is a fact having both values in the input. We use relation `C`

mainly to connect the subsequent rules. The construction is illustrated for relation R_i with arity k:

$$\begin{array}{lcl}
\mathtt{C}(x_1, x_1) & \leftarrow & \mathtt{R}_i(x_1, \ldots, x_k) \\
\mathtt{C}(x_1, x_2) & \leftarrow & \mathtt{R}_i(x_1, \ldots, x_k) \\
 & \ldots & \\
\mathtt{C}(x_{k-1}, x_k) & \leftarrow & \mathtt{R}_i(x_1, \ldots, x_k) \\
\mathtt{C}(x, y) & \leftarrow & \mathtt{C}(y, x) \\
\mathtt{C}(x, y) & \leftarrow & \mathtt{C}(x, z), \mathtt{C}(z, y)
\end{array}$$

Obviously C can be computed with a positive connected wILOG$^\neg$ program.

The following construction computes $enum_R(D)$ for every $D \in co(I)$ and every $R_i \in \sigma$. The construction is essentially the enumeration program given in Section A.1.1. But, instead of using a single, neutral, invented-value that serves as the end value for every enumeration, we create multiple end-values; one for every active domain value. The domain value then captures for which component the enumerations can be extended.

Delimiter constants are replaced by invented values obtained via the following rules:

$$\begin{array}{lcl}
\mathtt{WS}^{\mathtt{]}}(*, c) & \leftarrow & \mathtt{C}(c, c) \\
\mathtt{WS}^{\mathtt{[}}(*, c) & \leftarrow & \mathtt{C}(c, c) \\
\mathtt{WS}^{\mathtt{)}}(*, c) & \leftarrow & \mathtt{C}(c, c) \\
\mathtt{WS}^{\mathtt{(}}(*, c) & \leftarrow & \mathtt{C}(c, c)
\end{array}$$

For simplicity, the program in Figure 3 illustrates the construction where R_i is a binary relation. This can easily be extended to k-ary relations. Notice that `[`, `]`, `(`, and `)` here now variables.

Notice the use of negation in the last rule in Figure 3 to separates complete component-enumerations from partial component-enumerations for R_i, by checking that no more facts are missing in the enumeration. This is the only place in the construction where negation is used (except for negation over equalities). Particularly notice that unlike for the construction in Section A.1.1, the negation is over a connected fragment. Further, notice that the construction up to this point (excluding the last rule in Figure 3) yields a connected stratum. As the remaining construction is positive and can be viewed as the last stratum of the constructed program, the definition of semi-connected wILOG$^\neg$ allows the remaining rules to be unconnected. It is also for this reason that we can safely use constants beyond this point.

We finish the construction by replacing the invented values by their constant representations "[", "]", "(", *and* ")". Hence, obtaining the desired

$$
\begin{array}{lcl}
\mathtt{cENC}_i^{nil}(*, c) & \leftarrow & \mathtt{C}(c, c) \\
\mathtt{cENC}_i^{\mathtt{]}}(*, c, \mathtt{]}, nil) & \leftarrow & \mathtt{cENC}_i^{nil}(nil, c), \mathtt{WS}^{\mathtt{]}}(\mathtt{]}, c) \\
\mathtt{cENC}_i^{*}(s, c, v, t) & \leftarrow & \mathtt{cENC}_i^{\mathtt{]}}(s, c, v, t) \\
\mathtt{cmisses}_i(s, c, x_1, x_2) & \leftarrow & \mathtt{cENC}_i^{*}(s, c, \mathtt{]}, nil), \mathtt{cENC}_i^{nil}(nil, c), \mathtt{R}_i(x_1, x_2), \\
 & & \mathtt{C}(c, x_1), \mathtt{WS}^{\mathtt{]}}(\mathtt{]}, c). \\
\mathtt{cENC}_i^{\mathtt{)}}(*, c, \mathtt{)}, s) & \leftarrow & \mathtt{cENC}_i^{*}(s, c, \_, \_), \mathtt{cmisses}_i(s, c, \_, \_), \mathtt{WS}^{\mathtt{)}}(\mathtt{)}, c) \\
\mathtt{cENC}_i^{2}(*, c, x_2, s') & \leftarrow & \mathtt{cENC}_i^{\mathtt{)}}(s', c, \mathtt{)}, s), \mathtt{cENC}_i^{*}(s, c, \_, \_) \\
 & & \mathtt{cmisses}_i(s, c, \_, x_2), \mathtt{WS}^{\mathtt{)}}(\mathtt{)}, c) \\
\mathtt{cENC}_i^{1}(*, c, x_1, s_2) & \leftarrow & \mathtt{cENC}_i^{2}(s_2, c, x_2, s_1), \mathtt{cENC}_i^{\mathtt{)}}(s_1, c, \mathtt{)}, s_0), \\
 & & \mathtt{cENC}_i^{*}(s_0, c, \_, \_), \mathtt{cmisses}_i(s_0, c, x_1, x_2), \\
 & & \mathtt{WS}^{\mathtt{)}}(\mathtt{)}, c) \\
\mathtt{cENC}_i^{\mathtt{(}}(*, c, \mathtt{(}, s_3) & \leftarrow & \mathtt{cENC}_i^{1}(s_3, c, x_1, s_2), \mathtt{cENC}_i^{2}(s_2, c, x_2, s_1), \\
 & & \mathtt{cENC}_i^{\mathtt{)}}(s_1, c, \mathtt{)}, s_0), \mathtt{cENC}_i^{*}(s_0, c, \_, \_), \mathtt{WS}^{\mathtt{(}}(\mathtt{(}, c), \\
 & & \mathtt{WS}^{\mathtt{)}}(\mathtt{)}, c), \mathtt{cmisses}_i(s_0, c, x_1, x_2) \\
\mathtt{cENC}_i^{*}(s, c, v, t) & \leftarrow & \mathtt{cENC}_i^{\mathtt{(}}(s, c, v, t) \\
\mathtt{cmisses}_i(s_4, c, y_1, y_2) & \leftarrow & \mathtt{cENC}_i^{*}(s_4, c, \mathtt{(}, s_3), \mathtt{cENC}_i^{1}(s_3, c, x_1, s_2), \\
 & & \mathtt{cENC}_i^{2}(s_2, c, \_, s_1), \mathtt{cENC}_i^{\mathtt{)}}(s_1, c, \mathtt{)}, s_0) \\
 & & \mathtt{cENC}_i^{*}(s_0, c, \_, \_), \mathtt{cmisses}_i(s_0, c, y_1, y_2), \\
 & & \mathtt{WS}^{\mathtt{(}}(\mathtt{(}, c), \mathtt{WS}^{\mathtt{)}}(\mathtt{)}, c), x_1 \neq y_1 \\
\mathtt{cmisses}_i(s_4, c, y_1, y_2) & \leftarrow & \mathtt{cENC}_i^{*}(s_4, c, \mathtt{(}, s_3), \mathtt{cENC}_i^{1}(s_3, c, \_, s_2), \\
 & & \mathtt{cENC}_i^{2}(s_2, c, x_2, s_1), \mathtt{cENC}_i^{\mathtt{)}}(s_1, c, \mathtt{)}, s_0) \\
 & & \mathtt{cENC}_i^{*}(s_0, c, \_, \_), \mathtt{cmisses}_i(s_0, c, y_1, y_2), \\
 & & \mathtt{WS}^{\mathtt{(}}(\mathtt{(}, c), \mathtt{WS}^{\mathtt{)}}(\mathtt{)}, c), x_2 \neq y_2 \\
\mathtt{cmisses}_i^{proj}(s, c) & \leftarrow & \mathtt{cmisses}_i(s, c, \_, \_) \\
\mathtt{cENC}_i^{tmp}(*, \mathtt{[}, s) & \leftarrow & \mathtt{cENC}_i^{*}(s, c, \_, \_), \neg \mathtt{cmisses}_i^{proj}(s, c), \mathtt{WS}^{\mathtt{[}}(\mathtt{[}, c)
\end{array}
$$

Figure 3: Construction of component-enumerations for relation R_i.

component-enumerations, which are now encoded in the usual way:

$$
\begin{array}{lcl}
\texttt{cENC}_i(s, \text{“[”}, t) & \leftarrow & \texttt{cENC}_i^{tmp}(s, \_, t) \\
\texttt{cENC}_i^{cons}(s, v, t) & \leftarrow & \texttt{cENC}_i(s, v, t) \\
\texttt{cENC}_i^{cons}(s, \text{“]”}, nil) & \leftarrow & \texttt{cENC}_i^{]}(s, c, \_, \_), \texttt{cENC}_i^{nil}(nil, c) \\
\texttt{cENC}_i^{cons}(s, \text{“)”}, t) & \leftarrow & \texttt{cENC}_i^{)}(s, \_, \_, t) \\
\texttt{cENC}_i^{cons}(s, \text{“(”}, t) & \leftarrow & \texttt{cENC}_i^{(}(s, \_, \_, t) \\
\texttt{cENC}_i^{cons}(s, v, t) & \leftarrow & \texttt{cENC}_i^{1}(s, \_, v, t) \\
\texttt{cENC}_i^{cons}(s, v, t) & \leftarrow & \texttt{cENC}_i^{2}(s, \_, v, t)
\end{array}
$$

Generation of Partial-Enumerations For the generation of partial enumerations for I, i.e. $p\text{-}enum(I)$, we recall the construction in Section A.1.1, where $\texttt{ENC}_i(*, \text{“[”}, s) \leftarrow \texttt{ENC}_i^*(s, \_, \_), \neg\texttt{misses}_i^{proj}(s)$ is replaced by $\texttt{ENC}_i(*, \text{“[”}, s) \leftarrow \texttt{ENC}_i^*(s, \_, \_)$.

Selection of Full-component Enumerations We construct a program that recursively checks for every combination of the partial enumerations (from Section A.2.1) (one for each relation in $edb(Q)$) if they match a concatenation of full-component enumerations (from Section A.2.1) for the same components. This way we are able to collect enumerations $e_1, \ldots, e_n$, for which $e = e_1 e_2 \ldots e_n \in p\text{-}enum_{cseq}(I)$, where $n = |\sigma|$.

In the construction, the relation $\texttt{COMP}$ stores facts $\texttt{COMP}(e_1, \ldots, e_n, c_1, \ldots, c_n)$, where e_i refers to a position in a partial-enumeration for relation R_i, and c_i refers to a position in a complete-component enumeration for relation R_i, such that the tail of e_i has a prefix of values that matches exactly with c_i (except maybe for the closing bracket). Relation $\texttt{COMP}^{nil}$ denotes the initialisation for $\texttt{COMP}$, where every reference is to an end-value. Relation $\texttt{COMP}^*$ is the subset of $\texttt{COMP}$ containing only those facts for which the component-enumerations $c_1, \ldots, c_n$ are complete, i.e., they have the “[” working symbol on their first position. Notice that $\texttt{COMP}^*$ thus contains only facts, where $e_1, \ldots, e_n$ refer to full-component enumerations. Finally, $\texttt{COMP}^f$ is the subset of $\texttt{COMP}^*$ where every partial-enumerations $e_1, \ldots, e_n$ is also complete, and the component-enumerations are projected out:

$$
\begin{array}{lcl}
\texttt{COMP}^{nil}(nil_1, \ldots, nil_n, nil'_1, \ldots, nil'_n) & \leftarrow & \texttt{ENC}_1^{nil}(nil_1), \ldots, \texttt{ENC}_n^{nil}(nil_n), \\
 & & \texttt{cENC}_1^{nil}(nil'_1, c), \ldots, \texttt{cENC}_n^{nil}(nil'_n, c), \\
\texttt{COMP}(e_1, \ldots, e_n, c_1, \ldots, c_n) & \leftarrow & \texttt{COMP}^{nil}(e_1, \ldots, e_n, c_1 \ldots, c_n)
\end{array}
$$

Add for every $i \in [n]$ the following rule:

$$
\begin{array}{lcl}
\texttt{COMP}(e_1, \ldots, e_n, c_1, \ldots, c_n) & \leftarrow & \texttt{COMP}(e_1, \ldots, e_{i-i}, e'_i, e_{i+1}, \ldots, e_n, c_1, \ldots, c_{i-1}, c'_i, c_{i+1}, \ldots, c_n), \\
 & & \texttt{ENC}_i^{cons}(e_i, v, e'_i), \texttt{cENC}_i^{cons}(c_i, v, c'_i)
\end{array}
$$

Further,

$$
\begin{array}{lll}
\texttt{COMP}^*(e_1,\ldots,e_n) & \leftarrow & \texttt{COMP}(e_1,\ldots,e_n,c_1,\ldots,c_n),\\
 & & \texttt{cENC}_1(\_,\text{“[”},c_1),\ldots,\texttt{cENC}_n(\_,\text{“[”},c_n)\\
\texttt{COMP}(e_1,\ldots,e_n,c_1,\ldots,c_n) & \leftarrow & \texttt{COMP}^*(e_1,\ldots,e_n),\\
 & & \texttt{ENC}_1^{nil}(nil_1,d),\ldots,\texttt{ENC}_n^{nil}(nil_n,d),\\
 & & \texttt{cENC}_1^{cons}(c_1,\text{“]”},nil_1),\ldots,\texttt{cENC}_n^{cons}(c_n,\text{“]”},nil_n)\\
\texttt{COMP}^f(e_1,\ldots,e_n) & \leftarrow & \texttt{COMP}^*(e_1^{'},\ldots,e_n^{'})\\
 & & \texttt{ENC}_1(e_1,\_,e_1^{'}),\ldots,\texttt{ENC}_n(e_n,\_,e_n^{'})
\end{array}
$$

Generation of Full-Component Enumerations The construction is based on the concatenation program in Section A.1.1. However, instead of relating every part of the partial concatenation only with the current in-process enumeration sequentially, we keep track of all the partial-enumerations involved in the concatenation. Then, at the end, we use the relation $\texttt{COMP}^f$ to determine which of the concatenations of relation specific enumerations are in fact enumerations of full-component subsets of the input, resulting in the desired enumerations from $p\text{-}enum_{cseq}(I)$:

$$
\begin{array}{lll}
\texttt{ENC}^{nil}(*) & \leftarrow & \\
\texttt{represents}(nil,nil_1,\ldots,nil_n) & \leftarrow & \texttt{ENC}^{nil}(nil),\texttt{ENC}_1^{nil}(nil_1),\ldots,\texttt{ENC}_n^{nil}(nil_n)\\
\texttt{ENC}^{cons}(*,v,s) & \leftarrow & \texttt{ENC}_n^{cons}(\_,v,s_n),\texttt{represents}(s,\_,\ldots,\_,s_n)\\
\texttt{represents}(s,s_1,\ldots,s_n) & \leftarrow & \texttt{ENC}^{cons}(s,v,t),\texttt{ENC}_n^{cons}(s_n,v,t_n),\\
 & & \texttt{represents}(t,s_1,\ldots,s_{n-1},t_n)\\
\texttt{represents}(s,s_1,\ldots,s_{n-2},nil,s_n) & \leftarrow & \texttt{ENC}_{n-1}^{nil}(nil),\texttt{ENC}_n(s_n,v,t),\texttt{represents}(s,s_1,\ldots,s_n)\\
\texttt{ENC}^{cons}(*,v,s) & \leftarrow & \texttt{ENC}_{n-1}^{cons}(l,v,s_{n-1}),\texttt{represents}(s,s_1,\ldots,s_n),\\
 & & \texttt{ENC}_n(s_n,\_,\_)\\
\texttt{represents}(s,s_1,\ldots,s_n) & \leftarrow & \texttt{ENC}^{cons}(s,v,t),\texttt{ENC}_{n-1}^{cons}(s_{n-1},v,t_{n-1}),\\
 & & \texttt{represents}(t,s_1,\ldots,s_{n-2},t_{n-1},s_n),\\
 & & \texttt{ENC}_n(s_n,\_,\_)\\
 & & \ldots\\
\texttt{ENC}(s) & \leftarrow & \texttt{ENC}^{cons}(s,v,t),\texttt{represents}(s,s_1,\ldots,s_n),\\
 & & \texttt{COMP}^f(s_1,\ldots,s_n)
\end{array}
$$