An Algebra for Pomsets *

Stéphane Grumbach
I.N.R.I.A.
Rocquencourt BP 105
78153 Le Chesnay, France
Stephane.Grumbach@inria.fr

Tova Milo
Department of Computer Science
Tel Aviv University
Tel Aviv 69978, Israel
milo@math.tau.ac.il

Abstract

We study languages for manipulating *partially ordered* structures with *duplicates* (e.g. trees, lists). As a general framework, we consider the *pomset* (partially ordered multiset) datatype. We introduce an algebra for *pomsets*, which generalizes traditional algebras for (nested) sets, bags and lists. This paper is motivated by the study of the impact of different language primitives on the expressive power. We show that the use of partially ordered types increases the expressive power significantly. Surprisingly, it turns out that the algebra when restricted to both unordered (bags) and totally ordered (lists) intermediate types, yields the same expressive power as fixpoint logic with counting on relational databases. It therefore constitutes a rather robust class of relational queries. On the other hand, we obtain a characterization of PTIME queries on lists by considering only totally ordered types.

*Work supported in part by the Esprit Project BRA FIDE 2. Work was done while the second author was visiting University of Toronto, and supported by the Institute for Robotics and Intelligent Systems. An extended abstract appeared in the proceedings of the 5th International Conference on Database Theory, Prague 1995

1 Introduction

In the standard approach to database modeling two restrictions are imposed on database relations. (i) The elements are assumed to be un-ordered, and (ii) no duplicates are allowed. The impact of having order on database elements has been studied extensively, and was shown to be significant for the expressive power of query languages. In particular it was shown that fixpoint logic expresses all PTIME functions on ordered inputs [Imm86, Var82]. The impact of allowing duplicates in relations was studied in [GM93, GMK93, LW94]. It was also shown to have a strong effect on the expressive power of query languages, and in particular to result in an increased expressive power. In this paper, we study the interaction between order and duplicates, and the effect that the combined features have on the expressive power of query languages.

Results on the impact of order on the expressive power of languages focus on ***ordered domains***. There is another approach to order, namely the order related to the data types used in the query language, that we call the ***internal order***. Observe that many data types allow both multiple occurrences of the same element, and (partial) order on the occurrences (e.g. ***lists, trees***). We refer to such data types as ***ordered types***.

Previous research considered ***ordered domains*** and ***internal order*** as orthogonal issues. For example, the research on the expressive power of languages focused on languages for manipulating inputs of unordered types like sets and bags, and the intermediate data types used in the computation were also unordered [GM93, GMK93, Imm86, Var82]. This was the case even when the domain of the input was totally ordered. On the other hand, ordered types like lists or trees were mainly used when manipulating inputs of type list or tree [Ric92, GW92, SZL93, CRSG94].

Supporting ordered types in the data model is essential in databases for numerous advanced applications such as CAD/CAM, Software Engineering, text retrieval [SZL93, ACM93], statistical and multidimensional data [Ag 96], etc. The combined use of duplicates with partial order is natural in pratical data. In CAD for instance, compound objects can be designed from a collection of (distinct or similar) parts which should be assembled in some order. The design can be represented by partially ordered collections with repetitions for parts that occur in various positions. Partially ordered duplicates are also common in geographical information systems [LT92], where geometric components have multiple occurrences, and these occurrences are generally partially ordered. Different themes (such as ground occupancy, roads, etc.) are represented on different maps that share a lot of the geometry, resulting therefore in a duplication of the geometric cells of the map. On the other hand, various partial orders hold on the objects. For instance a road is a list of line segments. There are numerous examples of data with partially ordered duplicates, that would adequatly be reprensented with (partially) ordered types.

The paper focuses on algebraic languages for manipulating (partially) ordered types. We consider the ***pomset*** (partially ordered multiset) datatype, introduced by Pratt [Pra84] to model parallel processes, and present an algebra for ***pomsets***. The pomset type generalizes sets, bags, lists, trees, and other ordered types, and therefore provides a uniform representation for all these types. The algebra is an extension of the (nested) relational algebra [AB87]. Our aim is to investigate how the combined use of duplicates and (partial) order affects the expressive power of the languages, and increases the complexity.

Surprisingly, it turns out that the additional expressive power gained by using ordered types with duplicates is similar to that gained by using two well studied primitives — ***fixpoint*** and ***counting***. The use of duplicates introduces counting capabilities, and order enables the simulation of fixpoint. We show that the extension of the relational algebra with un-ordered types with duplicates (like bags) and totally ordered types (like lists) yields exactly the same expressive power as FO+FP+C [GO93].

To complete previous works that considered the effect of totally ordered domains on the expressive power of the language, we study here the effect of total internal order. We show that the algebra expresses all PTIME queries on lists. Note that another language for list manipulation, expressing PTIME, was presented in [CRSG94]. This language, however, required the use of nested structures. In contrast, our result shows that all PTIME queries can be expressed even without nesting.

In addition to order and duplicates, there is another parameter – ***nesting*** – that affects the expressiveness and complexity of query languages. Nesting of structures has been extensively studied in the context of unordered types like sets and bags. We study the combined effect of order, duplicates, and nesting. It was shown in [GM93] that the presence of duplicates significantly affect the expressive power of the nested algebra. In particular, two levels of nesting were proved to be sufficient for expressing all elementary functions. We show that duplicates with order are even more powerful, and one level of nesting is sufficient for expressing all elementary functions.

The paper is organized as follows. In the next section, we review some preliminaries and define the type ***pomset***. The algebra for pomsets, $\mathcal{P}$om-$\mathcal{A}$lg, is introduced in Section 3. In Section 4, we study properties of the algebra, and in Sections 5 and 6, we characterize its complexity and expressive power. Finally, in Section 7, we study the power of the nested algebra.

2 Queries on Complex Types

In this section, we present the basic framework of databases and queries over data with complex types, and recall the data complexity measures for queries. In the relational model of data, databases are collections of relations, i.e. sets of tuples. The data items, the tuples, are ***unordered***, and the relations contain no ***duplicates***, and cannot be ***nested***. Relaxing these assumptions leads to numerous distinct data types, such as the complex objects (nested sets) [Jac82, AB87, KV84, KRS85, AG91], the bags (sets with duplicates) [BK90, Mum90, Alb91, BS91, GM93, LW94], the lists (internal order), the ordered sets, and the pomsets (partially ordered multisets) [Pra84].

2.1 Partially Ordered Multisets - Pomsets

The pomset type generalizes sets, bags, lists, trees, and other ordered types, and therefore provides a uniform representation for all these types. Intuitively, a pomset can be viewed as a string with a partial order instead of a total order. The following formal definition (essentially the one of Pratt [Pra84]) is based on labeled partial orders.

Definition 2.1 *A partial order is a pair $(V, <)$ where $<$ is an irreflexive transitive binary relation on the vertex set V. A labeled partial order (lpo) is a structure $(V, \Sigma, \mu, <)$, where $(V, <)$ is a partial order, and $\mu : V \to \Sigma$ labels the vertices of V with elements of the set Σ.*

Two lpo's $(V, \Sigma, \mu, <)$ and $(V', \Sigma, \mu', <')$ over the same set of labels Σ are isomorphic if there exists a bijection $\tau : V \to V'$ such that for all $u, v \in V$, $\mu(u) = \mu'(\tau(u))$, and $u < v$ iff $\tau(u) <' \tau(v)$.

A pomset $\langle V, \Sigma, \mu, < \rangle$ is the isomorphism class of an lpo $(V, \Sigma, \mu, <)$. A pomset $\langle V, \Sigma, \mu, < \rangle$ is finite if V is finite.

Two pomsets $\langle V, \Sigma, \mu, < \rangle$ and $\langle V', \Sigma', \mu', <' \rangle$ are isomorphic if there exist bijections $\tau : V \to V'$ and $\nu : \Sigma \to \Sigma'$, such that for all $u, v \in V$ and for all $a \in \Sigma$, $\mu(u) = a$ iff $\mu'(\tau(u)) = \nu(a)$, and $u < v$ iff $\tau(u) <' \tau(v)$. We will denote by $\tau - \mu$ the pomset isomorphism.

Remarks : Pomsets are only defined up to isomorphism to hide the identities of the elements in V, so that only the cardinality of V counts, leaving Σ as the only important set underlying a pomset [Pra84]. This also enables us to assume for the sake of simplicity, that, in any two pomsets, the corresponding sets of vertices are disjoint. This hypothesis is important for algebraic operations.
Note the difference between lpo's, denoted with "( )", and pomsets, denoted with "$\langle\ \rangle$", and between isomorphisms on lpo's over the same set of labels, and isomorphisms on pomsets. The lpo's are only used to define pomsets, and in the following we shall only consider the latter.

Standard types such as sets, bags, partially ordered sets (pos), and lists, are subtypes of the pomset type, and are obtained by restricting the mapping μ and the kind of order in the pomset. (See figure). For example, sets require μ to be a one-one mapping, and no order, bags may have arbitrary μ, but no order, and lists require total order.

ORDER \ μ	one-one	many-one
EMPTY	set	bag
PARTIAL	pos	pomset
TOTAL	ordered set	list

If Σ is a collection of objects of ***type*** T, we denote the type of the pomset $\langle V, \Sigma, \mu, < \rangle$ by $\{T\}$. For a type T, the ***domain*** of $\{T\}$, is the set of all finite pomsets $\langle V, \Sigma, \mu, < \rangle$ where Σ is a collection of objects of type T.

From the implementation view point, a pomset $p = \langle V, \Sigma, \mu, < \rangle$ can be represented by a directed acyclic graph $G = (V, E)$ where each node $v \in V$ has a label in Σ attached to it, (different nodes may have the same label attached to them), and where an edge $(v_1, v_2) \in E$ iff $v_1 < v_2$.[1]

[1]A compact representation can be obtained by considering a transitively reduced [Moh89] version of the graph. i.e. by reducing the number of edges in the graph and removing edges (v_1, v_2) such that the graph

In the graph G, we consider the connected component corresponding to sets of vertices $V' \subseteq V$ that are related to each other by the symmetric and transitive closure of $<$ and not related to other vertices in $V - V'$. We overload the term connected component, and call the (sub)pomset $p' = \langle V', \Sigma, \mu \downarrow_{V'}, <\downarrow_{V' \times V'} \rangle$ corresponding to a connected component V' of the graph G, a ***connected component of the pomset*** p. A connected component p' of the pomset p may have several isomorphic (with respect to pomset isomorphism) occurrences in p. We say that a connected component ***n-belongs*** to a pomset if it has exactly n isomorphic occurrences.

We say that a pomset p is ***linear*** if $<$ defines a total order on V (lists and stacks can be represented by linear pomsets). We say that p is ***pseudo-linear*** if $<$ defines a total order on each connected component of p.

2.2 Databases and Queries

A ***pomset database*** is a set of named pomsets. (Following the relational model conventions, we shall sometimes refer to these pomsets as database relations.) A ***pomset schema*** is an expression $p : T$, where p is a pomset name, and T is a pomset type. An ***instance*** of $p : T$ is a pomset of type T. A ***database schema*** DB is a finite set of pomset schemas with distinct pomset names. An ***instance*** of DB is a mapping associating a pomset to every pomset schema in DB.

We next consider two practical examples. First, assume that U is an atomic type whose domain is an infinite set of constants. Consider the simple pomset type $\{U\}$. A database storing all the components of a complex manufactured product, can be defined as an instance of type $\{U\}$, where constants in U denote atomic components. It is rather natural to use a pomset to model compound objects, since their realization is based on sequential and parallel processes, for which the pomsets have been precisely introduced.

Let us now consider a more complex example from spatial information systems [LT92] with data for resource managment and transportation planing. Roads for instance can be represented by a type $\{[\textit{roadname,segment}]\}$, where ***roadname*** and ***segment*** are strings. This pomset is partially ordered. Each road (identified by its name) is defined as a sequence of segments. There are no duplicate. Consider now another relation for water pipes. Here there are no names, but the size of the pipe is associated with each segment. Water pipes can be represented by a type $\{[\textit{segment,size}]\}$, where ***segment*** is a string and ***size*** a rational number. In this later relation, there are duplicates (corresponding to pipes of the same size in the same location, but no partial order. The previous relations can be modified in the following ways. If we are interested in roads independently of their names, then the projection of the road relation on the segment attribute leads a relation with duplicates. For the pipe relation, we can be interested in a partial order on the segments, given for instance by the declivity, important for sewage pipes for instance. It should be clear that in this range of application, both duplicates and partial order play a fundamental role. In fact more generally transportation planing and facilities managment offer strong similarities

---

contains some other directed path from v_1 to v_2. In this representation, $v_1 < v_2$ iff there is a path from v_1 to v_2.

with sequential and parallel processes.

Queries on pomset databases are defined by extending the classical definition of [CH80] for relational queries. A query is a mapping from an input schema DB $= \{P_1, ..., P_n\}$ to an output schema S $= \{P_0\}$ with a single pomset, mapping instances of the input schema to instances of the output schema. Queries must be computable and generic, i.e. insensitive to isomorphisms on the databases (based on pomset isomorphisms). A query Q is ***generic*** if for each pomset isomorphism $\tau - \mu$ and each pomset P, $Q(\tau - \mu(P)) = \tau - \mu(Q(P))$.

We consider the ***data complexity*** of queries, i.e. the complexity of the evaluation of a query in terms of the size of the input databases. The size is defined with respect to a standard encoding [AHV94, Joh90]. The standard encoding of a pomset p is based on some labeled partial order $(V, \Sigma, \mu, <)$, and is similar to that of the binary relations representing μ and $<$. Note that in this encoding, each object in Σ, is repeated as many times as the vertices it labels in V. The size is therefore proportional to $(|\mu| + |<|)\log(|\mu| + |<|)$. This encoding may seem inefficient since pomsets can sometimes be encoded more compactly (for example, un-ordered pomsets like bags can be encoded by simply attaching to elements in Σ the number of vertices they label). Nevertheless, this encoding fits with the situation in real database systems, where duplicates are explicitly stored, sometimes precisely to avoid the cost of duplicate elimination.

Complexity classes of queries are defined straightforwardly by extending the definition for relational queries. We use as complexity measures the time and space used by a Turing machine to produce a standard encoding of the output database, starting from a standard encoding of the input database. For each Turing complexity class $\mathcal{C}$ there is a corresponding complexity class of queries, which, for simplicity, we also denote by $\mathcal{C}$.

It has been shown that order had a strong impact on the expressive power of query languages. In particular it was shown that fixpoint logic [GS86] captures exactly the class of queries computable in polynomial time over ordered relations [Imm86, Var82]. This is know to be false over unordered relations, since there are very basic queries that are not expressible in fixpoint logic. The order plays a fundamental role in this characterization. In [Gur83], recursive functions were used to capture polynomial time queries over bounded integers. In the next section, we define an algebra for partially ordered multisets. Interestingly, as shown in Section 5, when total order is available, it expresses exactly the class of polynomial time queries.

3 An Algebra for Pomsets

While for un-ordered data types like sets and bags, there is an accepted standard set of algebraic operations, this is not the case for ordered types, and different algebras for manipulating lists and trees exist in the literature [Ric92, GW92, SZL93]. Some of these languages share, however, a common property — to be an extensions of the relational algebra, and when restricted to non ordered structures with no duplicates, most of them yield the relational algebra. Following these lines, we present below a uniform algebra for (partially) ordered structures with duplicates, that is a strict extension of the relational algebra. The algebra is designed so that by restricting the order or the number of duplicates, one gets an algebra for manipulating specific kinds of ordered types. For example, when restricted to unordered structures with no duplicates it is the traditional relational algebra. When duplicates are allowed, it is the bag algebra [GM93]. When restricted to total orders, it is an algebra for lists.

To make the results of this paper general and applicable to previously presented languages, we consider a restricted set of operators — the relational algebra operators (adjusted to this context), and a small set of additional operators that are essential for exploiting the order in ordered types.

We first present a simple version of the algebra, that manipulates pomsets of tuples of atomic elements. (Nested pomsets, and the extensions in the algebra needed to enable manipulation of nested structures, are considered in Section 7). We start by presenting the algebraic operators. Next we study their properties, and in particular consider the dependencies of the operations.

Let U be the domain of atomic elements. The algebra contains a ***Tupling*** operator $[o_1, \ldots, o_k]$ that builds tuples of type U^k, and ***attribute projection x.i*** that extracts the i^{th} attribute of a tuple. The empty tupme (i.e. tuple of arity zero) is denoted by $[\,]$. Singleton pomsets are constructed using the ***pomsetting*** operator $\{o\}$, that builds a pomset of the form $\langle \{v\}, \{o\}, (v, o), \emptyset \rangle$. (Note that since pomsets are defined up to isomorphism on V, the actual value of v is irrelevant). We shall use below the term ***object*** to denote a vertex with its associated label. If no label is given then $\{\}$ creates the empty pomset $\langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$ denoted below by $\{\}$, or simply $\emptyset$.

There is a function testing the emptyness[2] of a pomset, denoted $=_\emptyset$, defined by $=_\emptyset(p) = \{[\,]\}$ if p is the empty pomset, and $=_\emptyset(p) = \{\}$, otherwise.

To construct more complex pomsets we use two classical operations on pomsets which have been considered in the literature [Pra84]. The ***additive union*** (also called ***shuffle*** in [Pra84]), and ***concatenation***. Let $p_1 = \langle V_1, \Sigma, \mu_1, <_1 \rangle$, $p_2 = \langle V_2, \Sigma, \mu_2, <_2 \rangle$ be two pomsets of type $\{T\}$. Assume without loss of generality that V_1 and V_2 are disjoint.

- ***Additive union,*** $\uplus$: $p_1 \uplus p_2$ is a pomset containing all the objects in p_1 and p_2, and preserving the order of those objects. $p_1 \uplus p_2 = \langle V_1 \cup V_2, \Sigma, \mu_1 \cup \mu_2, <_1 \cup <_2 \rangle$.

- ***Concatenation,*** $\oplus$: $p_1 \oplus p_2$ is a pomset containing all the objects in p_1 and p_2, preserving the order of the objects, and additionally making all the objects in p_1 smaller than those of p_2. $p_1 \oplus p_2 = \langle V_1 \cup V_2, \Sigma, \mu_1 \cup \mu_2, <_1 \cup <_2 \cup (V_1 \times V_2) \rangle$.

The operations $\uplus$ and $\oplus$ are associative and $\uplus$ is commutative. It turns out that these two operations are not sufficient for building all possible pomsets [Pra84]. This can be done by adding more operations to the language, as further discussed below.

Relational and bag operators

The first group of operations that we add to the algebra are the standard union, intersection,

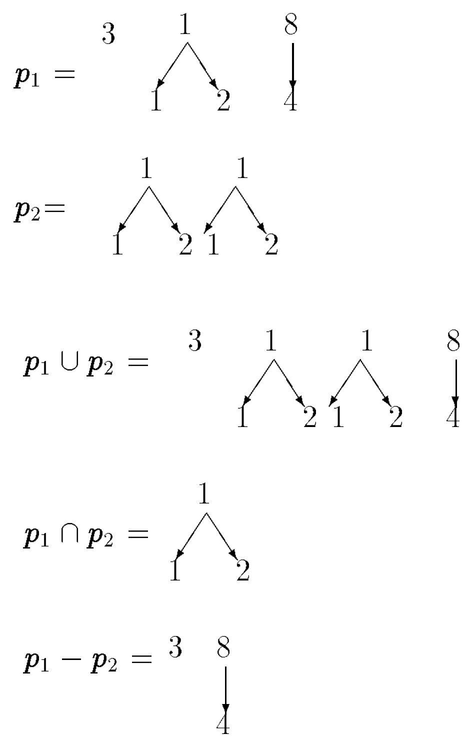
[2] The equality of two pomsets is not tractable (cf. the graph isomorphism problem [GJ79]) and so is not a desirable operation.

and subtraction. These relational operations were extended in [GM93] to handle bags. We now further generalize these operations for pomsets. In bags, all the occurrences of objects with the same label are isomorphic, and thus the operations are defined in terms of the number of occurrences of each label. The generalization to pomsets is based on the observation that in pomsets, objects with the same label may differ due to the relative order of objects. This relative order is captured by the connected components of the pomset. Thus, instead of considering the number of occurrences of individual labels, we define the operators with respect to the number of occurrences of the connected components.

Let $p = \langle V, \Sigma, \mu, < \rangle$, $p_1 = \langle V_1, \Sigma, \mu_1, <_1 \rangle$, $p_2 = \langle V_2, \Sigma, \mu_2, <_2 \rangle$ be three pomsets of type $\{T\}$, where $T = [U, ..., U]$ (U repeated k times). Assume that V_1 and V_2 are disjoint. Let $p' = \langle V', \Sigma, \mu\downarrow_{V'}, <\downarrow_{V' \times V'} \rangle$ be some connected component of p, p_1 or p_2.

- *Union,* $\bigcup$: $p_1 \cup p_2$ is a pomset, such that a connected component p' n-belongs to $p_1 \cup p_2$ iff p' r-belongs to p_1 and q-belongs to p_2 and $n = max(r, q)$.
- *Intersection,* $\bigcap$: $p_1 \cap p_2$ is a pomset, such that a connected component p' n-belongs to $p_1 \cap p_2$ iff p' r-belongs to p_1 and q-belongs to p_2 and $n = min(r, q)$.
- *Subtraction,* $-$: $p_1 - p_2$ is a pomset, such that a connected component p' n-belongs to $p_1 - p_2$ iff p' r-belongs to p_1 and q-belongs to p_2 and $n = max(0, r - q)$.

To illustrate the above definitions, consider the two pomsets p_1 and p_2 whose graph representation is described on the figure. The identifier of nodes are omitted in the figure since the result of the operations are independent of their values. The numbers in the figure represent the labels of the nodes, and the arrows represent the relative order. The graph representation of $p_1 \cup p_2$, $p_1 \cap p_2$, and $p_1 - p_2$ are illustrated.



Another group of relational operations that were extended in [GM93] to handle bags are projection (and its more general version MAP), selection, and Cartesian product. We now further generalize these operations for pomsets.

We first introduce a notation for functions defined in the algebra. Let $E(x)$ be an algebraic expression of type $\{T'\}$ with a special variable symbol of type T. Then $\lambda x.E(x)$ denotes the function mapping objects a of type T, to pomsets $E(a)$ of type $\{T'\}$. In the sequel, we will often omit the lambda, when the context is clear.

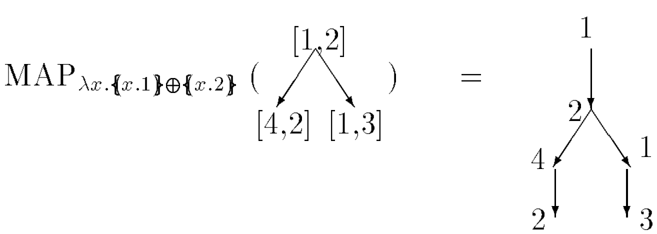
- *Restructuring,* MAP: Assume that φ is a lambda algebra expression[3] mapping objects of type T to pomsets of type $\{T'\}$. $\mathrm{MAP}_{\varphi}(p)$ is a pomset of type $\{T'\}$, obtained by replacing each $v \in V$ in p by the whole pomset $p_v = \varphi(\mu(v))$, preserving order both within and between the p_v's.

  This operation is also called in the literature on pomsets the ***expansion*** operation. (See [Pra84] for a formal definition.)

- *Cartesian product,* $\times$: Assume $p_1 : \{T_1\}$ and $p_2 : \{T_2\}$. The objects in $p_1 \times p_2$ are obtained from the Cartesian product of the objects in p_1 and in p_2. The order on the objects is the lexicographical order.

  $$p_1 \times p_2 = \langle V_1 \times V_2, \Sigma_1 \times \Sigma_2, \mu_1 \times \mu_2,$$
  $$\{([x_1, y_1], [x_1, y_2]) \mid (y_1, y_2) \in <_2\} \cup \{([x_1, y_1], [x_2, y_2]) \mid (x_1, x_2) \in <_1\}\rangle : \quad \{T_1 \times T_2\}$$

Example 3.1
$\mathrm{MAP}_{\lambda x.\{x.1\}\oplus\{x.2\}}$ denotes the pomset obtained by replacing each tuple by two objects corresponding to the first and second attributes of the tuple, and making the first attribute precede the second in the order. This is illustrated below, using (transitively reduced) graph representation of the pomsets.



As another example, $MAP_{\lambda x.\{[x.1,x.3]\}}$ computes the projection of each tuple in the pomset on its first and third arguments. For brevity, we shall denote the map operations when applied on pomset of tuples, projecting on the attributes $i_1, \ldots, i_n$ by $\Pi_{i_1,\ldots,i_n}$.

Observe that the ***selection*** operation, extended from the relational algebra, can be defined using the restructuring operation. For example,

$$\sigma_{\varphi=\varphi'}(p) = \mathrm{MAP}_{\lambda x.(=_{\emptyset}((\varphi(x)-\varphi'(x))\cup(\varphi'(x)-\varphi(x)))\times\{x\})}(p)$$

is the pomset obtained from p by removing all the vertices $v \in V$ not satisfying the equation $\varphi(\mu(v)) = \varphi'(\mu(v))$, and preserving the order on the remaining objects.

[3] φ may contain any algebra operator including MAP itself as well as the operators defined below.

Another operator important for bags [GM93] is duplicate elimination. This operator can be generalized to pomsets either by (1) considering connected components of the pomsets, and eliminating isomorphic occurrences of the same component, or (2) by considering individual objects and eliminating multiple occurrences of the same label (note that for unordered pomsets, i.e. bags, the two options are identical).

- *Duplicate elimination:*
  - ***Connected components,*** ϵ : $\epsilon(p)$ is a pomset with exactly one occurrence of each of the non-isomorphic connected components of p.
  - ***Individual objects,*** ϵ_n : $\epsilon_n(p)$ is a pomset where all the vertices $v \in V$ that are mapped by μ to the same label collapse to one unique vertex with this label, and where the order on the new objects is the maximal order consistent with that of the sources of the objects. More formally:
    $\epsilon_n(p) = \langle V', \Sigma, \mu', <' \rangle$ where $V' \subseteq V$ is such that $\mu' = \mu \downarrow_{V'}$ is 1-1, and for each $v_1, v_2 \in V'$ such that $\mu'(v_1) = a$ and $\mu'(v_2) = b$, $(v_1, v_2) \in <'$ iff there exists $v_1' \in \mu^{-1}(a)$, $v_2' \in \mu^{-1}(b)$ such that $(v_1', v_2') \in <$, and there is no $v_1'' \in \mu^{-1}(a)$, $v_2'' \in \mu^{-1}(b)$ such that $(v_2'', v_1'') \in R$, where R is the transitive closure of the relation R' defined by $R'(v_1', v_2')$ iff there exists v_1, v_2 such that $\mu(v_1) = \mu(v_1')$, $\mu(v_2) = \mu(v_2')$ and $(v_1, v_2) \in <$.

The previous definition is rather tedious, but it corresponds to a rather simple intuition. Consider the pomset

$$p = \langle \{v_1, v_2, v_3, v_4, v_5\}, \{1, 2, 3\}, \{(v_1, 1), (v_2, 2), (v_3, 3), (v_4, 3), (v_5, 1)\}, \{(v_1, v_2), (v_2, v_3), (v_4, v_5)\} \rangle$$

described graphically in the following figure. After identifying nodes with the same labels, the order is lost and we get an unordered pomset. Intuitively, this is because in the pomset there is one occurrence where 1 is greater than 2 that is greater than 3, and another occurrence where 3 is greater than 1. So globally 1, 2, and 3 are unordered. This also follows from the formal definition: when the two connected components are combined they result in a loop, so their elements are unordered.

$$\epsilon_n \left(\begin{array}{ccc} 1 & \quad & 3 \\ \downarrow & & \downarrow \\ 2 & & 1 \\ \downarrow & & \\ 3 & & \end{array} \right) = 1 \quad 2 \quad 3$$

Special pomset operators

To fully exploit the ordered types as a computational device, we need to be able to transform unordered types into partially ordered ones, and vice-versa. The relational and bag operators, do not support this. We introduce two operations which deterministically transform partially ordered pomsets into unordered ones and reciprocally.

- ***Order-destroy***, $\oplus^{-1}$: $\oplus^{-1}(\boldsymbol{p}) = \langle V, \Sigma, \boldsymbol{\mu}, \emptyset \rangle$ is an unordered pomset of type $\{T\}$, containing the same objects as $\boldsymbol{p}$.

- ***Listing***, Λ : This operation deterministically maps an **unordered** multiset (i.e. a pomset with an empty order) to a pseudo-linear pomset. More formally, $\Lambda(\langle V, \Sigma, \boldsymbol{\mu}, \emptyset \rangle) = \langle V, \Sigma, \boldsymbol{\mu}, <' \rangle$, where $<'$ is the minimal partial order such that for every label $\boldsymbol{o}$ in Σ with exactly $\boldsymbol{n}_o$ distinct $\boldsymbol{v}_i \in V$ for which $\boldsymbol{\mu}(\boldsymbol{v}_i) = \boldsymbol{o}$, $\boldsymbol{v}_1 <' \boldsymbol{v}_2 <' \ldots <' \boldsymbol{v}_{n_o}$. Otherwise, if $< \neq \emptyset$, then $\Lambda(\langle V, \Sigma, \boldsymbol{\mu}, < \rangle) = (\langle V, \Sigma, \boldsymbol{\mu}, < \rangle)$. Note that since pomsets are defined up to isomorphism on V, the specific order of the $\boldsymbol{v}_i$'s is irrelevant, and thus the operation is deterministic.

For instance, $\Lambda(\langle \{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3, \boldsymbol{v}_4, \boldsymbol{v}_5\}, \{1,2\}, \{(\boldsymbol{v}_1, 1), (\boldsymbol{v}_2, 1), (\boldsymbol{v}_3, 1), (\boldsymbol{v}_4, 2), (\boldsymbol{v}_5, 2)\}, \emptyset \rangle)$ generates the following partial order: $\boldsymbol{v}_1 \leq \boldsymbol{v}_2 \leq \boldsymbol{v}_3$ and $\boldsymbol{v}_4 \leq \boldsymbol{v}_5$.

Finally, to further exploit the internal order in a pomset, we define an operator that enables sequential iteration over components of the pomset, according to their relative order.

Iterator, I: The operator I realizes a structural-recursion [BBN91] on the pomsets. To simplify the presentation, we first explain how it behaves when applied on totally ordered pomsets (i.e. lists). Let $\emptyset$ denote the empty list of type $\{T\}$. Let $\boldsymbol{y} \neq \emptyset$ be a non empty list of type $\{T\}$, let $\boldsymbol{x}$ be a singleton list of type $\{T\}$ (i.e. a list containing one element), and let $\boldsymbol{f} :\to \{T'\}$, $\boldsymbol{g} : \{T\} \to \{T'\}$, and $\boldsymbol{h} : \{T\}, \{T\}, \{T'\} \to \{T'\}$ be three lambda algebra expressions. The iterator $I_{f,g,h}$ is defined as follows:

- $I_{f,g,h}(\emptyset) = \boldsymbol{f}$,
- $I_{f,g,h}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{x})$,
- $I_{f,g,h}(\boldsymbol{x} \oplus \boldsymbol{y}) = \boldsymbol{h}(\boldsymbol{x}, \boldsymbol{y}, I_{f,g,h}(\boldsymbol{y}))$.

Thus, the result of the iterator when applied on the empty list is determined by the algebra expression $\boldsymbol{f}$. The result when applied on a singleton list is defined by $\boldsymbol{g}$, and the result when applied on a list of length $\boldsymbol{n} > 1$ is defined recursively, using $\boldsymbol{h}$.

We next explain how the operator works when applied on partially ordered pomsets. Since lists are totally ordered, the operator can process all the elements of the list sequentially. Partially ordered pomsets do not have a total order on the individual elements, but the operator can still exploit the relative order of sub-parts of the pomsets. We say that a pomset $\boldsymbol{x}$ is ***serial*** iff there are two pomsets $\boldsymbol{x}_1, \boldsymbol{x}_2 \neq \emptyset$ such that $\boldsymbol{x} = \boldsymbol{x}_1 \oplus \boldsymbol{x}_2$. A pomset is called ***non-serial*** otherwise. Every pomset $\boldsymbol{x}$ can be viewed as a uniquely decomposed concatenation of (one or more) non-serial pomsets $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$, such that $\boldsymbol{x} = \boldsymbol{x}_1 \oplus \ldots \oplus \boldsymbol{x}_n$ (cf. Proposition 4.6).

Example 3.2 The pomset

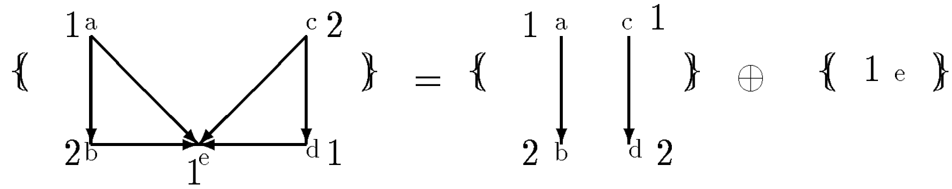
$$\boldsymbol{x} = \langle \{a, b, c, d, e\}, \{1, 2\}, \{(a, 1), (b, 2), (c, 1), (d, 2), (e, 1)\}, \{(a, b), (b, e), (c, d), (d, e), (a, e), (c, e)\} \rangle$$

is the concatenation of the non-serial pomsets

$\boldsymbol{x}_1 = \langle \{a, b, c, d\}, \{1, 2\}, \{(a, 1), (b, 2), (c, 1), (d, 2)\}, \{(a, b), (c, d)\} \rangle$,

and

$x_2 = \langle \{e\}, \{1\}, \{(e, 1)\}, \emptyset \rangle$.
Or in pictorial terms,



It turns out that the non-serial pomsets from which a pomset is composed are unique and can be computed efficiently (Proposition 4.6). When the iterator is applied to partially ordered pomsets, instead of sequentially processing single elements, it processes sequentially the non-serial pomsets from which the pomset is composed. Thus, for partially ordered pomsets, the iterator is defined by recursion on the serial structure of the pomset. The definition is the same as above, except that now $\emptyset$ denotes the empty pomset, $y \neq \emptyset$ a non empty pomset, and $x \neq \emptyset$ a non-serial pomset.

4 Properties of the Algebra

Let $\mathcal{P}$om-$\mathcal{A}$lg denote the algebra for pomsets containing all the operations as defined above. The algebra $\mathcal{P}$om-$\mathcal{A}$lg is very similar to the relational and bag algebras [GM93]. When restricted to unordered pomsets it is essentially the bag algebra, and if additionally no two vertices are allowed to have the same label (and each operation is followed by duplicate elimination) it yields the relational algebra. We use the notation $\mathcal{P}$om-$\mathcal{A}$lg$_\theta$, to denote the restriction of $\mathcal{P}$om-$\mathcal{A}$lg without the operation θ. We next study the properties and dependencies of the different operations. We first consider the duplicate elimination operator. Note that this operator received much attention in the context of object oriented languages [AK89, BP91]. It was shown in particular that it plays an important role to get complete languages. This operation is also very significant in the context of ordered types. The next propositions shows the limitations of $\mathcal{P}$om-$\mathcal{A}$lg$_{\epsilon_n}$. For that we introduce the notion of ***series-parallel*** pomsets. ***Series-parallel*** pomsets are defined as the smallest class containing singleton pomsets, and closed under parallel ($\uplus$) and series ($\oplus$) composition.

Proposition 4.1 $\mathcal{P}$om-$\mathcal{A}$lg$_{\epsilon_n}$ can construct only series-parallel pomsets.

Proof: Clearly the proposition is correct for expressions defining the empty pomset. We next prove it for non empty pomsets. The proof works by induction on the depth of the expressions defining the pomsets. In particular we show that every $\mathcal{P}$om-$\mathcal{A}$lg$_{\epsilon_n}$ expression e that constructs a non empty pomset has an equivalent expression of the form $e_1 \uplus \ldots \uplus e_n$ where each e_i is either a singleton pomset or an expression of the form $e'_i \oplus e''_i$ for some non-empty series-parallel pomsets e'_i, e''_i. (Note that each such e_i describes a connected component of the pomset.)

Basis: Clearly this is the case for the pomsetting operator $\{o\}$.

Induction: Let e', e'' be $\mathcal{P}$om-$\mathcal{A}$lg$_{\epsilon_n}$ expressions. Let $e' = e'_1 \uplus \ldots \uplus e'_n$. If e'' is an empty pomset then $e' = e' \circ e'' = e'' \circ e' = e' - e''$ for $\circ \in \{\uplus, \oplus, \bigcup\}$, and $\emptyset = e' \circ e'' = e'' \circ e' = e'' - e'$ for $\circ \in \{\times, \bigcap\}$, thus the claim follows from the induction hypothesis. Assume now that e'' is not empty and let $e'' = e''_1 \uplus \ldots \uplus e''_p$.

For $e = e' \uplus e''$ the equivalent expression is $e'_1 \uplus \ldots \uplus e'_n \uplus e''_1 \uplus \ldots \uplus e''_p$. For $e = e' \oplus e''$ the equivalent expression is $(e'_1 \uplus \ldots \uplus e'_n) \oplus (e''_1 \uplus \ldots \uplus e''_p)$.

For $e = e' \bigcup e''$ the equivalent expression is obtained by taking the $\uplus$ of the n occurrences of each connected component that n-belongs to $e' \bigcup e''$. The expression describing each such component can be any of the e'_i's or e''_i's describing this component. Similar constructions are performed for $\bigcap$, $-$, and ϵ. (Omitted)

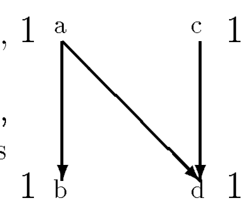
For MAP$_\varphi$ we take $e'_1 \uplus \ldots \uplus e'_n$ and replace every occurrence of an object c of type T by the series-parallel expression describing the pomset $\varphi(c)$. (Such expression exists because of the induction assumption).

For $\times$ we replace every constant c by the pomset e''_c obtained from e'' by replacing every constant d in e'' by a tuple constructed by "concatenating" c and d into one tuple.

The proof for $\oplus^{-1}(e')$, $\Lambda(e')$, and $=_\emptyset$ follows immediately from the definition of the operators.

Finally, consider $I_{f,g,h}(e')$. To construct the expression corresponding to $I_{f,g,h}(e')$ we first unfold the loop in the definition of I and then work as above on the resulting term. $\Box$

Note that there are pomsets that are not series-parallel. For example, it was shown in [Moh89] that the N-shaped pomset $\mathcal{N} = \langle\{a,b,c,d\}, \{1\}, \{(a,1),(b,1),(c,1),(d,1)\}, \{(a,b),(a,d),(c,d)\}\rangle$, with four objects all labeled with "1", is not series-parallel. It follows that:



Corollary 4.2 Not all pomsets can be constructed in $\mathcal{P}$om-$\mathcal{A}$lg$_{\epsilon_n}$.

It turns out that non series-parallel pomsets can be constructed using the duplicate elimination operator. For example, the N-shaped pomset described above can be constructed in $\mathcal{P}$om-$\mathcal{A}$lg as follows:

$$\mathcal{N} = \Pi_1(\epsilon_n((\{[1,a]\} \oplus \{[1,b]\}) \biguplus (\{[1,a]\} \oplus \{[1,d]\}) \biguplus (\{[1,c]\} \oplus \{[1,d]\}))).$$

Moreover, $\mathcal{P}$om-$\mathcal{A}$lg can construct **all** pomsets.

Proposition 4.3 All pomsets can be constructed using the operators $[]$, $\{\}$, $\uplus$, $\oplus$, ϵ_n, and Π.

Proof: The construction of arbitrary pomsets is similar to the construction of the N-shaped pomset above. Assume we want to construct a pomset $p = \langle V, \Sigma, \mu, < \rangle$. To do that we first construct a similar pomset $p' = \langle V, \Sigma', \mu', < \rangle$, where $\Sigma' = \{[\sigma, v] | \sigma \in \Sigma, v \in V\}$, and $\mu'(v) = [\mu(v), v]$. I.e. the structure of p' is the same as that of p, but the labels of nodes are modified - a new attribute is added to each label, and the value of this attribute is distinct for all nodes. To obtain p from p' we simply project out this additional attribute. We next explain how to construct p'.

Recall that p can be represented by a directed acyclic graph $G = (V, E)$. We first create a pomset p'' describing the edges in G. $p'' = \langle V'', \Sigma'', \mu'', <'' \rangle$, were $V'' = \{v_e | v \in V, e \in E$, and v is one of the endpoints of $e\}$, $\Sigma'' = \{[\sigma, v] | \sigma \in \Sigma, v \in V\}$, μ'' is such that $\mu(v_e) = [\mu(v), v]$ for every v and e, and $v_{e_1} < u_{e_2}$ iff $e_1 = e_2 = (v, u)$. (i.e. the pomset is a set of ordered pairs. Each pair represents one edge. The objects in the pair are labeled with the label and the identifier of the nodes at the endpoints of the edge. Thus two objects in the pomset have the same label iff they correspond to the same node in G). Clearly such pomset can be created using $\uplus$ and $\oplus$.

We can now obtain p' by applying ϵ_n on p'' to glue together the end points of the edges. To obtain the desired pomset p we further use Π to remove the additional attribute. $\Box$

It follows that:

Corollary 4.4 $\mathcal{P}$om-$\mathcal{A}$lg$_{\epsilon_n} \subset \mathcal{P}$om-$\mathcal{A}$lg.

Pomsets can be represented in the algebra using the operations $[], \{\}, \uplus, \oplus, \epsilon_n$, and Π, and symbols from $\Sigma \cup V$. We nevertheless often use in the following the semantical notation for pomsets, in terms of a set of vertices V, a labeling function $\mu : V \to \Sigma$, and a partial order over V.

We next consider the operations that map unordered structures to (partially) ordered structures (listing Λ), and vice versa (order-destroy $\oplus^{-1}$). We show that these two operators cannot be expressed by other operations.

Proposition 4.5

- $\mathcal{P}$om-$\mathcal{A}$lg$_\Lambda \subset \mathcal{P}$om-$\mathcal{A}$lg
- $\mathcal{P}$om-$\mathcal{A}$lg$_{\oplus^{-1}} \subset \mathcal{P}$om-$\mathcal{A}$lg

Proof: We start with the first claim. The ***depth*** of a pomset $p = \langle V, \Sigma, \mu, < \rangle$ is the maximal number of vertices $v_1, \dots v_n \in V$ such that $v_i < v_{i+1}$ for each $i \in \{1 \dots n-1\}$.

We show that for every expression ϕ in $\mathcal{P}$om-$\mathcal{A}$lg$_\Lambda$ and every constant c, there exists some constant $k_{\phi,c}$, such that if the depth of a pomset p is bounded by c, then the depth of the pomset $\phi(p)$ is bounded by $k_{\phi,c}$.

The first claim of the proposition then follows. Indeed, consider a non ordered pomset p were all the objects are labeled with a single label a, i.e. $p = \langle V, \{a\}, \{(v, a) | v \in V\}, \emptyset \rangle$, for some V. Since the depth of $\Lambda(p)$ depends on p, it cannot be bounded by any constant $k_{\phi,1}$. Thus Λ is not expressible in $\mathcal{P}$om-$\mathcal{A}$lg$_\Lambda$.

The proof works by induction on the size of ϕ. For all the operation except the iterator I the proof is trivial and follows from the definition. For $I_{f,g,h}$, if the input is empty or is non serial, then $I_{f,g,h}(\boldsymbol{input}) = f$ or $I_{f,g,h}(\boldsymbol{input}) = g(\boldsymbol{input})$ and the correctness follows immediately from the induction assumption. Otherwise, since the depth of the input is bounded by some constant k, I iterates at most k times. The depth of the input at the first iteration step is bounded by k, and the depth of the output is bounded by $k_{g,k}$. For the jth iteration step, $j \leq k$ the depth of the output is bounded by $k_{h,m}$, where m is the maximum between k and the bound on the output of the previous iteration. Q.E.D.

We next consider the second claim. We prove the result by showing that for every expression ϕ of $\mathcal{P}$om-$\mathcal{A}$lg$_{\oplus^{-1}}$ and for every constant c, there exists some constant $k_{\phi,c}$, such that if p is a pomset where (i) the number of connected components is bounded by c, and (ii) each connected component p' in p is the concatenation of at most c non serial pomsets, then $\phi(p)$ is a pomset where (i) the number of connected components is bounded by $k_{\phi,c}$, and (ii) each connected component p' in p is the concatenation of at most $k_{\phi,c}$ non serial pomsets.

To prove the claim we look at an N-shaped pomset p, where all the three branches are lists, each with n occurrences of an element a for some number n, i.e. $p = \langle V, \{a\}, \mu, <\}$ where $V = \{i_1, \ldots, i_n, j_1, \ldots, j_n, k_2, \ldots, k_{n-1}\}$, $\mu(v) = a$ for all $v \in V$, and $i_\ell < i_m$, $j_\ell < j_m$ and $k_\ell < k_m$ for all $\ell < m$ and $k_2 < i_1$, $k_{n-1} < j_n$. Note that p has a single non-serial connected component so $c = 1$. $\oplus^{-1}(p)$ creates an unordered pomset having $|V|$ connected components. The number of connected components is a function of V and thus cannot be bounded by any constant $k_{\phi,1}$. It follows that $\oplus^{-1}$ is not expressible in $\mathcal{P}$om-$\mathcal{A}$lg$_\Lambda$.

The proof works by induction on the size of ϕ. The only non trivial case is for the iterator I. As above, the rest of the proof is omitted. If the input to the iterator is empty or is non serial, then $I_{f,g,h}(input) = f$ or $I_{f,g,h}(input) = g(input)$ and the correctness follows immediately from the induction assumption. Otherwise, since each connected component is the concatenation of at most k non serial pomsets, for some constant k, I iterates at most k times. Clearly, the number of connected components in the input for the first iteration is bounded by k and so is the number of non serial pomsets from which each connected component is composed. Thus the number of connected components of the output of this iteration is bounded by $k_{g,k}$. For the jth iteration step, $j \leq k$, the number of connected components and the number of non serial pomsets composing each component is bounded by max$k_{h,m}$, where m is the maximum between k and the bound on the output of the previous iteration. $\Box$

We conclude this section by a technical observation on the complexity of "serializing" a pomset. It constitutes a fundamental tool in the tractability proof in Section 5. The main point here is the complexity bound.

Proposition 4.6 Every non empty pomset x has a unique decomposition into non-empty non-serial pomsets $x_1, \ldots, x_n$, such that $x = x_1 \oplus \ldots \oplus x_n$. This decomposition can be computed in time polynomial in the size of x.

Proof: The algorithm $decompose(x)$ sketched below gets as input a non-empty pomset x and returns an integer n and an array $result$ such that $result[1], \ldots, result[n]$ are non empty non-serial pomsets such that $x = result[1] \oplus \ldots \oplus result[n]$.

```
decompose(x) returns (n, result)
    n = 0; rest = x
    repeat while rest ≠ ∅
      n := n + 1
      [result[i], rest] := dec(rest)
    end;
end;
```

The algorithm uses a procedure $\boldsymbol{dec}(\boldsymbol{p})$ that given a non empty pomset $\boldsymbol{p}$ returns two pomsets $\boldsymbol{p}_1, \boldsymbol{p}_2$ such that $\boldsymbol{p}_1 \neq \emptyset$ is a non-serial pomset and $\boldsymbol{p} = \boldsymbol{p}_1 \oplus \boldsymbol{p}_2$. As mentioned above a pomset $\boldsymbol{p} = \langle V, \Sigma, \mu, < \rangle$ can be represented by a directed acyclic graph $G = (V, E)$ where each node $\boldsymbol{v} \in V$ has a label in Σ attached to it, and where an edge $(\boldsymbol{v}_1, \boldsymbol{v}_2) \in E$ iff $\boldsymbol{v}_1 < \boldsymbol{v}_2$. The procedure $\boldsymbol{dec}(\boldsymbol{p})$ uses this graph representation.

```
dec(p) returns (p1, p2)
  Let G = (V, E) be the transitively closed graph representation of p;
  Let S := sources(G)
  Repeat until there are no changes in S
    If there exist two nodes v1 ∈ S and v2 ∈ V − S, such that (v1, v2) ∉ E,
    then add v2 to S
  return the pomset induced by the nodes in S as p1
  return the pomset induced by the nodes in V − S as p2
end;
```

We next prove the correctness of the algorithm and the uniqueness of the decomposition. We first show that for every non-empty pomset $\boldsymbol{p}$, $\boldsymbol{dec}(\boldsymbol{p})$ returns two pomsets $\boldsymbol{p}_1$,$\boldsymbol{p}_2$ such that (1) $\boldsymbol{p}_1 \neq \emptyset$ is a non-serial pomset (2) $\boldsymbol{p} = \boldsymbol{p}_1 \oplus \boldsymbol{p}_2$, and (3) there are no other pomsets $\boldsymbol{p}'_1$, $\boldsymbol{p}'_2$ satisfying 1 and 2. Once this is proved, the correctness of the algorithm and the uniqueness of the decomposition is easily proved by induction on the number of calls for $\boldsymbol{dec}$. We omit the induction.

We first prove (1). Clearly, since $\boldsymbol{p}$ is not empty $\boldsymbol{sources}(\boldsymbol{p}) \neq \emptyset$ and thus $\boldsymbol{p}_1$ is not empty. To show that $\boldsymbol{p}_1$ is non-serial, we have to show that there are no two non-empty pomsets $\boldsymbol{y}, \boldsymbol{z}$ such that $\boldsymbol{p}_1 = \boldsymbol{x} \oplus \boldsymbol{y}$. The proof works by contradiction. We show below that such $\boldsymbol{x}$ must contain all the elements corresponding to the nodes in S, thus in fact must be equal to $\boldsymbol{p}$ - a contradiction to the non-emptiness of $\boldsymbol{y}$. We use induction on the number of iteration of the loop in $\boldsymbol{dec}$.

Basis: 0 iterations - $\boldsymbol{p}_1$ contains all the sources of $\boldsymbol{p}$. If $\boldsymbol{x}$ does not contain all the sources, then $\boldsymbol{y}$ must contain some of them. But in that case $\boldsymbol{p}_1 \neq \boldsymbol{x} \oplus \boldsymbol{y}$ since in $\boldsymbol{p}$ all the nodes representing sources are incomparable (non ordered) while in $\boldsymbol{x} \oplus \boldsymbol{y}$ all the nodes in x (and in particular those corresponding to sources) precede the nodes in $\boldsymbol{y}$ (and in particular those corresponding to the other sources of $\boldsymbol{p}$).

Induction: Assume that after the i^{th} iteration, the current S is included in $\boldsymbol{x}$. Let $\boldsymbol{v}_2$ be the node added to S in the $i+1$ iteration. Let $\boldsymbol{v}_1 \in S$ be the node causing the insertion. Since $(\boldsymbol{v}_1, \boldsymbol{v}_2) \notin E$, $\boldsymbol{v}_1$ does not precede $\boldsymbol{v}_2$ in $\boldsymbol{p}$. Thus $\boldsymbol{p}$ can not be equal to $\boldsymbol{x} \oplus \boldsymbol{y}$ where $\boldsymbol{v}_1$ belong to $\boldsymbol{x}$ and $\boldsymbol{v}_2$ belong to $\boldsymbol{y}$. Thus $\boldsymbol{v}_2$ must belong to $\boldsymbol{x}$ too.

We next prove (2). Clearly if $\boldsymbol{p}_2 = \emptyset$ then $\boldsymbol{p}_1 = \boldsymbol{p}$ and $\boldsymbol{p} = \boldsymbol{p}_1 \oplus \boldsymbol{p}_2$. If $\boldsymbol{p}_2 \neq \emptyset$ then from the algorithm it follows that every node $\boldsymbol{v}$ in $V - S$ must have incoming edges from **all** the nodes in S (otherwise $\boldsymbol{v}$ would be added to S). It follows that all the nodes in S precede all the nodes in $V - S$, thus $\boldsymbol{p} = \boldsymbol{p}_1 \oplus \boldsymbol{p}_2$.

Finally we prove (3). Again, we use contradiction. Assume that $\boldsymbol{p}_1, \boldsymbol{p}_2$ are not unique, and let $\boldsymbol{p}'_1, \boldsymbol{p}'_2$ be another alternative. First observe that $\boldsymbol{p}'_1$ must contain all the elements corresponding to the nodes in S, thus in fact must contain all the element is $\boldsymbol{p}_1$. (The proof is the same as that of (1) above.) Next observe that $\boldsymbol{p}'_1$ cannot contain elements not in S. This is because from the algorithm it follows that all the elements corresponding to nodes in S precedes all the other elements of $\boldsymbol{p}$, and in particular the elements in $\boldsymbol{p}'_1$ that are not in S (if such exist). Thus if $\boldsymbol{p}'_1$ has nodes not in S then $\boldsymbol{p}'_1 = \boldsymbol{p}_1 \oplus \boldsymbol{x}'$ for some non empty $\boldsymbol{x}'$, a contradiction to the non-seriality of $\boldsymbol{x}$. From the two observations above it follows that $\boldsymbol{p}'_1 = \boldsymbol{p}_1$, thus the decomposition is unique. $\Box$

5 Tractable Pomset Queries

We next study the complexity of evaluating queries in the pomset algebra. As in the classical relational case, we are aiming for characterizations of $\mathcal{P}$om-$\mathcal{A}$lg in terms of complexity classes of queries. It turns out that some of the operations are rather expensive, but when appropriately restricted, they become tractable.

5.1 Operations on Connected Components

First consider the four operations $\bigcup, \bigcap, -$, and ϵ, that operate on connected components of pomsets. Note that these operations require a graph isomorphism test. To have a tractable language, these operations must be restricted. Let F be a family of graphs for which graph isomorphism is polynomial, and where the test of membership in F is also polynomial. We restrict the operations so that if the input pomset is not in F, then the result of the operation is the empty pomset. The class of **planar graphs** satisfy these two requirements [HT74, HW74]. We assume in the following that the operations are restricted to pomsets whose (transitively reduced) graph representation is planar. This guarantees tractability of the operations.

Such a restriction might seem artificial, but it is not unusual in computer science. In robotics for instance, the 3-dimensional world is modeled with only 2 dimensions, when considering problems such as the shortest path between two points amid polyhedral obstacles, which is NP-hard in dimension 3 [CR87], and tractable in dimension 2.

5.2 The iterator

It is important to note that a careless use of the iterator may cause exponential explosion of the size of the result. Consider for example the case where: $f = \emptyset$, $g(x) = a$, and $h(x,y,z) = z \oplus z$. In this case, if l is a list, then $I_{f,g,h}(l)$ is a list of a's of length $2^{|l|}$.

Consider the algebra containing all the operators described above, including the iterator I. Given an algebra expression e, we call the problem of testing whether there exists an input p for which $e(p)$ is of size (at least) exponential in $\mid p \mid$, the ***exponential explosion problem***. It can be shown that:

Theorem 5.1 The exponential explosion problem is undecidable.

Proof: The proof is done by reduction from the problem of testing if a first-order sentence is satisfiable. For a first-order sentence θ, it is easy to build a $\mathcal{P}$om-$\mathcal{A}$lg expression e such that θ is satisfiable iff there exists an input $p = \langle V, \Sigma, \mu, < \rangle$ for which $e(p)$ is not empty. (The construction is the same as the one for relational algebra [Ull88], with additional application of duplicate elimination after each operation). Without loss of generality assume that p is a list, i.e. $<$ is total. Now, consider the expression $e'(p) = I_{f,g,h}(p) \times e(p)$, where $I_{f,g,h}$ is the expression with exponential growth defined at the beginning of this subsection. $e'(p)$ is of size exponential in the size of p iff θ is satisfiable. $\Box$

It is easy to verify that every expression in the algebra can be computed in some hyper-exponential time in the size of its input, that is with the time bounded by some function defined by a tower of exponentials. The height of the hyper-exponential depends on the number of imbricated iterators. The exponential growth is due to the duplicates generated in the computation. We consider below two restrictions on the language that ensure tractability by restricting the number of duplicates being generated. Both restrictions apply to the use of the iterator I.

Bounded Iterator: The first restriction is similar to the bounded fixpoint operator of [Suc93]. The idea is to restrict the iterator such that the size of the intermediate result in each step of the computation is bounded. The bound is defined using an algebra expression b. The bounded iterator $I_{f,g,h|b}$, when applied on a pomset p, behaves exactly like the unbounded operator, except that it ignores intermediate results whose size is bigger than $b(p)$. More formally:

$I_{f,g,h|b}(p) = I_{f,g,h'}(p)$, where $h'(x,y,z) = h(x,y,z)$ if $|h(x,y,z))| < |b(p)|$, and $h'(x,y,z) = z$ otherwise.

The key observation is that if only bounded iterators are used in the algebra, then $b(p)$ is guaranteed to be of size polynomial in p, and thus all the intermediate results are of size polynomial in the size of the input.

Let $\mathcal{P}$om-$\mathcal{A}$lg^b be the algebra containing all the operators of $\mathcal{P}$om-$\mathcal{A}$lg (with the above restrictions on operations on connected components), and where the iterator is bounded. The complexity of $\mathcal{P}$om-$\mathcal{A}$lg^b is considered in the next theorem.

Theorem 5.2 $\mathcal{P}$om-$\mathcal{A}$lg^b is in PTIME.

Proof: The input/output of the Turing machine computing the $\mathcal{P}$om-$\mathcal{A}$lg^b query is the standard encoding of the labeled partial orders representing the pomsets. The proof uses Proposition 4.6 to guarantee that the iterator can be computed in polynomial time. The essential problem is to show that the number of vertices used for the representation of the pomsets can be bounded. The proof follows from the following claims:

Claim (i) The maximal number of vertices at any step of the evaluation of the query is bounded by some polynomial in the size of the input.

Claim (ii) All the operations can be computed in polynomial time in the size of their input. That is the resulting order relation and labeling function are computable in polynomial time.

A polynomial bound on the number of vertices, $|V|$, is sufficient to polynomially bound the size of the whole pomset $\langle V, \Sigma, \mu, < \rangle$. Indeed, we have seen above that the number of labels, $|\Sigma|$, is bounded. Moreover, the partial order, $< \subseteq V \times V$, and the labeling function, $\mu \subseteq V \times \Sigma$, are by definition polynomially bounded in the size of V. For simplicity, we neglect the size of the order, $<$, and of the labeling function, μ, and consider only the bound on the size of the set of vertices, V, in the rest of the proof.

The proof is done by induction on the number of operations in the expression. The two claims are proved simultaneously by induction on the nesting of the operators in an algebraic expression $e(l_1, \ldots, l_n)$.

Basis: For zero operations this is by definition.

Induction: Assume that the assumption holds for expressions with at most i nested operators. We prove that it holds for one more level of nesting of the operators. Let e_1, e_2 be expressions with less than i nested operators, and let P_1, P_2 be their corresponding polynomials.

It is clear that $e_1 \uplus e_2$ can be computed in polynomial time, and the resulting polynomial bound on the number of vertices is $P_1 + P_2$. The same holds for $e_1 \oplus e_2$, with the same[4] resulting polynomial $P_1 + P_2$; and for $e_1 \times e_2$, with a resulting polynomial $P_1 \times P_2$.

The four operations that operate on the connected components of the pomsets, $\bigcup, \bigcap, -$, and ϵ, are computable in polynomial time in the size of their inputs, by assumption since they apply only on planar graphs, and both planarity, and planar graph isomorphism are computable in polynomial time. The resulting polynomial is bounded by $P_1 + P_2$ in the case of $e_1 \bigcup e_2$, $e_1 \bigcap e_2$, $e_1 - e_2$, and P_1 in the case of $\epsilon(e_1)$.

The individual object's duplicate elimination, ϵ_n, can be computed in polynomial time, by first computing the transitive closure of the order after identification of the vertices with same label. The resulting polynomial of $\epsilon_n(e_1)$ is P_1.

[4]Note that the size of the order, $<$, is actually quadratic in $P_1 + P_2$, but we neglect it.

The result is trivial in the case of ***order-destroy***, $\oplus^{-1}$, ***listing***, Λ, and emptyness test $=_{\emptyset}$.

$\mathrm{MAP}_{\lambda x.e_1} e_2$, can be computed in polynomial time with a resulting polynomial $P_1 \times P_2$. Each vertex $v \in e_2$ is replaced by a new set of vertices $\{(v, v')|v' \in e_1\}$. The new order is the lexicographic combination of the orders of e_1 and e_2 respectively, and the labeling function $\mu(v, v')$ returns the label of v' in $e_1(v)$. This can be computed in polynomial time by induction hypothesis.

The more subtle case concerns the bounded iterator. Consider the expression: $I_{f,g,h|b}(e)$, where f, g, h, b, and e are expressions with less than i nested operators, with associated polynomials P_f, P_g, P_h, P_b, and P_e respectively.

The iteration is done over e. By Proposition 4.6, e can be serialized in polynomial time, so the number of steps is bounded by the polynomial P_e. We prove that each iteration step can be performed in polynomial time. It is clear for the initial steps when f or g are applied (by induction hypothesis).

Now, the input of the function h is of the form $(x, y, I_{f,g,h}(y))$. Since the iterator is bounded, the size of $I_{f,g,h}(y)$ is bounded by the size of b, which is polynomial by induction hypothesis. Again by induction hypothesis, the function h can be computed in polynomial time in the size of its input, and generates a polynomial number of vertices. Therefore, from the combination of the two previous arguments, we can conclude that the function h can be computed in polynomial time at each iteration steps, and the number of vertices is bounded by $P_h(P_e \times P_b)$.

It follows that the full iteration can be computed in polynomial time in the size of the input, and the number of vertices generated is bounded by: $P_h(P_e \times P_b) + P_f + P_g$. $\Box$

There are various techniques to define a tractable iteration. We next present alternative definitions of the iterator.

Duplicate Elimination: Another way to restrict the number of duplicates generated in the computation is to enforce the use of the duplicate elimination operator. We say that an iterator $I_{f,g,h}$ is ***duplicate free*** if f, g and h have the form $\epsilon_n(f')$, $\epsilon_n(g')$ and $\epsilon_n(h')$ resp., for some algebra expressions f', g', h'. Duplicate free iterators cannot cause exponential explosion of the (intermediate) results.

Let $\mathcal{P}\text{om-}\mathcal{A}\text{lg}^{\epsilon}$ be the algebra containing all the operators of $\mathcal{P}\text{om-}\mathcal{A}\text{lg}$, (with the above restrictions on operations on connected components), and where only duplicate free iterator is used. It is easy to verify that:

$$\mathcal{P}\text{om-}\mathcal{A}\text{lg}^{\epsilon} \subseteq \mathcal{P}\text{om-}\mathcal{A}\text{lg}^{b} \subseteq \text{PTIME}.$$

In fact, a less drastic restriction, where ϵ_n is enforced only after the operations $\times$, MAP, and I, and when h has more than one occurrence of the variable y, is sufficient to prevent exponential explosion. But for the results considered in this paper, the above restrictive definition of the duplicate free iterator is sufficient.

We shall see in the following a special case where the two restrictions on the iterator yield equivalent languages. It is open whether in the general case the inclusion is strict.

6 The Expressive Power of the Algebra

We next study the expressive power of the pomset algebra. As in the classical relational case, we are aiming for characterizations of the expressiveness of $\mathcal{P}$om-$\mathcal{A}$lg in terms of complexity classes of queries. In particular, we show that even unordered inputs with no duplicates can benefit from the use of ordered data types.

6.1 RALG, BALG, and FO+FP+C

To measure the expressiveness gained by the use of ordered types, we compare the expressive power of the pomset algebra to that of the relational algebra (denoted below as RALG) [Ull88], the bag algebra (denoted as BALG) [GM93], and various extensions of first order logic (FO), such as fixpoint logic (FO+FP), and fixpoint logic with counting (FO+FP+C) [GO93].

The relational and bag algebras can be naturally derived from the pomset algebra by restricting it to respectively unordered structures with no duplicates, and unordered structures. We briefly describe below the language FO+FP+C (for more details see [GO93]).

FO+FP+C is a two-sorted logic, with the sorts ***domain*** (unordered) and ***integer*** (with a linear order). The semantics is based on finite structures with a finite segment of the integers. For any finite relational structure $\mathcal{A} = (A, R_1, \ldots, R_n)$, the structure ***with a finite segment of the integers*** $\mathcal{A}^*$ is defined by: $\mathcal{A}^* = \mathcal{A} \cup (\{0, ..., n-1\}, \leq)$, where $n = \mid A \mid$. If $\varphi(x)$ is a formula with a free variable x of sort domain, then $\#_x[\varphi(x)]$ is a term of sort integer, interpreted as the cardinality of $\{a \mid \varphi(a)\}$. FO+FP+C allows the manipulation of numbers from 0 to n^k for some k, by using tuples of integers of the form $[i_1, \ldots, i_k]$ where each i_j varies from 0 to $n-1$. In general, we will omit the exact representation of the numbers and consider directly numbers from 0 to some n^k.

6.2 Expressiveness

When comparing the pomset algebra to RALG, BALG, FO+FP, and FO+FP+C, one consider queries over the same kind of inputs. We thus restrict below our attention to $\mathcal{P}$om-$\mathcal{A}$lg queries mapping set inputs to set outputs. We first show that the pomset algebra is more powerful than the relational and bag algebras.

Theorem 6.1 FO = RALG $\subset$ BALG $\subset$ $\mathcal{P}$om-$\mathcal{A}$lg$^{\epsilon}$ $\subseteq$ $\mathcal{P}$om-$\mathcal{A}$lg^{b}.

The strict inclusion of the relational algebra in the bag algebra was proved in [GM93]. Clearly the bag algebra is included in the pomset algebra. The fact that the inclusion is strict follows from the fact that the parity query, $EVEN$, true if the cardinality of the input is even (which is not expressible in BALG [GM93, LW94]), is expressible in $\mathcal{P}$om-$\mathcal{A}$lg$^{\epsilon}$.

Proposition 6.2 $EVEN \in$ $\mathcal{P}$om-$\mathcal{A}$lg$^{\epsilon}$.

Proof: Let R be a relation. The parity of the cardinality of R is expressed by:

$$EVEN(R) \equiv =_{\emptyset}(I_{\lambda x.\epsilon_n(\emptyset), \lambda x.\epsilon_n(\{\![a]\!\}), \lambda x\ y\ z.\epsilon_n(\{\![a]\!\}-z)}(\Lambda(\mathrm{MAP}_{\{\![a]\!\}}(R))))$$

First a list of a's of length the cardinality of R is constructed. Then, the iterator realizes a flip-flop with two states $\{\![a]\!\}$ and $\emptyset$ on the previous list. $\Box$

As we show below, the iterator I is very similar to the traditional fixpoint operator. Thus, the fact that FO is strictly included in $\mathcal{P}$om-$\mathcal{A}$lg is not surprising. It turns out however, that $\mathcal{P}$om-$\mathcal{A}$lg is also more expressive than FO+FP.

To prove that, we use the next proposition. It shows that $\mathcal{P}$om-$\mathcal{A}$lg, even without the iterator I, can still compute EVEN in the presence of an order relation (which is not possible in FO+FP). Recall that we use the notation $\mathcal{P}$om-$\mathcal{A}$lg$_\theta$, to denote the restriction of $\mathcal{P}$om-$\mathcal{A}$lg without the operation θ. So $\mathrm{P}om{-}\mathrm{A}lg_I$ denotes $\mathrm{P}om{-}\mathrm{A}lg$ without I.

Proposition 6.3 EVEN $\in \mathcal{P}$om-$\mathcal{A}$lg$_I+<$.

Proof: (sketch) Let R be a k-ary relation. We extend $<$ to an order relation $<_k$ on k-ary tuples. Then, for a given relation R, we construct a $2k$-ary relation R', such that $[x_1,\ldots,x_k,y_1,\ldots,y_k] \in R'$ iff $[x_1,\ldots,x_k]$, $[y_1,\ldots,y_k]$ are in R and $[x_1,\ldots,x_k]$ is the immediate successor of $[y_1,\ldots,y_k]$ in R. R' can be easily defined in $\mathcal{P}$om-$\mathcal{A}$lg$_I$. To simplify the presentation, we assume that $k=1$ (the general case is handled similarly). Consider the pomset constructed by the expression:

$$E(R,<) = \mathrm{MAP}_{\{\!a\!\}}(\epsilon_n(\mathrm{MAP}_{\{\!x.1\!\}\oplus\{\!x.4\!\}}(\sigma_{x.2=x.3}R' \times R'))).$$

The pomset $E(R,<)$ contains two lists. One corresponding to tuples in odd position in the order, and the other corresponding to tuples in even positions in the order. Parity can now be checked by comparing the length of these two lists. In particular, R is of even cardinality iff the two lists are equal. Thus $\mathrm{EVEN}(R) \equiv =_\emptyset(=_\emptyset(E(R,<) - \epsilon(E(R,<))))$. $\Box$

We are now ready to show the following inclusions for the class of standard relational inputs.

Theorem 6.4 FO+FP $\subset \mathcal{P}$om-$\mathcal{A}$lg$^\epsilon \subseteq \mathcal{P}$om-$\mathcal{A}$lg^b.

Proof: (sketch) The inclusion of FO+FP in $\mathcal{P}$om-$\mathcal{A}$lg$^\epsilon$ is proved by induction, building for every FO+FP formula a $\mathcal{P}$om-$\mathcal{A}$lg$^\epsilon$ expression. The translation of everything except the FP operator is similar to the classical translation from calculus to relational algebra [Ull88], and is thus omitted. The only difference in the translation is that here we need to add an application of duplicate elimination after each operation. To translate the fixpoint operator we use the iterator I. Note that the number of iterations of FP on a formula ϕ is bounded by a polynomial in the size of the input p with the arity of the formula ϕ as exponent. Let Φ be the $\mathcal{P}$om-$\mathcal{A}$lg$^\epsilon$ expression corresponding to ϕ. To simulate the fixpoint computation of ϕ on p, we create a list L_p whose length is the bound on the number of iterations, and then use the iterator:

$$I_{\lambda x.\epsilon_n(\emptyset),\lambda x.\epsilon_n(\Phi(p)),\lambda x\ y\ z.\epsilon_n(z\cup\Phi(z))}(L_p),$$

to repeatedly compute Φ. The list L_p is created by:

$$L_p = \Lambda(\mathrm{MAP}_{[\,]}(Q \times \ldots \times Q)),$$

where Q is a relation containing all the objects in the active domain (constructed as in the standard relational case [Ull88]) and the Cartesian product is applied k times where k is the exponent of the polynomial bound of the computation.

The strict inclusion follows from the fact that $EVEN$ is not expressible in FO+FP. □

We proved that fixpoint can be simulated in $\mathcal{P}$om-$\mathcal{A}$lg. We next prove that the ***counting*** construct can also be simulated. In particular, we show that (the tractable portion of) $\mathcal{P}$om-$\mathcal{A}$lg, when restricted to types that are either unordered types (such as sets and bags) or totally ordered types (such as lists) yields exactly the same expressive power as first order logic augmented with fixpoint and counting. Let $\mathcal{P}\text{om-}\mathcal{A}\text{lg}^b|_{u,t}$ (respectively $\mathcal{P}\text{om-}\mathcal{A}\text{lg}^\epsilon|_{u,t}$) be the restriction of $\mathcal{P}\text{om-}\mathcal{A}\text{lg}^b$ (respectively $\mathcal{P}\text{om-}\mathcal{A}\text{lg}^\epsilon$) to pomsets that are either unordered or totally ordered.

This restriction is enforced as follows. Whenever, the output of an algebraic operation is neither of the unordered form, $p = \langle V, \Sigma, \mu, \emptyset \rangle$, nor of the totally ordered form, $p = \langle V, \Sigma, \mu, < \rangle$, where $<$ is a linear order over V, then the output is by definition the empty pomset.

Theorem 6.5 FO+FP+C $= \mathcal{P}\text{om-}\mathcal{A}\text{lg}^\epsilon|_{u,t} = \mathcal{P}\text{om-}\mathcal{A}\text{lg}^b|_{u,t}$.

The rest of this sub-section is devoted to the proof of the previous theorem. The proof works by induction on the structure of formulas and is very technical. Readers not interested in the details, can skip the rest of this sub-section and move to Subsection 6.3.

Clearly $\mathcal{P}\text{om-}\mathcal{A}\text{lg}^\epsilon|_{u,t} \subseteq \mathcal{P}\text{om-}\mathcal{A}\text{lg}^b|_{u,t}$. To prove the theorem it suffices to show that FO+FP+C $\subseteq \mathcal{P}\text{om-}\mathcal{A}\text{lg}^\epsilon|_{u,t}$ and that $\mathcal{P}\text{om-}\mathcal{A}\text{lg}^b|_{u,t} \subseteq$ FO+FP+C. We start with the first claim.

Lemma 6.6 FO+FP+C $\subseteq \mathcal{P}\text{om-}\mathcal{A}\text{lg}^\epsilon|_{u,t}$

Before going into the details of the proof, we consider a simple example of query in FO+FP+C. The equivalent algebra expression is rather complex, and the idea behind its construction is explained in the inductive proof below.

Example 6.1 Assume that R is a binary (symmetric) relation representing a graph. A graph is regular if all vertices have same degree. The following sentence expresses regularity:

$$\exists i \forall x \#_y R(x, y) = i.$$

Its algebraic equivalent expression is:

$$I_{\{\![\;]\!\},E(x),E(\oplus^{-1}(y))\bigcup z}(L)$$

where

$L = \Lambda(\mathrm{MAP}_{[\;]}(R))$

and

$E(i) \;\equiv\; =_{\emptyset}(\; \Pi_{[\;]}(\; (\Pi_1(R) - (i \times \epsilon(\Pi_1(R)))) \times ((i \times \epsilon(\Pi_1(R))) - \Pi_1(R))\;)\;)$

The idea behind this expression is that if the graph is regular then there must exist some integer i such that $\Pi_1(R)$ contains exactly i occurrences of each element (vertex) of R. The expression $E(i)$ checks if this condition holds for a given i. The iterator then basically iterates and apply this test for increasing i's – the test performed by $E(i)$ is repeated for all i smaller than the cardinality of the set of vertex of the graph.

The integer i (the input for E) is represented by a pomset containing i non-ordered occurrences of the empty tuple. The pomset constructed by teh sub-expression $(i \times \epsilon(\Pi_1(R)))$ contains exactly i occurrences of each element (vertex) of R. If the graph is regular of degree i then so does $\Pi_1(R)$. Thus the goal of the full expression E is to compare these two pomstes and test if they are equivalent.

Proof of Lemma 6.6: We follow the lines of the classical translation from calculus to algebra [Ull88, AB87, BM92]. It is based on an induction on the structure of the FO+FP+C formulas. We omit classical encodings not specific to FO+FP+C, and focus on the special treatment is needed for handling integers, counting expressions, and quantification over integers. An integer i is represented by a bag containing i occurrences of a constant. A counter i_t associated with a tuple t is represented by i occurrences of t. A quantification over an integer variable is simulated by using an iterator that checks all the possible assignments to that integer. Finally, we use the iterator I to simulate fixpoint computation.

The inductive construction is presented below. We treat simultaneously formulas and terms.

Basis: We distinguish the two sorts, ***domain*** and ***integer***. A ***domain*** constant is represented by itself in $\mathcal{P}$om-$\mathcal{A}$lg. An ***integer*** constant i is represented by a pomset $\langle \{v_1, \ldots, v_i\}, \{[\,]\}, \{(v_j, [\,]) | j = 1 \ldots i\}, \emptyset \rangle$, i.e. a bag containing i non-ordered occurrences of the empty tuple.

A variable x of sort ***integer*** is translated to a variable x^p of pomset type $\{\![U]\!\}$ (Note that queries in FO+FP+C do not have free variables of type integer, thus the pomset variable x^p above will be further treated when considering the quantification bounding the integer variables).

An atom of the form $R_i(x)$ is translated to R_i^a.

Finally, to translate an equality atom $x = y$ when x, y are of sort ***domain***, we use Cartesian product of the active domain (constructed as in the standard relational case using projection and Cartesian product), and selection by equality.

Induction: For the induction step it suffices to consider cases where ϕ is constructed using $\wedge, \neg$, $\exists$, FP, #, and $\leq$. Let ϕ_1, ϕ_2 be formulas (or terms) in FO+FP+C, and assume that E_1, E_2 are the corresponding $\mathcal{P}$om-$\mathcal{A}$lg$^{\epsilon}|_{u,t}$ expressions.

- $\#_x(\phi_1(\overline{x}))$ is translated to $\Pi_{v-\{x\}}(E_1)$, where $v - \{x\}$ denotes the attributes of E_1 (where each attribute matches one free variable of ϕ_1) except the one corresponding to the variable x.
- $\phi_1 \leq \phi_2$ is translated to $\epsilon(((E_1 \uplus \epsilon(E_1)) \bowtie \epsilon(E_2)) - (\epsilon(E_1) \bowtie E_2))$.

where the join is performed on attributes representing identical variables. (Join is implemented using $\times$, $\boldsymbol{\sigma}$, and MAP. The reason for choosing this translation is explained below. The result of E_1 is of the form $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n, \ldots, \boldsymbol{x}_n\}$ where $\boldsymbol{x}_k$ has $\boldsymbol{i}_k$ occurrences. The $\boldsymbol{x}_k$ tuples represent the possible truth assignments to the free variables in $\boldsymbol{\phi}_1$. The number of occurrences corresponds integer associated with each assignment (the counter). Similarly the result of E_2 is of the form $\{\boldsymbol{y}_1, \ldots, \boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_m, \ldots, \boldsymbol{y}_m\}$.

A tuple $\boldsymbol{t}$ belongs to $\epsilon(((E_1 \uplus \epsilon(E_1)) \bowtie \epsilon(E_2)) - (\epsilon(E_1) \bowtie E_2))$ iff $\boldsymbol{t} = \boldsymbol{x}_p \bowtie \boldsymbol{y}_q$ for some $\boldsymbol{x}_p \in E_1$ $\boldsymbol{y}_q \in E_2$, and $\boldsymbol{i}_p + 1 > \boldsymbol{j}_q$, i.e. $\boldsymbol{x}_p$ and $\boldsymbol{y}_1$ agree on the value of the common attributes, and the number of occurrences of $\boldsymbol{x}_p$ in E_1 is grater then the number of the occurrences of $\boldsymbol{y}_q$ in E_2. This is exactly what $\boldsymbol{\phi}_1 \leq \boldsymbol{\phi}_2$ computes.

- The treatment of $\wedge$ and $\neg$ is the standard one, with an additional application of duplicate elimination on the result. (It is important to note that due to the way we treat quantification of integer variables later on, the duplicate elimination does not cause loss of information regarding the counters associated with tuples). For $\wedge$ we use Cartesian product followed by equality selection for variables with the same names, projection for omitting multiple occurrences of the same variable name, and finally duplicate elimination. For negation we take the complement with respect to the active domain.

- The fixpoint operator is translated as in the proof of theorem 6.4. The number of iterations is at most polynomial, so a list L whose length is the upper bound on the number of iterations can be constructed using the expression $L = \Lambda(\mathrm{MAP}_{[\,]}(\boldsymbol{Q} \times \ldots \times \boldsymbol{Q}))$, where $\boldsymbol{Q}$ is a relation containing all the objects in the active domain, and the Cartesian product is applied $\boldsymbol{k}$ times where $\boldsymbol{k}$ is the exponent of the polynomial bound of the computation. Then the fixpoint is computed by applying the iterator on the list.

- For $\exists \boldsymbol{x} \boldsymbol{\phi}_1(\boldsymbol{x})$, where $\boldsymbol{x}$ is of constant type (i.e. not integer) we project out the column corresponding to $\boldsymbol{x}$.

- For $\exists \boldsymbol{x} \boldsymbol{\phi}_1(\boldsymbol{x})$, where $\boldsymbol{x}$ is of integer type we use $I_{E_1(\{\}), E_1(x), E_1(\oplus^{-1}(y)) \bigcup z}(L)$, Were L is the list of polynomial length constructed as above. The idea is to iterate over all the integers in the domain, compute the formula for each of them, and take the union of the result.

$\square$

Lemma 6.7 $\mathcal{P}\text{om-}\mathcal{A}\text{lg}^b|_{u,t} \subseteq$ FO+FP+C

Before going into the details of the proof, we consider a simple example of query in $\mathcal{P}$om-$\mathcal{A}\text{lg}^b|_{u,t}$.

Example 6.2 The following query computes the if-then-else, that is if R_1 is not empty then R_2, and otherwise R_3.

$$\text{IfThenElse}(R_1, R_2, R_3) \ \equiv \ (\ =_{\emptyset}(=_{\emptyset}(R_1)) \times R_2 \ ) \ \cup \ (\ =_{\emptyset}(R_1) \times R_3 \ )$$

Its calculus equivalent expression is:

$$\{ \ \boldsymbol{y} \ \mid \ (\exists \boldsymbol{x} R_1(\boldsymbol{x}) \wedge R_2(\boldsymbol{y})) \vee (\neg \exists \boldsymbol{x} R_1(\boldsymbol{x}) \wedge R_3(\boldsymbol{y}))\}$$

If R_1 is empty $=_{\emptyset}(R_1) = \{\![\ ]\!\}$, and $=_{\emptyset}(=_{\emptyset}(R_1)) = \}\!\{$, and conversely if R_1 is not empty.

Proof of Lemma 6.7: We prove the lemma by induction on the structure of the algebraic expression, by presenting a translation for every algebra operator. We represent bags using relations. The tuples in the relation are of the form $[\boldsymbol{i}, \boldsymbol{a}_1, \ldots, \boldsymbol{a}_n]$ where $\boldsymbol{i}$ is the number of occurrences of the tuple $[\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n]$ in the bag. We also represent lists using relations. The tuples in the relations are of the form $[\boldsymbol{i}, \boldsymbol{a}_1, \ldots, \boldsymbol{a}_n]$ where $\boldsymbol{i}$ is the index of the tuple $[\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n]$ in the list.

Basis: The algebra expression $\{\!\boldsymbol{e}\!\}$ (where $\boldsymbol{e}$ is a constant), is translated to $Q = \{[\boldsymbol{i}, \boldsymbol{x}] \mid \boldsymbol{x} = \boldsymbol{e} \wedge \boldsymbol{i} = 1\}$. The expression $\{\!\ \!\}$ is translated to $Q = \{\boldsymbol{x} \mid \boldsymbol{False}\}$. A relation R is translated to $Q = \{\boldsymbol{i}, \overline{\boldsymbol{x}} \mid R(\overline{\boldsymbol{x}}) \wedge \boldsymbol{i} = 1\}$ (where $\overline{\boldsymbol{x}}$ is a vector of variables of the arity of R).

Note that addition and multiplication can be computed in the calculus using the FP operator. For brevity we omit below the encoding and simple use $+$ and $\times$.

Induction: Assume that E_1, E_2 are algebra expressions, and ϕ_1, ϕ_2 are the corresponding calculus queries. Let E be an algebra query that is constructed from E_1, E_2. We show how to construct ϕ from ϕ_1, ϕ_2. We first consider the case where E_1, E_2 construct bags.

$\uplus$, $-$, $\bigcup$, $\bigcap$ **:** We change the two given formulas, if necessary, so that they have the same free variables.

If $E = E_1 \uplus E_2$, then: $\phi = \{\boldsymbol{i}, \overline{\boldsymbol{x}} | \exists \boldsymbol{k}_1, \boldsymbol{k}_2(\phi_1(\boldsymbol{k}_1, \overline{\boldsymbol{x}}) \vee (\boldsymbol{k}_1 = 0 \wedge \forall \boldsymbol{n} \neg \phi_1(\boldsymbol{n}, \overline{\boldsymbol{x}})))$
. $\wedge(\phi_2(\boldsymbol{k}_2, \overline{\boldsymbol{x}}) \vee (\boldsymbol{k}_2 = 0 \wedge \forall \boldsymbol{n} \neg \phi_2(\boldsymbol{n}, \overline{\boldsymbol{x}}))) \wedge (\boldsymbol{k}_1 + \boldsymbol{k}_1 > 0) \wedge (\boldsymbol{i} = \boldsymbol{k}_1 + \boldsymbol{k}_2)\}$.
The translation for $-$, $\bigcup$, and $\bigcap$ is similar except that $-$, $\boldsymbol{min}$, and $\boldsymbol{max}$ are used respectively instead of addition. ($\boldsymbol{min}$ and $\boldsymbol{max}$ can be described using calculus formulas).

$\oplus$ **:** We first change the two given formulas, if necessary, so that they have the same free variables. This operation has a non empty outpot in $\mathcal{P}$om-$\mathcal{A}$lg$^b|_{u,t}$ if either both E_1 and E_2 are singleton pomsets, or when at least on of the two is empty. Otherwise the result is the empty pomset. This can be easily checked in the calculus (omitted). Let φ_1, φ_2 be the formulas testing emptiness and being a singleton pomset resp. Then

$$\phi = \{\boldsymbol{i}, \overline{\boldsymbol{x}} | (\varphi_1(\phi_1) \wedge \phi_2(\boldsymbol{i}, \overline{\boldsymbol{x}}) \vee (\varphi_1(\phi_2) \wedge \phi_1(\boldsymbol{i}, \overline{\boldsymbol{x}}) \vee (\varphi_2(\phi_1) \wedge \varphi_2(\phi_2) \wedge ((\phi_1(\boldsymbol{i}, \overline{\boldsymbol{x}}) \vee (\phi_2(1, \overline{\boldsymbol{x}}) \wedge \boldsymbol{i} = 2)))\}.$$

$\times$: If $E = E_1 \times E_2$, we change the two formulas so that their free variables are distinct. Then: $\phi = \{i, \overline{x}, \overline{y} | \exists k_1 \exists k_2 \phi_1(k_1, \overline{x}) \wedge \phi_2(k_2, \overline{y}) \wedge (i = k_1 \times k_2)\}$.

ϵ_n, ϵ: Since we are dealing with bags, the two operators have the same effect. Thus the expressions $E = \epsilon_n(E_1)$ and $E = \epsilon(E_1)$ are both translated to $\phi = \{1, \overline{x} | \exists j \phi_1(j, \overline{x})\}$.

$=_\emptyset$: If ϕ_1 has free variables $x_1, \ldots, x_n$ then $=_\emptyset(E_1)$ is translated to $\neg\exists x_1, \ldots, x_n\ \phi_1(x_1, \ldots, x_n)$. Else, if ϕ_1 has no free variable it is translated to $\neg\phi_1$.

MAP : This operation is the most difficult to encode. The number of occurrences of a tuple $\overline{x}$ in $\mathrm{MAP}_{E_2}(E_1)$, is the sum over all the tuples $\overline{y}$ such that $\overline{x} \in E_2(\overline{y})$, of the number of occurrences of $\overline{y}$ times the number of occurrences of $\overline{x}$ in $E_2(\overline{y})$. We sum the number of occurrences using the fixpoint operator. The idea is to count for every two integers i, j and every tuple $\overline{x}$ how many tuples $\overline{y}$ with i occurrences in E_1 have j occurrences of $\overline{x}$ in $E_2(\overline{y})$. Every k such tuples contribute to the result $i \times j \times k$ occurrences of $\overline{x}$. We use the fixpoint operator to perform this iteration over i and j. At each iteration we increment i, and then iterate over j counting how many tuples $\overline{y}$ for which $\overline{x}$ appears j times in $E_2(\overline{y})$ have i occurrences in E_1, and add $i \times j \times k$ to the result of the previous step.

Observe that for each $\overline{x}$ we only need to consider i values that are no greater than the maximal number of occurrences of $\overline{y}$ for some $\overline{y}$. Also observe that the size of $E_2(\overline{y})$ is bounded by a constant c_{E_2} that depends only on the operations in E_2. (This is because the input is always a singleton set, and $\mathcal{P}$om-$\mathcal{A}$lg^b increases the size of its input polynomialy (see proposition 5.2). Thus we only have to consider j's not greater than c_{E_2}.

Since the fixpoint operator is inflationary, all intermediate results are accumulated. We thus use one attribute as an index denoting the iteration number: We use the FP operator to build a relation R with tuples of the form $(i, j, s\overline{x})$ where i, j denote the index of the iterations in the two loops and for a given $\overline{x}$ $s = \Sigma_{p=1 \ldots i-1} \Sigma_{q=1 \ldots c_{E_2}} p \times q \times k + \Sigma_{q=1 \ldots j} i \times q \times k$.

Finally, to compute $\mathrm{MAP}_{E_2}(E_1)$ we take the resulting relation and select the tuples with the highest i, j components.

The formula ϕ computing $\mathrm{MAP}_{E_2}(E_1)$ is presented below. Observe that E_2 has the form $\lambda x.f(x)$, thus its corresponding formula is of the form $\phi_2(k, \overline{x}, \overline{y})$ where $\phi_2(k, \overline{x}, \overline{y})$ is true iff $\overline{x}$ has k occurrences in $E_2(\overline{y})$.

$$\phi = \{s, \overline{x} | \exists i, j\ FP_R(\varphi(R))(i, j, s, \overline{x}) \wedge i = maxcounter(\overline{x}) \wedge j = c_{E_2}\}$$

where $\varphi(R) = \{i, j, s, \overline{x}\ |$

%%% $\boldsymbol{x}$, $\boldsymbol{i}$, and $\boldsymbol{j}$ are legal

$(\exists \boldsymbol{k}_1, \boldsymbol{k}_2, \overline{\boldsymbol{y}}\ \phi_1(\boldsymbol{k}_1, \overline{\boldsymbol{y}}) \wedge \phi_2(\boldsymbol{k}_2, \overline{\boldsymbol{x}}, \overline{\boldsymbol{y}}))\ \wedge(\boldsymbol{i} \leq \boldsymbol{maxcounter}(\overline{\boldsymbol{x}})) \wedge (\boldsymbol{j} \leq c_{E_2})$

%%% the iteration step

$\wedge($

%%% first iteration

$(\boldsymbol{R} = \emptyset \wedge \boldsymbol{i} = 1 \wedge \boldsymbol{j} = 1 \wedge \boldsymbol{s} = \boldsymbol{count}(1, 1, \overline{\boldsymbol{x}}))$

%%% subsequent iteration

$\vee($

%%% take the result of the previous iteration

$\exists \boldsymbol{i}', \boldsymbol{j}', \boldsymbol{s}'\ \boldsymbol{R}(\boldsymbol{i}', \boldsymbol{j}', \boldsymbol{s}'\overline{\boldsymbol{x}}) \wedge (\neg \exists \boldsymbol{i}''\boldsymbol{j}'', \boldsymbol{s}''\ \boldsymbol{R}(\boldsymbol{i}'', \boldsymbol{j}'', \boldsymbol{s}''\overline{\boldsymbol{x}}) \wedge ((\boldsymbol{i}' < \boldsymbol{i}'') \vee (\boldsymbol{i}' = \boldsymbol{i}'' \wedge \boldsymbol{j}' < \boldsymbol{j}'')))$

%%% compute the indexes of the new iteration

$\wedge((\boldsymbol{j}' = c_{E_2} \wedge \boldsymbol{j} = 1 \wedge \boldsymbol{i} = \boldsymbol{i}' + 1) \vee (\boldsymbol{j} = \boldsymbol{j}' + 1 \wedge \boldsymbol{i} = \boldsymbol{i}))$

%%% compute the new sum

$\wedge \boldsymbol{s} = \boldsymbol{s}' + \boldsymbol{count}(\boldsymbol{i}, \boldsymbol{j}, \overline{\boldsymbol{x}})))\}$

$\boldsymbol{maxcounter}(\overline{\boldsymbol{x}}) = \{\boldsymbol{i}|$

$\exists \overline{\boldsymbol{y}}, \boldsymbol{k}\ (\phi_1(\boldsymbol{i}, \overline{\boldsymbol{y}}) \wedge (\phi_2(\boldsymbol{k}, \overline{\boldsymbol{x}}, \overline{\boldsymbol{y}}) \wedge \neg \exists \boldsymbol{j}, \overline{\boldsymbol{y}'}, \boldsymbol{k}'(\phi_1(\boldsymbol{j}, \overline{\boldsymbol{y}'}) \wedge (\phi_2(\boldsymbol{k}', \overline{\boldsymbol{x}}, \overline{\boldsymbol{y}}) \wedge \boldsymbol{i} < \boldsymbol{j}))$

and

$\boldsymbol{count}(\boldsymbol{i}, \boldsymbol{j}, \overline{\boldsymbol{x}}) = \{\boldsymbol{s} \mid$

$(\exists \boldsymbol{k}\ \#_{\overline{y}}(\phi_1(\boldsymbol{i}, \overline{\boldsymbol{y}}) \wedge \phi_2(\boldsymbol{j}, \overline{\boldsymbol{x}}, \overline{\boldsymbol{y}})) = \boldsymbol{k} \wedge \boldsymbol{s} = \boldsymbol{i} \times \boldsymbol{j} \times \boldsymbol{k}) \bigvee (\neg \exists \overline{\boldsymbol{y}}(\phi_1(\boldsymbol{i}, \overline{\boldsymbol{y}}) \wedge \phi_2(\boldsymbol{j}, \overline{\boldsymbol{x}}, \overline{\boldsymbol{y}}) \wedge \boldsymbol{s} = 0);$

$\oplus^{-1}$: Order-destroy has no effect on bags. Thus $\phi = \{\boldsymbol{i}, \overline{\boldsymbol{x}} | \phi_1(\boldsymbol{i}, \overline{\boldsymbol{x}})\}$.

Λ: For $E = \Lambda(E_1)$ to be a list, E_1 must contain multiple occurrences of a single element, i.e. $\phi_1 = (\boldsymbol{i}, \overline{\boldsymbol{a}})$ for some integer $\boldsymbol{i}$ and object $\overline{\boldsymbol{a}}$. In this case, ϕ should represent a list of $\boldsymbol{i}\ \overline{\boldsymbol{x}}$'s. Thus $\phi = \{\boldsymbol{j}, \overline{\boldsymbol{x}} | (\neg \exists \boldsymbol{i}_1, \overline{\boldsymbol{x}_1}, \boldsymbol{i}_2, \overline{\boldsymbol{x}_2} \phi_1(\boldsymbol{i}_1, \overline{\boldsymbol{x}_1}) \wedge \phi_1(\boldsymbol{i}_2, \overline{\boldsymbol{x}_2}) \wedge \overline{\boldsymbol{x}_1} \neq \overline{\boldsymbol{x}_2}) \wedge (\exists \boldsymbol{i}\ \phi_1(\boldsymbol{i}, \overline{\boldsymbol{x}}) \wedge \boldsymbol{j} \leq \boldsymbol{i})\}$

I: On bags $I_{E_2, E_3, E_4 | E_5}(E_1) = E_3(E_1)$. We translate $E_3(E_1)$ as explained above (and below).

We next consider the case where E_1 and E_2 construct lists.

$\uplus$: The operator cannot be applied unless one of the arguments is the empty list. In this case, the result equals the other argument. $\phi = \{\overline{\boldsymbol{v}} | (\neg \exists \overline{\boldsymbol{v}'} \phi_2(\overline{\boldsymbol{v}'}) \wedge \phi_1(\overline{\boldsymbol{v}})) \vee (\neg \exists \overline{\boldsymbol{v}'} \phi_1(\overline{\boldsymbol{v}'}) \wedge \phi_2(\overline{\boldsymbol{v}}))\}$

$-$: For lists, the operator tests if the two input lists are identical if so the result is empty, otherwise it is equal to the first argument. $\phi = \{\overline{\boldsymbol{v}} | \phi_1(\overline{\boldsymbol{v}}) \wedge (\neg \forall \overline{\boldsymbol{v}'}\ \phi_1(\overline{\boldsymbol{v}'}) \leftrightarrow \phi_2(\overline{\boldsymbol{v}'}))\}$

$\bigcup$: The operator cannot be applied unless the two arguments are equivalent. In this case the result equals to one of the argument. The calculus formula is similar to the one above (omitted).

$\bigcap$: For lists, the operator tests if the two input lists are identical if so returns one of them, otherwise the result is empty. The formula is similar to the ones above (omitted).

$\oplus$: If $E = E_1 \oplus E_2$, then $\phi = \{i, \overline{x} | \phi_1(i, \overline{x}) \vee (\exists j \phi_2(j, \overline{x}) \wedge i = j + \#_{\overline{x}}(\phi_1(\overline{x})))\}$.

$\times$: If $E = E_1 \times E_2$, then $\phi = \{i, \overline{x}, \overline{y} | \exists i_1, i_2\ \phi_1(i_1, \overline{x}) \wedge \phi_2(i_2, \overline{y}) \wedge i = i_1 \times i_2\ \}$.

ϵ_n, ϵ: The operator ϵ has no effect on its input list. ϵ_n can be applied only if E_1 is a list of the form $x_1 \ldots x_1 . x_2 \ldots x_2 \ldots x_n \ldots x_n$ where all x_i are distinct - in which case the result is a list $x_1 . x_2 \ldots x_n$, or if E_1 is a list where for every two constants x_i, x_2 in the list x_i appears both before and after some occurrence of some x_j - in which case the result is a bag with one occurrence of each x_i.

It is easy to define a calculus formulas testing which is the case. (omitted).

In the first case we use $\phi' = \{i, \overline{x} | \phi_1(i, \overline{x}) \wedge \neg \exists j\ \phi_1(j, \overline{x}) \wedge j < i\}$ to select the first occurrence of each x_i and then use the FP operator to replace the indexes of the x_1 by $i \ldots n$. (Omitted).

For the second case we use $\{1, \overline{x} | \exists j\ \phi_1(j, \overline{x})\}$.

MAP: To simulate $\mathrm{MAP}_{E_2}(E_2)$ we use the fixed point operator. We sketch the idea below. We iterate over the elements in the list and at the ith iteration take the ith element in the list (encoded as $[i, \overline{y}]$) and replace it by $\{i', \overline{x} | \phi_2(i'', \overline{x}, \overline{y}) \wedge i = k + i''\}$ were k is a counter counting how many tuples were already inserted into the result list in the previous iterations. The new value of k is obtained by adding to the current k the length of the list $E_2(y)$, i.e. the number $k' = \#_{k'', \overline{x}}(\phi_2(k'', \overline{x}, \overline{y}))$.

As in the translation of MAP for the case of bags (detailed above) we have to take into account that the FP operator is inflationary and have a special index attribute recording the iteration number. The result is then obtained by selecting the tuples having the highest index.

$\oplus^{-1}$: If $E = \oplus^{-1}(E_1)$, then the result is a bag defined by $\phi = \{i, \overline{x} | \#_{\overline{y}}(\exists j \phi_1(j, \overline{x}) \wedge \overline{y} = \overline{x}) = i\}$.

Λ: Listing can be applied, having a non empty result, only when E_1 has at most one element. In this case it is simply the identity function.

I: Let $E = I_{E_2, E_3, E_4 | E_5}(E_1)$, and let $\phi_1(i, \overline{x})$, $\phi_2(i, \overline{x})$, $\phi_3(i, \overline{x}, \overline{y})$, $\phi_4(i, \overline{x}, j, \overline{y}, k\overline{z})$, $\phi_5(i, \overline{x})$ be formulas such that for every $i, j, k, \overline{x}, \overline{y}, \overline{z}$, $\phi_1(i, \overline{x})$ is true iff $\overline{x}$ is the ith tuple in the list constructed by E_1, $\phi_2(i, \overline{x})$ is true iff either E_2 constructs a bag and $\overline{x}$ occurs i times in the bag, or E_2 constructs a list and $\overline{x}$ is the ith tuple in the list, $\phi_3(i, \overline{x}, \overline{y})$ is true iff either $E_3(\overline{y})$ constructs a bag and $\overline{x}$ occurs i times in the bag, or it constructs a list and $\overline{x}$ is the ith tuple in the list, $\phi_4(i, \overline{x}, \overline{y}, P_1, P_2)$ is true iff P_1 is a relation encoding a list, P_2 is a relation encoding a list or a bag and either $E_3(\overline{y}, P_1, P_2)$ constructs a bag and $\overline{x}$ occurs i times in the bag, or it constructs a list and $\overline{x}$ is the ith tuple in the list, $\phi_5(i, \overline{x})$ is true iff either E_5 constructs a bag and $\overline{x}$ occurs i times in the bag, or it constructs a list and $\overline{x}$ is the ith tuple in the list. The formulas $\phi_1, \ldots, \phi_5$ can be constructed based on the induction assumption. To simulate the restriction on output size (imposed by the bounded iterator) we define a formula $\phi'(i, \overline{x}, \overline{y}, P_1, P_2)$ that computes ϕ_4, tests if the size of the output is larger than that of ϕ_5, and if so returns P_2. The size of the output pomsets are computed as follows: If the output is a list than the size is simple the highest i such that $[i, \overline{x}]$ is in the output (for some $\overline{x}$). If the output is a bag that the size is computed by summing up all the i such that $[i, \overline{x}]$ is in the output. The sum is computed using the FP operator, with a technique similar to that used in the translation of MAP above. (Details omitted).

The iterator is now simulated using the fixpoint operator, using the formulas ϕ_1, ϕ_2, ϕ_3 and ϕ'. Since the fixpoint operator is inflationary, all intermediate results are accumulated. We thus use one attribute as an index recoding the iteration number, and then select the tuples with highest index. Note that the number of iterations need to be performed by the FP operator is bounded by the length of the list constructed in E_1. We start with length equals to the highest i satisfying $\phi_1(i, \overline{x})$, and decrease it repeatedly. At each iteration, the relation P_1 contains the tuples in E_1 corresponding to list members with index greater than the current length (The i component of the tuples is replaced by $i - \mathit{current\_lenght}$ to reflect the relative location of the tuples in the list.) The relation P_2 is the result of the previous iteration step. The output of the current step is computed using the formula ϕ'.

The construction of the FP formula is very similar to the one used for encoding the MAP operator above, and is left as an exercise to the reader.

□

Proof of Theorem 6.5: We considered above cases where E_1, E_2 are of bag/list type, and explained the possible typing of the results in each case, and how this case can be checked by calculus formulas. To translate a formula, we follow the above translation steps and at each intermediate result we use a disjunction over all the possibilities. This concludes the proof. □

A computational model characterizing FO+FP+C was introduced in [GO93] as a Turing machine with a relational store inspired from [AV91]. There are some tight connections between the computation on the Turing tape of the computational device and the expressions manipulating lists in $\mathcal{P}$om-$\mathcal{A}$lg$|_{u,t}$.

It is open if the use of general partially ordered types with duplicates further increases the expressiveness of the language. We conjecture that: $\mathcal{P}\text{om-}\mathcal{A}\text{lg}^b|_{u,t} \subset \mathcal{P}\text{om-}\mathcal{A}\text{lg}^b$. More generally, the expressive power resulting from non series-parallel pomsets is badly understood. We conjecture that the query ***source***, giving the objects with no predecessor, on N-shaped graphs cannot be expressed in $\mathcal{P}$om-$\mathcal{A}$lg.

6.3 Ordered Input/Output

To complete previous works that considered the impact of totally ordered domains on the expressive power of the language [Imm86, Var82], we study here the effect of having a total internal order. We show that $\mathcal{P}$om-$\mathcal{A}$lg^b expresses all PTIME queries on ordered types with total internal order and duplicates. It is open if the same result holds for the other tractable version of $\mathcal{P}$om-$\mathcal{A}$lg, $\mathcal{P}$om-$\mathcal{A}$lg$^\epsilon$.

Theorem 6.8 For input/output of type list, $\mathcal{P}$om-$\mathcal{A}$lg^b=PTIME.

Proof: The inclusion in PTIME follows from Theorem 5.2. The other direction is proved by showing that the computation of a polynomial time Turing machine M can be simulated by the algebra. The proof requires three steps: (i) encoding of the input on the tape, (ii) simulation of the computation of the TM, mapping the encoded input to an encoded output, and finally (iii) decoding of the output. All three steps have to be done in the algebra.

Encoding/Decoding

We consider a Turing machine whose input and output tape are lists of the form $\boldsymbol{enclist}(\boldsymbol{l})$ defined below, for a list $\boldsymbol{l}$. Without loss of generality, we assume below that $\boldsymbol{l}$ is a list of tuples[5]. $\boldsymbol{enclist}(\boldsymbol{l})$ is the following encoding. If $\boldsymbol{l}$ is a list containing $\boldsymbol{m}$ $\boldsymbol{n}$-ary tuples $[\boldsymbol{a}_1^1, \ldots, \boldsymbol{a}_n^1] \ldots [\boldsymbol{a}_1^m, \ldots, \boldsymbol{a}_n^m]$, then $\boldsymbol{enclist}$ encodes $\boldsymbol{l}$ into a string over the TM's alphabet $\{0, [,], \#\}$ as follows:

$$\boldsymbol{enclist}(\boldsymbol{l}) = [\boldsymbol{enc}(\boldsymbol{a}_1^1)\# \ldots \#\boldsymbol{enc}(\boldsymbol{a}_n^1)] \ldots [\boldsymbol{enc}(\boldsymbol{a}_1^m)\# \ldots \#\boldsymbol{enc}(\boldsymbol{a}_n^m)].$$

where $\boldsymbol{enc}$ is a function encoding distinct constants by sequences of 0's of distinct lengths.

We explain below how the encoding of an arbitrary list $\boldsymbol{l}$ into a list representing the TM tape is done. The encoding process has three steps.

1. First, we "flatten" the list of tuples and make it a list of atomic elements. For that, every tuple in the list is replaced by a list representing it. Let "[", "]", and "#" be new atomic symbols. The flattened version $\boldsymbol{l}_f$ of $\boldsymbol{l}$ is constructed by

   $$\boldsymbol{l}_f = \boldsymbol{I}_{\emptyset,\lambda x.g(x),\lambda xyz.g(x)\oplus z|l}(\boldsymbol{l})$$

   where

   $$\boldsymbol{g}(\boldsymbol{x}) = \{[\} \oplus \{\boldsymbol{x}.1\} \oplus \{\#\} \oplus \ldots \{\boldsymbol{x}.\boldsymbol{n}\} \oplus \{]\}$$

2. Next, we build list $\boldsymbol{c} = \boldsymbol{a}_1 \ldots \boldsymbol{a}_i$ of constants, containing all the atomic elements, besides [,], and #, in $\boldsymbol{l}_f$, in the order of last occurrence. $\boldsymbol{c}$ is constructed by

   $$\boldsymbol{c} = \boldsymbol{I}_{\emptyset,\lambda x.\emptyset,\lambda xyz.\mathrm{IfThenElse}(\mathrm{IN}(y,z),z,z\oplus y)|l}(\boldsymbol{l})$$

   where IfThenElse is defined as in example 6.2, and IN is defined as follows

   $$\mathrm{IN}(\boldsymbol{a}, \boldsymbol{p}) = \boldsymbol{\sigma}_{\lambda x.a=x}(\boldsymbol{p} \oplus \{[\} \oplus \{]\} \oplus \{\#\})$$

[5] A list of atomic elements (1-ary tuples) can be easily constructed from a list of 1-ary tuples (atomic elements) using the MAP operator.

3. Finally, we construct $\boldsymbol{enclist}(\boldsymbol{l})$ by replacing each occurrence of an atomic element $\boldsymbol{a}_j$ in $\boldsymbol{l}_f$, by $\boldsymbol{j}$ consecutive 0's, (where $\boldsymbol{j}$ denotes the index of the element in the list $\boldsymbol{c}$ constructed above). This is done by carrying a (bounded) iteration over $\boldsymbol{l}_f$. Let @, & be two new symbols. At the first iteration step we build the list $\boldsymbol{a}_1 @ 0 @ \boldsymbol{a}_2 @ 00 @ \ldots @ \boldsymbol{a}_i @ 0^i \& \boldsymbol{l}_f$, containing the constants in $\boldsymbol{c}$, their encoding as lists of 0's, and the flattened list $\boldsymbol{l}_f$. (This combined list is build using nested iteration on $\boldsymbol{c}$).

   Then, at each iteration step $\boldsymbol{j} > 1$, we construct a list of the form $\boldsymbol{a}_j @ 0^j @ \ldots \boldsymbol{a}_i @ 0^i \& \boldsymbol{l}_f^j$, where $\boldsymbol{l}_f^j$ is the list obtained from $\boldsymbol{l}_f^{j1}$ by replacing each occurrence of the constant $\boldsymbol{a}_{j-1}$ by its encoding, i.e. by a list of $\boldsymbol{j} - 1$ zeros. (This is accomplished using nested (bounded) iteration and MAP).

   The list obtained at the end of the above iteration has the form & $\boldsymbol{enclist}(\boldsymbol{l})$. The encoding task is completed by removing the & symbol at the head.

   The expressions realizing this iteration is long and but straight forward to construct and is thus omitted.

The decoding process is analogous – First, every (maximal) subsequence 0^j of 0's is replaced by the corresponding constant $\boldsymbol{a}_j$. Then the list is "un-flattened" to obtain a list of tuples.

TM Simulation

We next show that the iterator I can be used to simulate the computation of the Turing machine $\boldsymbol{M}$. The input of each iteration step is a list encoding the current tape of the Turing machine, the location of the head, and the state. The output of the step is a list encoding the tape of the Turing machine after one additional move, the new state, and the new location of the head. The fact that the content of the tape is $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_k$, the state $\boldsymbol{q}$, and the head is on the $\boldsymbol{i}$ cell, is simply represented by the list $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_{i-1}, \boldsymbol{q}, \boldsymbol{c}_i, \ldots, \boldsymbol{c}_k$.

The length of the computation being simulated using this technique depends on the length of the list on which the iterator is applied. A list of polynomial length can be constructed using the Cartesian product on the initial list. Thus any PTIME computation can be simulated by applying the iterator on such lists.

Assume we have a list $\boldsymbol{l}' = \boldsymbol{enclist}(\boldsymbol{l})$ for some $\boldsymbol{l}$. Let $\boldsymbol{q}_0, \ldots, \boldsymbol{q}_n$ be the states of the TM, and let $\boldsymbol{q}_0$ be the starting state. The $\mathcal{P}$om-$\mathcal{A}$lg^b formula below simulates a TM computation on $\boldsymbol{l}'$ of time $|\boldsymbol{l}'|^k$ for some arbitrary $\boldsymbol{k}$.

$$TM(\boldsymbol{l}') = I_{\{q_0\} \oplus l', \lambda x.\mathrm{MOVE}(\{q_0\} \oplus l'), \lambda x\ y\ z.\mathrm{MOVE}(z) | (l' \times)^{k-1} l'}((\boldsymbol{l}' \times)^{k-1} \boldsymbol{l}')$$

Where

$$\mathrm{MOVE}(\boldsymbol{p}) = \mathrm{MOVE}_{q_0, a_0} \oplus \mathrm{MOVE}_{q_0, a_1} \oplus \ldots \oplus \mathrm{MOVE}_{q_n, a_m}$$

$\boldsymbol{a}_0, \ldots, \boldsymbol{a}_m$ are the alphabet of the tape. MOVE_{q_i, a_j} tests if the TM is in state $\boldsymbol{q}_i$ with the head on the letter $\boldsymbol{a}_j$. If so it simulates one step of the TM, and otherwise returns an empty list. We show below how to define MOVE_{q_i, a_j} for the case of a right move where $\boldsymbol{\delta}(\boldsymbol{q}_i, \boldsymbol{a}_j) = (\boldsymbol{q}_i', \boldsymbol{a}_j', \boldsymbol{R})$. The case of a left move or no move (for final states) is handled similarly.

$\mathrm{MOVE}_{q_i,a_j}(\boldsymbol{p}) =$
$\quad \mathrm{IfThenElse}(\mathrm{FIND}_{q_i,a_j}(\boldsymbol{p}), \mathrm{BEFORE}_{q_i}(\boldsymbol{p}) \oplus \{\boldsymbol{a}'_j\} \oplus \{\boldsymbol{q}'_i\} \oplus \mathrm{DEL\_FIRST}(\mathrm{AFTER}_{q_i}(\boldsymbol{p})), \emptyset)$

Where

IfThenElse is defined as in example 6.2, and

$\mathrm{FIND}_{q_i,a_j}(\boldsymbol{p}) = \Pi_{[\ ]}(\boldsymbol{\sigma}_{\lambda x.x=a_j}(\mathrm{FIRST}(\mathrm{AFTER}_{q_i}(P)))$

$\mathrm{AFTER}_{q_i}(\boldsymbol{p}) = \mathrm{INVERSE}(\mathrm{BEFORE}_{q_i}(\mathrm{INVERSE}(\boldsymbol{p})))$

$\mathrm{INVERSE}(\boldsymbol{p}) = I_{\emptyset,\lambda x.x,\lambda x\ y\ z.z\oplus x}(\boldsymbol{p})$

$\mathrm{BEFORE}_{q_i}(\boldsymbol{p}) = \mathrm{DEL\_LAST}(I_{\emptyset,\lambda x.\sigma_{\lambda u.u=q_i}(x),\lambda x\ y\ z.\mathrm{IfThenElse}(\sigma_{\lambda u.u=q_i}(x),x,\mathrm{IfThenElse}(\epsilon_n(\Pi_{[\ ]}(z)),x\oplus z,\emptyset))})$

$\mathrm{DEL\_LAST}(\boldsymbol{p}) = I_{\emptyset,\emptyset,\lambda x\ y\ z.x\oplus z}$

$\mathrm{DEL\_FIRST}(\boldsymbol{p}) = \mathrm{INVERSE}(\mathrm{DEL\_LAST}(\mathrm{INVERSE}(\boldsymbol{p})))$

$\mathrm{FIRST}(\boldsymbol{p}) = I_{\emptyset,\lambda x.x,\lambda z\ y\ z.z}(\mathrm{INVERSE}(\boldsymbol{p}))$

This concludes the proof.
□

Note that unlike the alternative language for polynomial time list manipulation [CRSG94], the previous algebra does not require the use of nested structures, i.e. lists of tuples of lists. In contrast, our result shows that all PTIME queries can be expressed even when only flat lists are available.

Note that the inclusion in PTIME is due to the use of the bounded iterator. If non-bounded iterator is used, we obtain all elementary functions. Let $\mathcal{E}$, denotes the set of ***elementary*** queries, i.e. of hyper-exponential complexity.

Theorem 6.9 $\mathcal{P}$om-$\mathcal{A}$lg= $\mathcal{E}$, for input/output of type list.

Proof: As in Theorem 6.8 above, we simulate the computation of a TM on the encoded list, and then decode the output, with two minor changes. First we use the general iterator and not the bounded version. Second, to simulate a hyper-exponential computation we need to iterate over a list of hyper-exponential length. Such a list is constructed as follows. Given a list $\boldsymbol{l}$ with $\boldsymbol{n}$ arbitrary members, the expression below constructs a list of length $O(2^n)$.
$E(\boldsymbol{l}) = I_{\emptyset,\lambda x.\{a\},\lambda x\ y\ z.z\oplus z}(\boldsymbol{l})$

Then, a list of length $\boldsymbol{hyper(i)(n)}$ is constructed using
$L(p) = E^i(l)$.

The second direction follows from the fact that every $\mathcal{P}$om-$\mathcal{A}$lg operation increases at most exponentially the size of its input.

□

7 Nesting

We next consider ordered types with nesting. Nesting of structures has been extensively studied in the context of unordered types like sets [HS88, HS89, GV90, GV91] and bags [GM93]. In this section, we study the combined effect of order, nesting, and duplicates. Nested pomsets can be used to represent ordered types with components of some ordered type (e.g. lists of trees). In nested pomsets, every object can be a nested pomset or a tuple with attributes that are nested pomsets. It turns out that nesting increases drastically the expressive power of the language. This is due to the operations linking different levels of nesting.

All the $\mathcal{P}$om-$\mathcal{A}$lg operators presented in Section 3 are extended naturally to handle nested pomsets. The algebra has one additional operator. Let $p = \langle V, \Sigma, \mu, < \rangle$ be a pomset of type $\{T\}$.

- ***Powerset,*** $\mathcal{P}$: Intuitively, $\mathcal{P}(p)$ computes the set of all sub-pomsets of p. More formally, $\mathcal{P}(p) = \langle V_{\mathcal{P}}, \Sigma_{\mathcal{P}}, \mu_{\mathcal{P}}, \emptyset \rangle$ is an unordered pomset of type $\{\{T\}\}$, where $\mu_{\mathcal{P}}$ is a 1-1 function, and where for every[6] pomset $p' = \langle V', \Sigma, \mu', <' \rangle$, where $V' \subseteq V$, $\mu' = \mu \downarrow_{V'}$, and $<' \subseteq < \downarrow_{V'}$, there exists $v \in V_{\mathcal{P}}$ such that $\mu_{\mathcal{P}}(v) = p'$.

As in the nested relation (bag) algebra, the powerset operator may cause exponential growth of the output, and the resulting language is no longer tractable. We therefore remove the restrictions from the iterator and the other operations and consider the full pomset algebra.

Note that the nested relation (bag) algebra has an additional operator, ***set(bag)-destroy***, that is used to destroy one level of set (bag) nesting [AB87, GM93]. This operation can be simulated in $\mathcal{P}$om-$\mathcal{A}$lg using the MAP operator. In particular, $\mathrm{MAP}_{\lambda x.x}$ destroys one level of pomset nesting. For every pomset p of type $\{\{T\}\}$, $\mathrm{MAP}_{\lambda x.x}(p)$ is a pomset of type $\{T\}$ obtained by expanding each $v \in V$ in p into a whole pomset $p_v = \mu(v)$, preserving order both within and between the p_v's.

In the rest of this section, we study the effect that different levels of nesting have on the expressive power of $\mathcal{P}$om-$\mathcal{A}$lg. A complex type can be viewed as a tree with nodes representing the tuple and pomset constructors. The **nesting** of a type T is the maximal number of pomset nodes in a path from the root to a leaf. We denote the algebra when restricted to pomsets with nesting of depth k, $\mathcal{P}$om-$\mathcal{A}$lg^k. The simple algebra investigated

[6]Note that since pomsets define isomorphism classes of lpo's, two isomorphic lpo's will have only one occurrence in $\mathcal{P}(p)$.

in previous sections allows only one level of pomsets, thus corresponds to $\mathcal{P}$om-$\mathcal{A}$lg^1. We next consider $\mathcal{P}$om-$\mathcal{A}$lg^2, one stage higher in the nesting hierarchy.

We study the relationship between the nested pomset algebra and the nested relation and nested bag algebras. We denote the nested relation (bag) algebra when restricted to set nesting of depth k, RALG^k (BALG^k) [HS88, GV91, GM93]. We first compare RALG^2, BALG^2 and $\mathcal{P}$om-$\mathcal{A}$lg^2 restricted to queries over (nested) sets.

Theorem 7.1 RALG$^2 \subset$ BALG$^2 \subset \mathcal{P}$om-$\mathcal{A}$lg^2.

The strict inclusion of RALG^2 in BALG^2 was proved in [GM93]. It was also shown there that BALG^2 is included in PSPACE. The strict inclusion BALG^2 in $\mathcal{P}$om-$\mathcal{A}$lg^2 follows from the next proposition.

Theorem 7.2 $\mathcal{P}$om-$\mathcal{A}$lg$^2 = \mathcal{E}$, for unnested, unordered inputs/outputs.

Proof: We first show that every $hyper(i)$-time bounded query over unnested bags can be expressed in $\mathcal{P}$om-$\mathcal{A}$lg^2. The proof is based on an encoding of the configurations of a Turing machine in a set (i.e. unordered pomset with no duplicates) of 4-ary tuples of type $[\{U\}, \{U\}, U, U]$, and using the iterator I to simulate the moves of the machine. The first (second) attribute is used as index for the time (the cell on the tape). (An integer i is represented by an unordered pomset d_i containing i occurrences of a given constant.) A tuple $[d_i, d_j, a, q]$ is in the set iff at time i, the j^{th} cell of the tape contains the symbol a, and the head of the machine is on that cell in state q (if the head is elsewhere, then instead of q we have a special constant). Each iteration of I modifies the set so that it represents the successive configuration of the Turing machine. The formulas that (1) take the input bag and encodes it, i.e. build the set representing the TM tape, (2) compute the set representing the next TM state from the previous one, and (3) decoding the TM tape, i.e. build the output bag, are rather standard and very similar to the one used in [GM93] for simulating this process. (Omitted). The formulas there used the bag algebra BALG, which is a subset of $\mathcal{P}$om-$\mathcal{A}$lg.

To encode a computation of $hyper(i)$-time complexity over an input of size n, the iterator is applied on a list of length $hyper(i)(n)$. Each iteration step simulates one additional step of the TM computation. The list over which the iterator is applied is constructed as follows. Given a pomset p with n members (of arbitrary type), the expression $N(p) = \Lambda(\oplus^{-1}(\text{MAP}_a(D)))$ constructs a list of a's of length n. The next expression defines a list of length 2^n:

$E(p) = N(\mathcal{P}(\text{MAP}_{\lambda x.x}(\mathcal{P}(N(p)))))$ The first powerset application constructs a nested pomset containing n pomsets representing lists of length 1 to n. The MAP removes one level of nesting and result in an unnested pomset containing the n lists. The second powerset application construct a nested pomset containing 2^n pomsets (This is because it includes all pomsets containing some subset of the n lists, and there are 2^n such combinations). The last application of N constructs a list of length 2^n.

Finally, a list of length $hyper(i)(n)$ is constructed by $L(p) = E^i(p)$.

For the other direction note that every $\mathcal{P}$om-$\mathcal{A}$lg^2 query is in some hyper-exponential complexity, since every operation increases at most exponentially the size of its input. $\Box$

The above theorem deals with unordered unnested data. Theorem 6.9 showed that $\mathcal{P}$om-$\mathcal{A}$lg^1 computes all elementary functions on totally ordered data. To conclude this section we consider queries with general partially ordered, nested, input/output.

Theorem 7.3 For every $k \geq 3$, $\mathcal{P}$om-$\mathcal{A}$lg^k expresses exactly all the elementary queries over inputs of nesting $(k-1)$.

Proof:

The proof is based on encoding pomsets of nesting hight k in a bag of nesting $max(k, 2)$, and then using a result from [GM93] stating that every elementary queries over bags of nesting $(k-1)$ can be expressed by a bag algebra query that uses bags of nesting k. (Recall that the bag algebra is a subset of $\mathcal{P}$om-$\mathcal{A}$lg).

We sketch below how to encode an unnested pomset in a bag of nesting 2. The general case is handled similarly.

The idea is to (1) change (nondeterministically) the labels of objects in the pomset p so that so that all objects have distinct labels. The new label of an object o is of the form $[o, i]$ where i is bag containing i occurrences of some constant a. (The additional level of nesting is needed in order to choose a renaming function.) Then (2) encode the resulting pomset p' by a bag of tuples, were a tuple $[o_1, i_1, o_2, i_2]$ is in the set iff the object $[o_1, i_1]$ precedes the object $[o_2, i_2]$ in p'. The resulting bag is of bag nesting 2.

To compute this encoding we (1) construct all the possible sets of such tuples (The possible sets are constructed by applying the powerset operator on a set representing the active domain[7], (2) use the technique of Proposition 4.3 to construct from each such set the pomset represented by the set (this is done by using MAP and applying the expression of Proposition 4.3 on all the pomsets), and (3) select the sets where the pomset resulting from (2) equals to p'.

This gives us all the possible encodings. We then simulate the computation simultaneously on all these encodings, and decode the result. Since the computation is deterministic, all computations result in the same output.

The output is then decoded using the expression of Proposition 4.3.

□

Note that a similar result was obtained in [GM93] for BALG^k. It was also shown there that RALG$^k \subset$ BALG^k. It follows that:

Theorem 7.4 RALG$^k \subset$ BALG$^k = \mathcal{P}$om-$\mathcal{A}$lg^k, for every $k \geq 3$, and input/output of nesting height $(k-1)$.

8 Conclusion

The pomset data type generalizes all the classical collection types manipulated in databases, and this was the initial reason for carrying on the present research. We proposed an algebra

[7] The nesting level used here is 3. In the general case, for objects o_i of nesting depth $d \geq 1$, it will be d+2, i.e. the nesting depth of p plus 1

for pomsets which extends classical algebras for sets, bags and lists. Our main results apply to sub-cases of the general context of pomsets. We have proved that when restricted to specific types, we obtain nice equivalences with either logic (FO+FP+C) or complexity classes (PTIME, $\mathcal{E}$). This shows the robustness of some subsets of the algebra. Nevertheless, the precise expressive power of the full pomset datatype (including non series-parallel) is still an open question.

The restrictions imposed on the types, or on the operators to obtain the results deserve some discussion. The restriction to either totally ordered or totally unordered types is necessary to obtain the equivalence with fixpoint logic with counting. Totally unordered types allow the representation of tuples associated to a number, while the totally ordered types allow the representation of integers. More arguable is the restriction to planar pomsets used to obtain the tractability result. There is no doubt that such restrictions are often made in computer science to obtain tractable problems, but there is no formal evidence, that planar pomsets can be easily avoided in practice when writing queries. Nevertheless, in the examples that we considered, no non-planar pomset arised. The restriction on the operations follows the same goal, obtaining tractable languages. The fixpoint bounded in size in a technique which has already proved its efficiency. Again here there is no formal evidence that it is easy to write query for tractable problems which do not result in an exponential explosion, but here again, this never arised in the examples that we consider.

We claim that there are numerous practical data with duplicates and partial order among them. We have given some examples at the begining of the paper. We believe that the pomset algebra could also be used for the formalization of statistical databases, a field which suffer from a lack of mathematical foundation. Statistical databases contain statistical aggregate information about generally very large sets of raw data on collections such as a population for instance. Statistical tables contain tuples which are often partially ordered, and contain duplicates. A table for instance given information about people born in a given interval, can be ordered according to an order on the intervals. Moreover, statistical databases generally contain data arising from different inputs resulting in many duplicates that are maintained for efficiency reason. The knowledge of the order and the duplicates allows the use of complex aggregate functions, such as averages, etc. The notion of duplicates might also be generalized to a more semantical concept. Indeed, a yearly average is redundant with respect to a monthly average over the same period for instance. A pomset algebra enriched with a richer notion of duplicates might help the formalization of these sorts of data.

Acknowledgments: The authors wish to thank Vassos Hadzilacos and Leonid Libkin for technical remarks, and the anonymous referees for helpful comments.

References

[AB87] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. In *Very Large Data Bases Journal*, 4:4, pp. 727–794 (1995)

[ACM93] S. Abiteboul, S. Cluet, and T. Milo. Querying and updating the file. In *Proc. 19th Int'l Conf. on Very Large Data Bases*, 73-84, 1993.

[AG91] S. Abiteboul and S. Grumbach. A rule-based language with functions and sets. *ACM Transactions on Database Systems*, 16(1):1–30, March 1991.

[AHV94] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1994.

[AK89] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 159–173, 1989.

[Alb91] J. Albert. Algebraic properties of bag data types. In *Proc. 17th Int'l Conf. on Very Large Data Bases*, pages 211–219, 1991.

[AV91] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proc. ACM Symp. on Theory of Computing*, New Orleans, May 209-219, 1991.

[BBN91] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Proc. 3rd Int. Workshop on database programming languages*, pages 9–19, Nafplion, Aug. 1991. Morgan Kaufman.

[BK90] C. Beeri and Y. Kornatzky. Algebraic optimization of object oriented query languages. In *Proc. 3rd Intl. Conf. on Database Theory, ICDT 90, Paris, France*, 72-88, 1990.

[BM92] C. Beeri and T. Milo. Functional and predicative programming in oodb's. In *Proc. 11th Symp. on Principles of Database Systems San-Diego*, 176-190, 1992.

[BP91] J. Van Den Bussche and J. Paredaens. The expressive power of structured values in pure OODB's. In *Proc. 10th Symp. on Principles of Database Systems*, 291-299, 1991.

[BS91] V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with sets/bags/lists. In *Proc. 18th Int. Col. on Automata, Languages and Programming*, 1991.

[CH80] A. Chandra and D. Harel. Computable Queries for Relational Data Bases. *Journal of Computer and System Sciences*, 21(2):156–178, Oct. 1980.

[CR87] J. Canny and J. Reif. New lower bound techniques for robot motion planing problems. In *IEEE Conf. on Foundations of Computer Science*, pages 49–60, 1987.

[CRSG94] L. Colby, E. Robertson, L. Saxton, and D. Van Gucht. Query language for list-based complex objects. In *Proc. 13th ACM Symp. on Principles of Database Systems*, pages 179–189, 1994.

[Ag 96] S. Agarwal et al. On the computation of multidimensional aggregates. In *Proc. of Intl. Conf. on Very Large Data Bases*, 506-521, 1996.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability A Guide to the theory of NP-Completeness.* Freeman, 1979.

[GM93] S. Grumbach and T. Milo. Towards tractable algebras for bags. In *Proc. 12th ACM Symp. on Principles of Database Systems*, pages 49–58, Washington, May 1993.

[GMK93] S. Grumbach, T. Milo, and Y. Kornatzky. Calculi for bags and their complexity. In *4th Int. Workshop on Database Programming Languages*, New-York City, 65-97, Septembre 1993. Morgan Kaufmann.

[GO93] E. Grädel and M. Otto. Inductive definability with counting on finite structures. In *Proc. of Computer Science Logic 92*, pages 231–247. LNCS, n702, 1993.

[GS86] Y. Gurevich and S. Shelah. Fixed-point extensions of first-order logic. *Annals of Pure and Applied Logic*, 32:265–280, 1986.

[Gur83] Y. Gurevich. Algebras of feasible functions. In *Proc IEEE Foundations of Computer Science*, pages 210–214, 1983.

[GV90] S. Grumbach and V. Vianu. Playing games with objects. In *Proc. Int. Conf. on Database Theory*, pages 25–38, Paris, dec 1990. Springer Verlag. LNCS 470.

[GV91] S. Grumbach and V. Vianu. Tractable query languages for complex object databases. In *Proc. 10th ACM Symp. on Principles of Database Systems*, pages 315–327, Boulder, May 1991.

[GW92] S. Ginsburg and X. Wang. Towards a unified approach to querying sequenced data. In *Proc. 11th ACM Symp. on Principles of Database Systems*, pages 293–300, San Diego, 1992.

[HS88] R. Hull and J. Su. On the expressive power of database queries with intermediate types. In *Proc. 7th ACM Symp. on Principles of Database Systems*, 39-51, 1988.

[HS89] R. Hull and J. Su. Untyped Sets, Invention and Computable Queries. In *Proc. 8th ACM Symp. on Principles of Database Systems*, 347-359, 1989.

[HT74] J.E. Hopcroft and R.E. Tarjan. Efficient planarity testing. *J. of the ACM*, 21:549–568, 1974.

[HW74] J.E. Hopcroft and J.K. Wong. Linear time algorithm for isomorphism of planar graphs. In *Proc 6th ACM Symp. on THeory of Computing*, pages 172–184, 1974.

[Imm86] N. Immerman. Relational queries computable in polynomial time. *Inf. and Control*, 68:86–104, 1986.

[Jac82] B. Jacobs. On Database Logic. *Journal of the ACM*, 29(2):310–332, 1982.

[Joh90] D.S. Johnson. ***Handbook of Theorical Computer Science***, volume A, chapter A catalog of complexity classes, pages 67–162. North Holland, J. Van Leeuwen Ed., 1990.

[KRS85] H.F. Korth, M.A. Roth, and A. Silberschatz. Extended Algebra and Calculus for not 1NF Relational Databases. Technical report, Department of Computer Science, University of Texas at Austin, 1985.

[KV84] G.M. Kuper and M.Y. Vardi. A new approach to database logic. In ***Proc. 3rd ACM Symp. on Principles of Database Systems***, 86-97, 1984.

[LT92] R. Laurini and Thompson. ***Fundamentals of Spatial Information Systems.*** The A.P.I.C. Series, No. 37. Academic Press, 1992.

[LW94] L. Libkin and L. Wong. New techniques for studying set languages, bag languages and aggregate functions. In ***Proc. 13th ACM Symp. on Principles of Database Systems***, 155-166, 1994.

[Moh89] R. H. Mohring. ***Algorithms and Order, I. Rival Ed.***, chapter Computationally Tractable Classes of ordered Sets, pages 105–193. Kluwer Academic Publishers, 1989.

[Mum90] I.S. Mumick. et al. The magic of duplicates and aggregates. In ***Proc. 16th Intl. Conf. on Very Large Databases, Brisbane, Australia***, 264-277, 1990.

[Pra84] V. Pratt. The pomset model of parallel processes: Unifying the temporal and the spatial. In ***Seminar on Concurrency***, pages 180–196, Pittsburgh, 1984. LNCS 197.

[Ric92] J. Richardson. Supporting lists in a data model (a timely approach). In ***Proc. 18th Intl. Conf. on Very Large Databases***, Vancouver, 127-138, 1992.

[Suc93] D. Suciu. Bounded fixpoints for complex objects. In ***4th Int. Workshop on Database Programming Languages***, New-York City, 263-281, Septembre 1993. Morgan Kaufmann.

[SZL93] B. Subramanian, S. B. Zdonik, and T. W. Leung. Ordered types in the aqua data model. In ***4th Int. Workshop on Database Programming Languages***, New-York City, 115-135, Septembre 1993. Morgan Kaufmann.

[Ull88] J.D. Ullman. ***Database and Knowledge Base Systems.*** Computer Science Press, 1988.

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