Tractable Iteration Mechanisms for Bag Languages

Preliminary Report

Latha S. Colby **Leonid Libkin**

Bell Laboratories/Lucent Technologies
600 Mountain Avenue
Murray Hill, NJ 07974, USA
Email: {|colby, libkin|}@research.bell-labs.com

Abstract. The goal of this paper is to study tractable iteration mechanisms for bags. The presence of duplicates in bags prevents iteration mechanisms developed in the context of sets to be directly applied to bags without losing tractability. We study two constructs for controlling tractability of iteration over bags. The deflationary fixpoint construct keeps removing elements from a bag until a fixpoint is reached. The bounded fixpoint construct is an inflationary iteration mechanism that never exceeds some predefined bounding bag. We study these constructs in the context of a standard (nested) bag algebra. We show that the deflationary and bounded inflationary fixpoint constructs are equally expressive and strictly more expressive than their set-based counterparts. We also show that, unlike in the set case, the bag algebra with bounded fixpoint fails to capture all PTIME queries over databases with ordered domains. We then show that adding just one construct, which can be used to assign unique tags to duplicates, captures the class of all polynomial time queries over bags when a total ordering on the domain of atomic elements is available. Finally, we compare the expressive powers of the bag algebra and the nested relational algebra with aggregate functions in the presence of these fixpoint operators.

1 Introduction

While much of database theory is based on the theory of sets, in recent years, there has been a growing trend towards research on other collection data types such as bags and lists. An important goal in the design of query languages is to strike a reasonable balance between expressiveness and tractability. We use the term tractability to mean polynomial-time computable. The focus of this paper is on studying *tractable* iteration mechanisms for bags.

Such mechanisms have been developed in the context of set languages [11, 12, 14, 19, 21]. Most typically, an inflationary fixpoint construct is used for flat relations (sets of tuples). It was shown by Vardi [21] and by Immerman [14] that the relational algebra, when augmented with the inflationary fixpoint construct, can express all polynomial time queries over sets in the presence of a total ordering on the domain of elements. For nested relations, this causes intractability, as too many sets at a different level of nesting can be constructed. For instance, the powerset operator is definable via an inflationary fixpoint operator. Thus, several techniques have been developed in order to restrict the fixpoint operator. In [12], no operation creating additional levels of nesting can be iterated over; and in [19] a bound for the result of the fixpoint operator is precomputed. Both approaches give us precisely the PTIME queries over nested sets when a total order on the domain of atomic elements is available (this follows from the results in the papers cited above and in [13]).

It is shown in [10] that tractability may be obtained, in the context of complex-object languages, by a combination of restrictions and assumptions about the input database. They considered families of calculi with restrictions on set nesting and showed that if the input database is dense[1] with respect to its types, then the inflationary and partial fixpoint extensions of the corresponding calculus, express exactly the PTIME and PSPACE queries, respectively. They obtained similar expressiveness results by considering range-restricted versions of the calculi.

In the case of bags, the presence of duplicates prevents us from directly extending the results from the set-oriented framework. For instance, finite convergence for the inflationary fixpoint is not guaranteed because one can keep adding elements to a bag indefinitely. Iteration schemes with a predetermined finite number of iteration steps (such as, for example, loops in [16]) are also prone to intractability problems. For example, the function $\lambda x.(x\ \textit{bag--union}\ x)$, when applied repeatedly, will result in an exponential blow-up due to repeated doubling [5]. Techniques for controlling recursion in the presence of duplicates were presented in [5] in the context of nested lists and in [9] in the context of partially-ordered flat multisets. Tractability was achieved in [5] by controlling the number of recursion steps and the operations within the recursion steps, and in [9] by using a size-bounded structural recursion scheme.

As pointed out in [3, 8, 16], most real-life database systems provide query languages based on bag semantics and it is therefore natural to investigate expressiveness and tractability issues in the bag framework. We look at various ways of increasing expressiveness while maintaining tractability in a pure bag-oriented setting. In particular, we consider adding various tractable fixpoint operators to the standard nested bag algebra developed in [8, 7, 16, 17].

We first introduce a *deflationary fixpoint operator*, *dfp* which repeatedly *removes* elements from a bag, as opposed to inflationary fixpoint which adds elements. Thus, we avoid both nontermination and exponential blowup. We then introduce a (bag) *bounded fixpoint operator*, *bfp*, based on the one introduced by Buneman and studied in [19] in the context of nested relations. The main idea of this operator is that before the iteration starts, a bounding set or bag is computed, and after each step the intersection of the current result and the

[1] Density is measured in terms of the ratio of the cardinality of the database to the cardinality of the set of all objects of a type that are constructible from the set of atomic elements in the database.

bound is taken. Thus, the result of the fixpoint never exceeds this precomputed bound, and this avoids exponential blowup in the size of the result. It should be noted that the bounding is based on an element-wise comparison as opposed to a global size bound such as in the one used for controlling intractability in [9].

There are two ways of introducing the fixpoint operators. We can define a *set*- or *bag*-based fixpoint, depending on whether duplicates are or are not eliminated at each iteration step. We then consider these operators in the context of the standard (nested) bag algebra from [16], which we denote by $\mathcal{BQL}$ here. We prove that the set-based (bounded inflationary and deflationary) fixpoints are strictly weaker than the bag-based fixpoints, and that the bounded inflationary fixpoints are equivalent to their corresponding deflationary fixpoints.

In the case of sets, the (nested) relational algebra with the (bounded) fixpoint operator captures the class of all polynomial-time queries on (nested) relations over ordered domains. Does an equivalent result hold in the case of the bag algebra? We answer this question in the negative by showing that the bounded fixpoint language fails to capture PTIME. However, the solution to this problem is remarkably simple. Only one extra primitive is required to capture all PTIME queries. We show that the *gen* primitive, introduced in [17], when added to the bounded fixpoint bag algebra gives us a language that captures all PTIME queries (in the presence of an order on the domain). This operator was originally defined in the context of $\mathcal{NRL}^{\mathrm{nat}}$ which is the nested algebra with aggregates. It takes a number n and generates a sequence of numbers from 0 to n. By definition, this is a non-polynomial operation since it takes a number whose binary representation is of size $\log n$ and generates output that is of size $O(n \log n)$. However, in a bag setting, the corresponding operation is polynomial, since the numbers are coded in unary, that is, n is represented as a bag of n empty tuples.

It should be pointed out that the PTIME characterization presented in this paper is different from those of [5] and [9] for lists where a total ordering of *all* elements (including duplicates) is available (as opposed to an order relation on the domain). Also, unlike [9], we do not consider exponential primitives such as powerset when dealing with bags of nesting depth greater than one, and thus our PTIME characterization is not restricted to flat inputs and outputs. We believe that a similar PTIME result can be obtained by replacing the bounded (or deflationary) fixpoint constructs with a bounded structural recursion construct where the structural recursion is based on the insert presentation of bags (see [17]) and the bounding is similar to those used for the fixpoint constructs in this paper. Structural recursion with a size-based bound as in [9] would yield a similar characterization and would not require the *gen* operation since it would be expressible using such a recursion scheme.

The intuition behind the use of *gen* is rather simple. The order relation on the domain is needed for capturing polynomial-time queries over sets so that the order in which elements appear on a Turing machine tape can be modeled. Now assume that we have a bag $\{\!|a, a, a|\!\}$. This can be encoded on a Turing machine tape as $\{\!|enc(a)\#enc(a)\#enc(a)|\!\}$, assuming that $\{\!|, |\!\}, \#$ are in the alphabet and $enc(a)$ is the encoding of a. Thus, from the point of view of a polynomial-time

TM there is the first a, the second a and the third a, whereas the bag algebra, even with the order relation, cannot distinguish between these a's! To eliminate this mismatch, we can use the *gen* operator as a tagging primitive. Note that *gen* does *not* force bag-based objects into set-based objects, because it uses bags in an essential way, as will be seen later.

Another interesting consequence of adding the *gen* operator is that the difference in expressive powers between the bag-based and set-based bounded fixpoints disappears. In other words, the set-based fixpoint algebras are equivalent to the bag-based fixpoint algebras in the presence of the *gen* operator.

We investigate the relationship between the bag languages and languages with aggregates in the presence of fixpoint operators. In [17], it was shown that the bag algebra $\mathcal{BQL}$ has the same expressive power as a nested relational language with aggregate functions $\mathcal{NRL}^{\mathrm{nat}}$. We show that adding the equivalent of the *gen* operation to $\mathcal{NRL}^{\mathrm{nat}}$ results in a language that is extremely powerful. In the presence of certain additional operators and set-based bounded fixpoint, it expresses all polynomial-time computable functions on natural numbers, but it can also express many EXPTIME computable functions.

Organization The next section introduces the basic bag algebra $\mathcal{BQL}$, and the (deflationary and bounded inflationary) bag-based and set-based fixpoint operators. In Section 3, we study the relationship between these fixpoint constructs. The characterization of the PTIME queries over bags is given in Section 4. In Section 5, we study the connections between bag languages and set languages with aggregates and fixpoints. Some open problems are listed in Section 6.

2 Bag algebra and fixpoint operators

In this section, we give an overview of the bag language $\mathcal{BQL}$, and introduce the different fixpoint operators.

2.1 Bag algebra

Figure 1 contains the expressions of the language $\mathcal{BQL}$ (Bag Query Language) [16, 17]. The design of this language is based on a general framework for the design of query languages over collection types [2]. It must be noted that languages like $\mathcal{BQL}$ normally have three equally expressive components: the algebra, the calculus, and the comprehension language, cf. [2]. In this preliminary report we use only the algebra; the calculus is very helpful in doing some inductive proofs and will be used, together with the algebra, in the full version.

The types of $\mathcal{BQL}$ are given by the grammar

$$s, t ::= b \mid unit \mid s \times t \mid \{\!|s|\!\},$$

where b is a base type whose domain is an unspecified infinite set, type *unit* has the unique element denoted by $()$, elements of type $s \times t$ are pairs (x, y) where x

$$\frac{}{id^s : s \to s} \qquad \frac{h : r \to s \quad g : s \to t}{g \circ h : r \to t} \qquad \frac{g : r \to s \quad h : r \to t}{\langle g, h \rangle : r \to s \times t}$$

$$\frac{}{\pi_1^{s,t} : s \times t \to s} \qquad \frac{}{\pi_2^{s,t} : s \times t \to t} \qquad \frac{}{!^s : s \to unit}$$

$$\frac{}{b\_\eta^s : s \to \{\!|s|\!\}} \qquad \frac{}{b\_\mu^s : \{\!|\{\!|s|\!\}|\!\} \to \{\!|s|\!\}} \qquad \frac{f : s \to t}{b\_map(f) : \{\!|s|\!\} \to \{\!|t|\!\}}$$

$$\frac{}{K\{\!||\!\}^s : unit \to \{\!|s|\!\}} \qquad \frac{}{\uplus^s : \{\!|s|\!\} \times \{\!|s|\!\} \to \{\!|s|\!\}} \qquad \frac{}{b\_\rho_2^{s,t} : s \times \{\!|t|\!\} \to \{\!|s \times t|\!\}}$$

$$\frac{}{\dot{-}^s : \{\!|s|\!\} \times \{\!|s|\!\} \to \{\!|s|\!\}} \qquad \frac{}{\varepsilon^s : \{\!|s|\!\} \to \{\!|s|\!\}}$$

Fig. 1. Expressions of $\mathcal{BQL}$

is of type s and y is of type t, and elements of type $\{\!|s|\!\}$ are finite bags containing elements of type s.

Let us briefly review the semantics, cf. [17]. id is the identity function. $g \circ h$ is the composition of functions g and h; that is, $(g \circ h)(x) = g(h(x))$. The bang ! produces () on all inputs. π_1 and π_2 are the two projections on pairs. $\langle g, h \rangle$ is pair formation; that is, $\langle g, h \rangle(x) = (g(x), h(x))$. $K\{\!||\!\}$ produces the empty bag. $\uplus$ is additive bag union; for example, $\uplus(\{\!|1,2,3|\!\}, \{\!|2,2,4|\!\})$ returns $\{\!|1,2,3,2,2,4|\!\}$. b_η forms singleton bags; for example, $b\_\eta(3)$ evaluates to the singleton bag $\{\!|3|\!\}$. b_μ flattens a bag of bags; for example, $b\_\mu(\{\!|\{\!|1,2|\!\}, \{\!|1,3|\!\}, \{\!|2,4|\!\}|\!\})$ evaluates to $\{\!|1,1,2,2,3,4|\!\}$. $b\_map(f)$ applies f to every item in the input bag; for example, $b\_map(\lambda x.1+x)\ \{\!|1,2,1,6|\!\}$ evaluates to $\{\!|2,3,2,7|\!\}$ and $b\_map(\lambda x.1)\ \{\!|1,2,1,6|\!\}$ evaluates to $\{\!|1,1,1,1|\!\}$. $b\_\rho_2(x,y)$ pairs x with every item in the bag y; for example, $b\_\rho_2(3, \{\!|1,2,3,1|\!\})$ returns $\{\!|(3,1),(3,2),(3,3),(3,1)|\!\}$. We use $\dot{-}$ to denote bag difference; for example, $\dot{-}\ (\{\!|1,1,2,3,3|\!\}, \{\!|1,2,2|\!\}) = \{\!|1,3,3|\!\}$. Finally, ε eliminates duplicates: $\varepsilon(\{\!|1,1,2,2,2|\!\}) = \{\!|1,2|\!\}$.

We shall always omit the type superscripts as the most general types can be inferred. We shall also occasionally use the infix notation for operations like $\dot{-}$ and $\uplus$, that is, we will write $B \dot{-} B'$, $B \uplus B'$, etc.

The language $\mathcal{BQL}$ as presented here was introduced in [16]; it is also equivalent to the polynomial fragment of the BALG algebra of [8]. The operations max, min, eq, $member$, $subbag$ and many others are also definable in it [16] (max and min are maximal and minimal bag intersections, and eq, $member$ and $subbag$ test for equality, membership and containment). Following [17], for notational convenience we add booleans (truth value represented by $\{\!|()|\!\}$ and false by $\{\!||\!\}$) and the conditional construct *if-then-else*. We also use the λ-notation, i.e. we write $\lambda x.f(x)$ provided x is of object type (that is, no higher-order functions are allowed). For syntactic convenience, we define functions Π, σ and $\times$ to denote projection, selection and cartesian product on bags. These constructs do not add expressive power. For example, Π_i can be defined as $b\_map(\pi_i)$.

In what follows, $\mathcal{L}(p_1, \ldots, p_n)$ is the notation for a language $\mathcal{L}$ augmented with primitives $p_1, \ldots, p_n$. We shall often use the language $\mathcal{BQL}(\leq)$, where the function $\leq : b \times b \rightarrow \{\!|unit|\!\}$ testing a linear order on the elements of base types is available.

The following is from [2, 17]:

Proposition 1. *Every function expressible in $\mathcal{BQL}(\leq)$ has polynomial-time complexity with respect to the size of the input.* □

2.2 Fixpoint operators

As we mentioned before, we must define fixpoint operators over bags that do not lead to nontermination and maintain tractability. To this end, we look at two possibilities for controlling the fixpoint computation. Both use the idea of bounds. The first construct, *the deflationary fixpoint*, removes elements from some initial bag at each step of the iteration. In contrast, the *bounded fixpoint*, keeps adding elements as long as they are within some precomputed bound. The iteration terminates when there is no change in the result of two successive iteration steps.

Let us give the formal definitions. Both deflationary and bounded fixpoints have the following typing rule:

$$\frac{f : s \times \{\!|t|\!\} \rightarrow \{\!|t|\!\} \qquad g : s \rightarrow \{\!|t|\!\}}{dfp_{f,g} : s \rightarrow \{\!|t|\!\}} \qquad \frac{f : s \times \{\!|t|\!\} \rightarrow \{\!|t|\!\} \qquad g : s \rightarrow \{\!|t|\!\}}{bfp_{f,g} : s \rightarrow \{\!|t|\!\}}$$

To define the semantics of these operations, assume that we are given an input object x of type s. Let $B = g(x)$. This is the "bound" for the computation. We define two families of bags:

- $Y_0 = \{\!||\!\}$, $Y_{i+1} = (Y_i \uplus f(x, Y_i))\ min\ B$;
- $Z_0 = B$, $Z_{i+1} = Z_i \ \dot{-}\ f(x, Z_i)$.

Now $bfp_{f,g}(x)$ is defined to be Y_i where $Y_i = Y_{i+1}$ and i is the smallest such. We define $dfp_{f,g}(x)$ to be Z_i where $Z_i = Z_{i+1}$ and i is the smallest such. It is easy to see that in both cases the i at which the computation stops is at most the cardinality of the bounding (or initial) bag B.

It should be noted that the definition in [19] allows types of the form $\{\!|t_1|\!\} \times \{\!|t_2|\!\} \times \ldots \times \{\!|t_m|\!\}$ to be used in place of $\{\!|t|\!\}$ in the definition of the bounded fixpoint for set languages. The operations $\cup$ and $\cap$ are performed componentwise. It is then shown in [19] that this is only a matter of convenience, that is, no expressiveness is gained. Similar results can be shown in the bag setting.

To simulate the more general fixpoint, we encode each tuple $(B_1, \ldots, B_m)$ of type $\{\!|t_1|\!\} \times \{\!|t_2|\!\} \times \ldots \times \{\!|t_m|\!\}$ by a bag B of type $\{\!|\{\!|t_1|\!\} \times \{\!|t_2|\!\} \times \ldots \times \{\!|t_m|\!\}|\!\}$, where for each $x \in B_i$, there exists a tuple of the form $(\{\!||\!\}, ..., \{\!|x|\!\}, ..., \{\!||\!\})$ in B

($\{\!|x|\!\}$ occurs in the ith position). For example, $(\{\!|a,b|\!\},\{\!|\,|\!\},\{\!|c|\!\})$ is represented by $\{\!|(\{\!|a|\!\},\{\!|\,|\!\},\{\!|\,|\!\}),(\{\!|b|\!\},\{\!|\,|\!\},\{\!|\,|\!\}),(\{\!|\,|\!\},\{\!|\,|\!\},\{\!|c|\!\})|\!\}$. Each Y_i (or Z_i) is represented using this encoding and is decoded into the original representation before the fixpoint operation f is applied. The bounding bag $g(x)$ is represented using the same encoding. The encode and decode steps are easily expressible in $\mathcal{BQL}$ and are simpler[2] than those used in [19]. Thus, for the sake of simplicity we use fixpoints as they are defined above in this report.

Recursive queries such as the transitive closure of a graph can be expressed using *bfp* by translating the corresponding solutions from the set case in [19] verbatim to the bag case. For transitive closure, one uses $B = \varepsilon((\Pi_1(R) \uplus \Pi_2(R)) \times (\Pi_1(R) \uplus \Pi_2(R)))$ as the bound, where R is the binary relation representing the set of edges. That is, B is the complete graph on the set of nodes. Then the composition of relations is iterated until the transitive closure is constructed.

As another example, we show how to define the parity of the cardinality of a bag using the deflationary fixpoint construct. Let

$g = b\_map(!)$ and

$f = \lambda(x,y).\mathit{if}\ eq(y, b\_\eta(!(y)))\ \mathit{then}\ K\{\!|\,|\!\}(!(y))\ \mathit{else}\ b\_\eta(!(y)) \uplus b\_\eta(!(y))$

In other words, for each n-element bag x, $g(x)$ returns the bag of n units $()$, and f returns the empty bag if its input is $\{\!|()|\!\}$ and it returns $\{\!|(),()|\!\}$ otherwise. Then $dfp_{f,g}(x)$ is $\{\!|()|\!\}$ if n is odd, and $\{\!|\,|\!\}$ if n is even, thus giving us the parity test. Note that we did not use the order relation in this example.

Finally, we define the *set-based bounded fixpoint* bfp^{set} and the *set-based deflationary fixpoint* dfp^{set}. Their typing rules are exactly the same as those for *bfp* and *dfp*. The semantics of bfp^{set} is defined similar to the semantics of *bfp* except that B is defined as $\varepsilon(g(x))$, not as $g(x)$. That is, the result produced at each iteration step has no duplicates. This corresponds precisely to the bounded fixpoint for set languages that was studied in [19]. The semantics of dfp^{set} is defined analogously.

Proposition 2. *Every function definable in $\mathcal{BQL}(\leq, dfp)$, or $\mathcal{BQL}(\leq, bfp)$, or $\mathcal{BQL}(\leq, bfp^{\text{set}})$, or $\mathcal{BQL}(\leq, dfp^{\text{set}})$ has polynomial-time complexity with respect to the size of the input.*

Proof sketch: The proof is by a simple induction argument. All functions expressible in $\mathcal{BQL}(\leq)$ are polynomial-time computable. Suppose that y is an input to $bfp_{f,g}$. The size of $g(y)$ (and hence the size of the result of $bfp_{f,g}$) is bounded by a polynomial p on the size of y. From the definition of bounded fixpoint, we see that the number of iteration steps in the computation of $bfp_{f,g}$ is no greater than the cardinality of $g(y)$, and each iteration step is polynomial-time computable, from which polynomial-time computability of $bfp_{f,g}(y)$ follows. The proofs for *dfp*, bfp^{set} and dfp^{set} are similar. □

[2] In [19], the encodings are chosen so that there is no increase in set height. This is necessary for the proof of the conservativity result presented in that paper.

3 Relative expressive power of fixpoint operators

In this section, we study the relationship between the various fixpoint operators from the previous section. Our first result is this:

Theorem 3. *(a) $\mathcal{BQL}(dfp)$ and $\mathcal{BQL}(bfp)$ have the same expressive power, and (b) $\mathcal{BQL}(dfp^{\mathrm{set}})$ and $\mathcal{BQL}(bfp^{\mathrm{set}})$ have the same expressive power.*

Proof sketch: We show that *dfp* is expressible in $\mathcal{BQL}(bfp)$ and, vice versa, that *bfp* is expressible in $\mathcal{BQL}(dfp)$. The main idea behind the simulation of *dfp* in terms of *bfp* is to use *bfp* to compute the complement of the result of the deflationary fixpoint, and similarly for the converse simulation.

Lemma 4. *Let f be a function of type $s \times \{\!|t|\!\} \to \{\!|t|\!\}$ and g be of type $s \to \{\!|t|\!\}$. Let $f' = \lambda(y, z).f(y, (g(y) \dot{-} z))$. Then $dfp_{f,g}(o) = g(o) \dot{-} bfp_{f',g}(o)$ for any object o of type s.*

Proof: Fix o of type s and let $Y_i : \{\!|t|\!\}$ denote the ith iteration of $bfp_{f',g}(o)$, and Z_i denote the ith step of $dfp_{f,g}(o)$. We show by induction on i, that $g(o) \dot{-} Y_i = Z_i$. The lemma will follow from this. If $i = 0$, then this follows from $Z_0 = g(o)$ and $Y_0 = \{\!||\!\}$. Assume $g(o) \dot{-} Y_i = Z_i$ and prove $g(o) \dot{-} Y_{i+1} = Z_{i+1}$:

$$\begin{array}{ll}
g(o) \dot{-} Y_{i+1} & \\
= g(o) \dot{-} ((Y_i \uplus f(o, (g(o) \dot{-} Y_i))) \ min \ g(o)) & \text{by definition of } bfp \text{ and } f' \\
= g(o) \dot{-} ((Y_i \uplus f(o, Z_i)) \ min \ g(o)) & \text{by the hypothesis} \\
= g(o) \dot{-} (Y_i \uplus f(o, Z_i)) & \text{since } A \dot{-} (B \ min \ A) \equiv A \dot{-} B \\
= (g(o) \dot{-} Y_i) \dot{-} f(o, Z_i) & \text{since } (A \dot{-} B) \dot{-} C = A \dot{-} (B \uplus C) \\
= Z_i \dot{-} f(o, Z_i) & \text{by the hypothesis} \\
= Z_{i+1} & \text{by definition of } dfp
\end{array}$$

The converse is established in the following lemma.

Lemma 5. *Let f be a function of type $s \times \{\!|t|\!\} \to \{\!|t|\!\}$ and g be of type $s \to \{\!|t|\!\}$. Let $f' = \lambda(y, z).f(y, (g(y) \dot{-} z))$. Then $bfp_{f,g}(o) = g(o) \dot{-} dfp_{f',g}(o)$, for any object o of type s.*

Proof: As before, fix $o : s$ and let Y_i and Z_i denote ith stage of the computation of $bfp_{f,g}(o)$ and $dfp_{f',g}(o)$, resp. Again, it suffices to show that $g(o) \dot{-} Z_i = Y_i$ for all i. The base case is the same as in Lemma 4. Now assume $g(o) \dot{-} Z_i = Y_i$ and prove $g(o) \dot{-} Z_{i+1} = Y_{i+1}$.

First note that all Z_js are subbags of $g(o)$. From this, using the equations for reasoning about the equivalence of bag expressions from [6], calculate

$$\begin{array}{ll}
g(o) \dot{-} Z_{i+1} & \\
= g(o) \dot{-} (Z_i \dot{-} f'(o, Z_i)) & \text{by definition of } dfp \\
= g(o) \dot{-} (Z_i \dot{-} f(o, (g(o) \dot{-} Z_i))) & \text{by definition of } f' \\
= g(o) \dot{-} (Z_i \dot{-} f(o, Y_i)) & \text{by the hypothesis} \\
= (g(o) \dot{-} Z_i) \uplus ((Z_i \min f(o, Y_i)) \dot{-} (Z_i \dot{-} g(o))) & \text{by (P8) of [6, p. 333]} \\
= (g(o) \dot{-} Z_i) \uplus (Z_i \min f(o, Y_i)) & \text{since } Z_i \subseteq g(o)
\end{array}$$

On the other hand,

$$\begin{array}{ll} Y_{i+1} \;=\; (Y_i \uplus f(o, Y_i)) \min g(o) & \text{by definition of } \mathit{bfp} \\ = ((g(o) \dot{-} Z_i) \uplus f(o, Y_i)) \min g(o) & \text{by the hypothesis} \\ = [(g(o) \dot{-} Z_i) \min g(o)] \;\uplus [f(o, Y_i) \min(g(o) \dot{-} (g(o) \dot{-} Z_i))] & \text{by (P12) of [6]} \\ = (g(o) \dot{-} Z_i) \uplus (Z_i \min f(o, Y_i)) & \text{since } Z_i \subseteq g(o) \end{array}$$

which proves the lemma.

Using these lemmas, one can show by a straightforward induction argument, that *bfp* is expressible in $\mathcal{BQL}(\mathit{dfp})$ and vice versa, thus proving Theorem 3(a).

We now sketch the proof of Theorem 3(b). Let $\mathit{dfp}^{\mathrm{set}}_{f,g}$ be an expression in $\mathcal{B}(\mathit{dfp}^{\mathrm{set}})$, and let f' be constructed as in the proofs of Lemmas 4 and 5. Then, for any object o,

$$\mathit{dfp}^{\mathrm{set}}_{f,g}(o) = \mathit{dfp}_{f,(\varepsilon \circ g)}(o) = \varepsilon(g(o)) \dot{-} \mathit{bfp}_{f',(\varepsilon \circ g)}(o) = \varepsilon(g(o)) \dot{-} \mathit{bfp}^{\mathrm{set}}_{f',g}(o)$$

Using this equation and its symmetric analog, one can easily conclude that $\mathit{dfp}^{\mathrm{set}}$ and $\mathit{bfp}^{\mathrm{set}}$ are interdefinable, from which Theorem 3(b) follows. □

Next, we compare the expressive powers of the set- and bag-based fixpoints.

Theorem 6. *$\mathcal{BQL}(\mathit{bfp})$ is strictly more expressive than $\mathcal{BQL}(\mathit{bfp}^{\mathrm{set}})$. Also, $\mathcal{BQL}(\leq, \mathit{bfp})$ is strictly more expressive than $\mathcal{BQL}(\leq, \mathit{bfp}^{\mathrm{set}})$. Similar results hold for dfp and $\mathit{dfp}^{\mathrm{set}}$.*

Proof sketch: The inclusion is obvious as $\mathit{bfp}^{\mathrm{set}}$ can be simulated with *bfp*: $\mathit{bfp}^{\mathrm{set}}_{f,g} = \mathit{bfp}_{f,\varepsilon \circ g}$. To prove strictness, let a be an object of base type b, and let M_a be the collection of all bags of the form $\{\!|a, \ldots, a|\!\}$. For any function $f : \{\!|b|\!\} \to \{\!|\mathit{unit}|\!\}$, let $\mathrm{TRUE}(f, a) = \{\mathit{card}(x) \mid x \in M_a, f(x) = \{\!|()|\!\}\}$.

To prove separation, we need the following proposition.

Proposition 7. *For every $\mathcal{BQL}(\leq, \mathit{bfp}^{\mathrm{set}})$ function $f : \{\!|b|\!\} \to \{\!|\mathit{unit}|\!\}$, and every object a of type b, the set $\mathrm{TRUE}(f, a)$ is either finite or co-finite. In particular, the parity test is inexpressible in $\mathcal{BQL}(\leq, \mathit{bfp}^{\mathrm{set}})$.*

This proposition and the observation made above that the parity test is definable in $\mathcal{BQL}(\mathit{bfp})$ prove the theorem.

To sketch the proof of Proposition 7, we need a definition first. Given a number $k > 0$, define the class OBJ_k of *k-objects* as follows. First, every object of the base type and the object $()$ of type *unit* belong to OBJ_k. A pair (x, y) is a k-object if both its components are. Finally, a bag is a k-object if it has at most k distinct elements and each of them is a k-object. Now, we prove the following lemma.

Lemma 8. *Let $f : s \to t$ be a $\mathcal{BQL}(\leq, \mathit{bfp})$ function, and let $k > 0$. Then there exists a number $c > 0$, that depends only on k and f, such that for any x of type s in OBJ_k, it is the case that $f(x) \in \mathrm{OBJ}_c$.*

We prove this lemma by induction on the $\mathcal{BQL}(\leq, bfp)$ expressions. Let us give a few cases for illustration. If $f = b\_\mu$ and $x \in \mathrm{OBJ}_k$, then $f(x) \in \mathrm{OBJ}_{k^2}$. Indeed, if $x = \{\!|B_1, \ldots, B_n|\!\}$ with at most k of B_is being distinct, and each B_i having at most k distinct elements, then $B_1 \uplus \ldots \uplus B_n$ has at most k^2 distinct elements. Assume that $f = b\_map(g)$ and $x \in \mathrm{OBJ}_k$. By induction hypothesis, find c_0 such that $g(y) \in \mathrm{OBJ}_{c_0}$ for y in OBJ_k. Then we can take c to be $\max(c_0, k)$: indeed, $f(x)$ contains at most k distinct objects, each being a c_0-object. Finally, if $f = bfp_{g,h}$, then for each k, the constant c is determined by h, since if a bag $B \in \mathrm{OBJ}_c$ and B' is a subbag of B, then $B' \in \mathrm{OBJ}_c$.

Given the lemma (which applies to every $\mathcal{BQL}(\leq, bfp^{\mathrm{set}})$ function as well), we fix k and consider an expression of the form $bfp^{\mathrm{set}}_{f,g}$. When applied to a k-object x, it first computes a bound, $\varepsilon(g(x))$. Since $g(x) \in \mathrm{OBJ}_c$ for some fixed c, the bound has at most c elements and thus the fixpoint computation can be simulated directly in $\mathcal{BQL}(\leq)$. Applying this argument inductively, we obtain that for every $k > 0$ and every $\mathcal{BQL}(\leq, bfp^{\mathrm{set}})$ expression f, there is a $\mathcal{BQL}(\leq)$ expression f' such that $f(x) = f'(x)$ whenever $x \in \mathrm{OBJ}_k$. In particular, every $f : \{\!|b|\!\} \to \{\!|unit|\!\}$ coincides with some $\mathcal{BQL}(\leq)$ function f' on bags from M_a. It follows from the results of [16, 17] that $\mathcal{BQL}(\leq)$ can test only finite or co-finite cardinalities of bags from M_a, which completes the proof of Theorem 6. □

In particular, the theorem above shows that the set-based bounded fixpoint we defined is different from $\varepsilon \circ bfp$, since the parity test is definable using $\varepsilon \circ bfp$, but is not definable using bfp^{set}.

The question arises: what does one have to add to $\mathcal{BQL}(bfp^{\mathrm{set}})$ in order to express bfp? It turns out that we only need to add one extra primitive that will play the crucial role in the next section.

4 Capturing all PTIME queries on nested bags

It was shown in [19] that adding the bounded fixpoint to a nested *set* language is sufficient to capture all PTIME queries over nested *sets*, if a linear order is available on the base type. One may ask if a similar result holds for bags. Somewhat surprisingly, the answer is no.

Let us first recall the operator *gen*, introduced in [16]. Its type is $\{\!|unit|\!\} \to \{\!|\{\!|unit|\!\}|\!\}$. We denote the bag of n units, $\{\!|(), \ldots, ()|\!\}$, by $\underline{n}$. On the input $\underline{n}$, *gen* produces $\{\!|\underline{0}, \ldots, \underline{n}|\!\}$. For example,

$$gen(\{\!|(), (), ()|\!\}) = \{\!|\{\!||\!\}, \{\!|()|\!\}, \{\!|(), ()|\!\}, \{\!|(), (), ()|\!\}|\!\}.$$

Note that *gen* is polynomial-time computable. In contrast, the analogous operation gen^{nat} on natural numbers defined as $gen^{\mathrm{nat}}(n) = \{0, ..., n\}$, is not a polynomial operation.

This operator is quite powerful and can compute some queries that are not definable in $\mathcal{BQL}$, for example, the parity test, see [16]. The theorem below demonstrates that $\mathcal{BQL}(\leq, bfp)$ fails to capture all PTIME queries over bags, in particular, *gen*.

Theorem 9. *The function gen is not definable in* $\mathcal{BQL}(\leq, bfp)$.

Proof. Recall the definition of *k-objects* from the proof of Proposition 7. Assume that f is a function of $\mathcal{BQL}(\leq, bfp)$ that implements *gen*. Then, by Lemma 8, there exists a number c such that $f(x) \in \mathrm{OBJ}_c$ for any input x to *gen*, since $x \in \mathrm{OBJ}_1$. However, $gen(\underline{c}) \in \mathrm{OBJ}_{c+1} - \mathrm{OBJ}_c$. Thus, *gen* is not $\mathcal{BQL}(\leq, bfp)$-definable. $\square$

Now we define the class $\mathrm{PTIME}^{\mathrm{bag}}$ of polynomial-time queries over nested bags. In what follows, we restrict ourselves to *product-of-bag* types, that is, types of the form $\{\!|t_1|\!\} \times \ldots \times \{\!|t_m|\!\}$, where t_is are arbitrary types. In other words, we are interested in queries that take a tuple of bags as an input and produce outputs that are tuples of bags. This restriction is often made when one captures a complexity class over relations or complex objects, cf. [19, 20]. Extension to scalar types can be achieved rather straightforwardly, for example, by using a function extracting an element from a singleton set.

We use the standard encoding scheme such as the one in [1]. Given a set of values $A = \{a_1, \ldots, a_n\}$ of the base type b such that $a_1 < \ldots < a_n$, we encode a_i as the binary representation of i. We use 0 to encode the unique element of type *unit*. Next, using the brackets $\{\!|, |\!\}, (,)$ and the separator # we encode complex objects, relative to the set A. By the standard encoding of an object we now mean the one relative to the *active domain* of the object.

Consider two types s and t. We say that a function f from objects of type s to objects of type t that does not extend the active domain of its input, belongs to $\mathrm{PTIME}^{\mathrm{bag}}_{s,t}$ if there exists a polynomial-time Turing machine M such that: (1) when the input tape does not have the standard encoding of an object of type s, it prints a special symbol meaning "error" on its tape and stops, and (2) when the input tape contains the standard encoding of an object of type s (that is, the encoding relative to A, the active domain), it returns the encoding of $f(x)$, relative to A.

Finally, we define $\mathrm{PTIME}^{\mathrm{bag}}$, the class of polynomial-time queries over nested bags, to be the union of $\mathrm{PTIME}^{\mathrm{bag}}_{s,t}$ for all pairs of (product-of-bags) types s and t. The following can be seen from Theorem 9.

Corollary 10. $\mathcal{BQL}(\leq, bfp) \subset \mathrm{PTIME}^{\mathrm{bag}}$. $\square$

The main result of this section characterizes the class $\mathrm{PTIME}^{\mathrm{bag}}$.

Theorem 11. *The language* $\mathcal{BQL}(\leq, bfp, gen)$ *expresses precisely the class of queries in* $\mathrm{PTIME}^{\mathrm{bag}}$: $\mathcal{BQL}(\leq, bfp, gen) = \mathrm{PTIME}^{\mathrm{bag}}$.

Proof sketch: The inclusion $\mathcal{BQL}(\leq, bfp, gen) \subseteq \mathrm{PTIME}^{\mathrm{bag}}$ follows from Proposition 2 and the polynomiality of *gen*. For the reverse inclusion, assume that a query f of type $s \to t$ is computable by a PTIME machine M, whose number of steps is bounded by a polynomial $p(n)$, where n is the length of the input. It is not hard to construct an expression g that, given an object x whose encoding takes n cells, produces $\underline{m}$, where $p(n) \leq m$. This gives us the required count. Applying *gen* to $\underline{m}$, we obtain a representation of the tape (i.e., each cell is now identified by its unique label). The rest of the proof follows the standard idea: an input is encoded, then the machine M's actions are simulated on it, and the result is decoded back into an object. Since we use the bounded fixpoint in our language, let us just give an idea of how the bound is computed and the work of M is simulated. Assume for simplicity that each cell is either 0 or 1 (i.e., there are no other symbols in the alphabet; in fact, one needs three bits to encode the alphabet that contains all appropriate delimiters). It can change its value at most m times. The idea of the simulation is that when the ith cell changes its value, we look at all pairs $(\underline{i}, \underline{l})$ in the working bag (which is of type $\{\!|\{\!|unit|\!\} \times \{\!|unit|\!\}|\!\}$), find the maximum such l and add $(\underline{i}, \underline{l+1})$ to the bag. Thus, we can use $gen(\underline{m}) \times gen(\underline{m})$ as the bound for the fixpoint computation on the working bag that simulates M. When the simulation is done, a bag B is computed. One can use B to get the contents of the tape as follows: look at the initial value of the ith cell and the parity of the bag $\sigma_{\pi_1(x)=\underline{i}}(B)$. This determines if the value of the cell has changed during the computation. Since this parity test can be computed using either the fixpoint operation or gen, we get the encoding of the result which can then be decoded into the corresponding object. More details and the routine encoding and decoding schemes will be given in the full version. $\square$

Since the primitive gen assigns unique tags to duplicates, it is sufficient to simulate bfp with bfp^{set}. That is,

Proposition 12. *$\mathcal{BQL}(bfp^{\text{set}}, gen)$ can express bfp.* $\square$

From this we conclude:

Corollary 13. $\mathcal{BQL}(\leq, bfp^{\text{set}}, gen) = \text{PTIME}^{\text{bag}}$. $\square$

5 Aggregation and fixpoint operators

In this section, we study the relationship between bag languages and languages with aggregation in the presence of fixpoint operators. In [17], it is shown that $\mathcal{BQL}$ has the same expressive power as the nested relational language with aggregate functions over the natural numbers, denoted by $\mathcal{NRL}^{\text{nat}}$. It is known how to capture PTIME over the nested relations [10, 13, 19], and it is also known how to capture many complexity classes for arithmetic functions [4, 15]. Thus, one might ask if the correspondence between $\mathcal{BQL}$ and $\mathcal{NRL}^{\text{nat}}$ allows us to enrich the latter to capture PTIME in both worlds: relational and arithmetical. We shall prove a few initial results indicating that it is hard to find a natural extension like this.

First, let us review how the language $\mathcal{NRL}^{\text{nat}}$ is obtained. Take $\mathcal{BQL}$ and replace each bag operator with its set analog. For example, replace $\uplus$ with union and $\dot{-}$ with set difference. Then add the type of natural numbers $\mathbb{N}$ with the usual arithmetic operations $+, *, \dot{-}$ and 1 as a constant (that is, $K1 : unit \to \mathbb{N}$), and the summation construct $\sum(f) : \{s\} \to \mathbb{N}$, provided f is of type $s \to \mathbb{N}$. When applied to a set $\{x_1, \ldots, x_n\}$, the summation construct yields $\sum_{i=1}^{n} f(x_i)$. For instance, if $f = K1 \circ !$ (that is, $f(x) = 1$ for any x), then $\sum(f)$ is the cardinality function.

In the proof of equivalence of expressive power, each natural number n is translated into $\underline{n}$, that is, $\{\!|(), \ldots, ()|\!\}$, n times. Correspondingly, *gen* is translated into $gen^{\mathrm{nat}} : \mathbb{N} \to \{\mathbb{N}\}$, $gen^{\mathrm{nat}}(n) = \{0, 1, \ldots, n\}$. Note that unlike *gen*, gen^{nat} is *not* a polynomial-time operation.

Our first result shows that adding gen^{nat} to $\mathcal{NRL}^{\mathrm{nat}}$ makes the language already very powerful. Recall that a rudimentary set is a set of tuples of natural numbers definable by a formula of bounded arithmetic in the language L_{PA} of Peano arithmetic [15]. Equivalently, it is the class of the linear time hierarchy languages over the natural numbers. A function is *rudimentary* if it is majorized by a polynomial and its graph is a rudimentary set.

Proposition 14. *All rudimentary functions are definable in* $\mathcal{NRL}^{\mathrm{nat}}(gen^{\mathrm{nat}})$. □

Thus, $\mathcal{NRL}^{\mathrm{nat}}(gen^{\mathrm{nat}})$ is very powerful; for example, there are NP-complete rudimentary sets that are definable in this language. However, not all polynomial-time computable functions are definable in $\mathcal{NRL}^{\mathrm{nat}}(gen^{\mathrm{nat}})$. For example, the function $d(x, y) = 2^{|x| \cdot |y|}$, where $|\, x\,|$ is the length of the binary representation of x, is in PTIME [15], but every function in $\mathcal{NRL}^{\mathrm{nat}}(gen^{\mathrm{nat}})$ is majorized by a polynomial. We do not know if adding the function $d(x, y)$ to $\mathcal{NRL}^{\mathrm{nat}}(gen^{\mathrm{nat}})$ suffices to capture all PTIME functions on natural numbers, but we can show the following, using Cobham's characterization of PTIME, cf. [18].

Proposition 15. *Every polynomial-time computable function on natural numbers is definable in* $\mathcal{NRL}^{\mathrm{nat}}(gen^{\mathrm{nat}})$ *augmented with* $d(x, y)$ *and the set bounded fixpoint construct.* □

One might ask why the function $|\, x\,|$ is not mentioned in Proposition 15. It turns out that this function is definable as the cardinality of the set $S_x = \{y \mid y \leq x, y = 2^w \text{ for some } w\}$. Notice that the following first-order formula (cf. [15]) $\phi(y) \equiv \forall v \leq y\ \forall u < y.\ (v \neq 1 \wedge u \cdot v = y) \to (\exists z < v.\ 2 \cdot z = v)$ holds iff y is a power of 2, and thus this test can be expressed in $\mathcal{NRL}^{\mathrm{nat}}(gen^{\mathrm{nat}})$ since all the quantification is bounded.

However, the language of Proposition 15 is very powerful. We can use some of the results from [4] to show the following.

Proposition 16. *Let f be a EXPTIME-complexity function on natural numbers such that f is majorized by a polynomial. Then f is definable in* $\mathcal{NRL}^{\mathrm{nat}}(gen^{\mathrm{nat}})$ *augmented with* $d(x, y)$ *and the set bounded fixpoint construct.* □

Thus, it is hard to find a reasonable balance between equally expressive and tractable languages over bags, and languages over sets with aggregate functions.

6 Conclusions and open problems

We presented preliminary results on increasing the expressive power of languages for bag-based complex objects without losing tractability. We defined deflationary and bounded inflationary fixpoint operators and showed that they are equally expressive and strictly more expressive than their set-based counterparts. We showed that these fixpoint operators are not sufficient to capture the class of all PTIME queries over bags and that the *gen* operator fills the gap. Finally, we studied the effects of adding the fixpoint operators and the *gen* primitive to languages with aggregate functions.

We now discuss some of the problems currently under investigation. The use of nesting is a key technique in achieving the characterization of PTIME over bags. We would like to find a language that captures PTIME over *flat* bags. Of course, one can just use $\mathcal{BQL}$ restricted to queries from flat inputs to flat outputs, but it would be desirable to find a natural flat language. Note that the conservative extension property [17] does not help us here, because $\mathcal{BQL}$ only possesses this property beyond the first level of nesting.

We know that the class of queries $\text{PTIME}^{\text{bag}}$ is captured by $\mathcal{BQL}(\leq, bfp, gen)$. Can we obtain similar characterizations for other complexity classes, for example, L^{bag}, NL^{bag} and NC^{bag}? For example, a characterization of NC queries over nested relations that uses divide-and-conquer recursion was given in [20]. Does a similar recursion mechanism (essentially the structural recursion on the union presentation, cf. [2]), when added to $\mathcal{BQL}(\leq, gen)$, capture NC^{bag}? More generally, let $\mathcal{C}$ be a complexity class, and $\mathcal{L}$ a set language of the form $\mathcal{NRA}(\mathbf{p}, \leq)$ that captures all $\mathcal{C}$ queries over sets. Here $\mathcal{NRA}$ is the nested relational algebra and $\mathbf{p}$ is some family of primitives. Is there a systematic way of deriving a new family of bag primitives $\mathbf{p}_b$ such that $\mathcal{BQL}(\leq, \mathbf{p}_b, gen)$ captures $\mathcal{C}^{\text{bag}}$? Note that *gen* has to be included for any class above L, unless it is definable with $\mathbf{p}_b$.

We are continuing to investigate the relationship between $\mathcal{BQL}$ and $\mathcal{NRL}^{\text{nat}}$ in the presence of *gen* and fixpoints. The power of gen^{nat} seems to be essential to express many arithmetic operations (e.g., minimization or various primitive recursion schemas), but makes it hard to find a tractable language with aggregates that would be equally expressive as some tractable bag language. However, a related operator that takes an n-element set and returns $\{0, \ldots, n\}$ *is* expressible in $\mathcal{NRL}^{\text{nat}}$ in the presence of order. We believe that this observation may help us model more arithmetic in $\mathcal{NRL}^{\text{nat}}$ without gen^{nat} and thus find a reasonable balance between tractable bag languages and set languages with aggregates.

Acknowledgements We thank Limsoon Wong and Tim Griffin for their comments. The first author would also like to thank Dirk Van Gucht for several insightful discussions during the early stages of this work.

References

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

2. P. Buneman, S. Naqvi, V. Tannen, L. Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, September 1995.
3. S. Chaudhuri and M. Vardi. Optimization of *real* conjunctive queries. In *Proceedings of the 12th Symposium on Principles of Database Systems*, Washington DC, 1994.
4. P. Clote. Computation models and function algebras. *Proc. Logic and Computational Complexity*, Springer LNCS 960, 1994, pages 98–130.
5. L. S. Colby, E. L. Robertson, L. V. Saxton, and D. Van Gucht. A query language for list-based complex objects. In *Proceedings of the 13th Symposium on Principles of Database Systems*, pages 179–189, Minneapolis, MN, 1994.
6. T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedings ACM SIGMOD*, 1995, pages 328–339.
7. S. Grumbach, L. Libkin, T. Milo and L. Wong. Query languages for bags: expressive power and complexity. *SIGACT News* 27(2): 30–37, 1996.
8. S. Grumbach and T. Milo. Towards tractable algebras for bags. In *Proceedings of the 12th -SIGART Symposium on Principles of Database Systems*, pages 49–58, Washington, DC, May 1993.
9. S. Grumbach and T. Milo. An algebra for pomsets. In *Proceedings of the International Conference on Database Theory*, Prague, 1995, pages 191–207, 1994.
10. S. Grumbach and V. Vianu. Tractable query languages for complex object databases. *J. Comput. and Syst. Sci.* 51(2): 149–167, 1995.
11. Y. Gurevich and S. Shelah. Fixed-point extensions of first-order logic. *Annals of Pure and Applied Logic* 32 (1986), 265–280.
12. M. Gyssens and D. Van Gucht. A comparison between algebraic query languages for flat and nested databases. *Theoretical Computer Science* 87 (1991), 263–286.
13. M. Gyssens, D. Van Gucht and D. Suciu. On polynomially bounded fixpoint construct for nested relations. In *Proceedings of 5th Workshop on Database Programming Languages*, Gubbio, Italy, 1995. Available as Springer Electronic WiC publication.
14. N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68:86–104, 1986.
15. J. Krajíček. *Bounded Arithmetic, Propositional Logic, and Complexity Theory.* Cambridge University Press, 1995.
16. L. Libkin and L. Wong. Some properties of query languages for bags. In *Proceedings of the 4th Workshop on Database Programming Languages*, Springer Verlag, 1994.
17. L. Libkin and L. Wong. Query languages for bags and aggregate functions. *J. Comput. and Syst. Sci.*, to appear. Extended abstract in *PODS'94*.
18. H. Rose. *Subrecursion: Functions and Hierarchies.* Oxford, 1984.
19. D. Suciu. Fixpoints and bounded fixpoints for complex objects. In *Proceedings of the 4th Workshop on Database Programming Languages*, Springer Verlag, 1994.
20. D. Suciu and V. Tannen. A query language for NC. In *Proceedings of the 13th Symposium on Principles of Database Systems*, Minneapolis, MN, 1994.
21. M. Vardi. The complexity of relational query languages. In *Proceedings of the 14th ACM Symposium on Theory of Computing*, pages 137–146, 1982.

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