A Query Language for List-Based Complex Objects

Latha S. Colby
Data Parallel Systems, Inc.
colby@dpsi.com

Edward L. Robertson
Indiana University
edrbtsn@cs.indiana.edu.

Lawrence V. Saxton
University of Regina
saxton@cs.uregina.ca

Dirk Van Gucht
Indiana University
vgucht@cs.indiana.edu.

Abstract

We present a language for querying list-based complex objects. The language is shown to express precisely the polynomial-time generic list-object functions. The iteration mechanism of the language is based on a new approach wherein, in addition to the list over which the iteration is performed, a second list is used to control the number of iteration steps. During the iteration, the intermediate results can be moved to the output list as well as re-inserted into the list being iterated over. A simple syntactic constraint allows the growth rate of the intermediate results to be tightly controlled which, in turn, restricts the expressiveness of the language to PTIME.

1 Introduction

Until the mid to late 1980's, the theory of query languages was predominantly developed in the context of database models that support *non-recursive* data types that can be defined from a family of basic types via a finite number of applications of the *tuple* and *finite-set* type constructors. Research during the late 1980's and early 1990's broadened the allowed data types in two respects. First, and driven by the object-oriented methodology, the *pointer (i.e., reference)* type was added to accommodate recursive data types and object creation. Secondly, and driven by the needs of specialized data applications [MV93], data types such as *multisets* (also called bags) [GM93, LW93], *finite-lists* [GW92, CSV94], *streams* [PSV92] etc., were added.

This paper is situated in the theory of query languages for database models that support non-recursive data types defined via the tuple and finite-*list* type constructors. We call these types collectively the *list-based complex object* types (or simply *list-object* types). Observe that the finite-set types and recursive types are not considered list-based complex object types. The main contribution of this paper is the introduction of a query language, called $\mathcal{L}_p$, which characterizes the class of feasible, i.e., polynomial-time queries over arbitrary list-based complex objects. We will first summarize results concerning set-based query languages, and relate and compare these with our results.

Theoretical issues related to set-based query languages have been investigated by several researchers (e.g., [AV90, AV91, CH82, CH80, Imm86, Var82]). The earliest significant result in this area, relevant to the research presented in this paper, was the theorem by Immerman [Imm86] and Vardi [Var82] stating that the query language FO + lfp (i.e., first-order logic augmented with the least fixed point construct) characterizes the class of polynomial time queries over *flat, ordered* relational databases. In sharp contrast, in the case of nested relations (relations whose values may be relations in turn), when the lfp construct is added to the nested relational algebra, the language is equivalent to the nested algebra + powerset [GV88]. Similar languages were studied in a more general setting of set-based (non-recursive) complex objects by Abiteboul and Beeri [AB87], Kuper and Vardi [KV93] and Hull and Su [HS89, HS93]. These languages were shown to express exactly all the *elementary queries* by Hull and Su [HS91]. It has been shown, recently, by Suciu [Suc93] that if a *bounded* fixed point operator were added to the nested relational algebra (as opposed to the unbounded lfp construct used in the aforementioned papers), the resulting language expresses precisely the PTIME queries on *ordered* nested relations. There have also been several interesting efforts to apply functional programming paradigms, in particular *list comprehension* [Tri91], *structural recursion* [BTBN91, BTBW92], and *typed lambda calculus* [HKM93] to the design of set-based query languages.

SIGMOD/PODS 94 - 5/94 Minneapolis, Minnesota USA
© 1994 ACM 0-89791-639-5/94/0005..$3.50

The core idea in these paradigms is to reason about sets as terms representing *duplicate-free* lists or a sequence of set constructions, to use structural recursion to manipulate these terms, and to interpret these manipulations as if they were done on the actual sets. In [IPS91a, IPS91b], Immerman, Patnaik and Stemple presented an elegant, functionally oriented language for sets called SRL (the set-reduce language). This language uses an iterative construct called set-reduce whose traversal depends on the order of the elements in the set (intuitively, set-reduce views the set as a duplicate free list). They show how different complexity classes can be captured by introducing syntactic restrictions on SRL. In particular, when the level of nesting of sets is restricted to a maximum of one, SRL captures precisely the PTIME queries over ordered sets.

It is tempting to think that the techniques mentioned above can be straightforwardly adapted to yield a list-based language for the feasible queries. However there are serious technical difficulties, some specifically due to the properties of lists and others related to the properties of complex objects. The latter difficulties are easy to appreciate in view of the previously mentioned results by Hull and Su etc., but can be overcome by a straightforward restriction on the iteration construct as shown by Suciu. The former difficulties are more subtle. Clearly, it is unlikely that a natural adaptation of the Immerman-Vardi approach can work for lists. The finite convergence property for their least-fixed point construct is guaranteed by the facts that no new atomic values are introduced by the query language *and* by the duplicate-free nature of sets. It is obviously this second fact which does not extend to the list setting. (In this regard, the **for**-loop relational algebra [Cha81], which also characterizes the polynomial queries over ordered flat relations, is a much better candidate for adaptation to list query languages.) Even though the functional query language approaches present elegant platforms, difficulties still arise. These difficulties are evident in the SRL language of Immerman, Patnaik and Stemple. As mentioned earlier, when the level of nesting of sets is restricted to a maximum of one, SRL captures precisely the PTIME queries. However, without that restriction, SRL's expressive power grows beyond exponential (this is the difficulty consistent with the Hull-Su result). However, far more important to this paper is that if SRL is considered in the context of list-types, in particular by allowing duplicates and replacing the set-reduce operator by the corresponding list-reduce operator, SRL climbs in expressive power from PTIME to the expressive power of the *primitive recursive functions* over lists.

We, therefore, believe that the key to designing a PTIME expressive language in the list-based complex object setting is to have a tightly controlled recursion or iteration scheme. The iteration mechanisms in the aforementioned languages such as lfp, set-reduce and structural recursion are either unsuitable or too powerful for this setting. The $\mathcal{L}_p$ language introduced in this paper for list-based complex objects contains a fundamentally different iteration mechanism. The central feature of the language is the list traversal operator which may be described as a (repeated) *filter–map* operator since it applies functions to selected elements during the traversal. Unlike the typical *map* operation, this list traversal operator takes two lists as inputs, where the sequential iteration takes place over one list while global selections are performed on the second. Partial results can be re-inserted into the list being traversed. The convergence of the list traversal operator is (essentially) guaranteed by limiting re-selection and by progressive traversal steps. The expressiveness is limited by using a simple syntactic constraint to tightly control the growth rate of intermediate results that can be recursed over. In fact, this language expresses precisely the (generic) PTIME queries over list-based complex objects. In particular, the language does not require any restrictions on the *height* of the complex objects. Interestingly, extending the type system to recursive types, which allow objects to have arbitrarily deep nesting, increases the complexity of the language beyond PTIME. This behavior is consistent with the results by Hull and Su[HS93] that show that the power of various complex object languages increases from the elementary queries to computable queries when the underlying (set-based) type system is extended from non-recursive to recursive.

The rest of this paper is organized as follows. In Section 2, we establish some preliminaries regarding list-object types and the class of computations on these objects that is relevant to our study. Section 3 contains the description of the $\mathcal{L}_p$ language. In Section 4, we give sketches of proofs showing that $\mathcal{L}_p$ expresses precisely the PTIME queries (full details are in [CSV93]). Examples showing that the complexity is sharply increased either by permitting high growth rates of intermediate data objects involved in the recursion step or by allowing recursive types are presented in Section 5.

2 List-object types and functions

We first define list-object types and their corresponding domains. These types are formed from a basic type and repeated applications of tuple and list type constructors. It is important to note that our list types only support

homogeneous lists, i.e., lists whose elements are all of the same type. We then define list-object functions. We will be interested in list-object functions that are *computable* and *generic*.

2.1 Types and Domains

List-object types have much in common with the *non-recursive* complex object types studied by Abiteboul and Beeri [AB87], and Hull and Su [HS93]. The domain elements of a non-recursive complex object type all have a bounded nesting depth of tuple and set constructions. List-object types are essentially non-recursive complex object types wherein the set-constructor is replaced by the list-constructor.

Let $\mathcal{U}$ be an enumerable set of constants. Let $Dom(\alpha)$ denote the set of domain elements of type α. The set of list-object types $\mathcal{T}$ and Dom are defined recursively from the basic type $\mathcal{B}$, and tuple and list constructors as follows:

1. $\mathcal{B} \in \mathcal{T}$ and $Dom(\mathcal{B}) = \mathcal{U}$;
2. If $\alpha_1, ..., \alpha_n \in \mathcal{T}$, $n \geq 0$, then $[\alpha_1, ..., \alpha_n] \in \mathcal{T}$ and $Dom([\alpha_1, ..., \alpha_n]) = \{[t_1, ..., t_n] \mid \forall\, i\ (1 \leq i \leq n \Rightarrow t_i \in Dom(\alpha_i))\}$;
3. If $\alpha \in \mathcal{T}$ then $(\alpha) \in \mathcal{T}$ and $Dom((\alpha)) = \{(t_1, ..., t_k) \mid (k \geq 0) \wedge \forall\, i\ (1 \leq i \leq k \Rightarrow t_i \in Dom(\alpha))\}$.

A *list-object database schema* is a sequence $< \alpha_1, ..., \alpha_n >$, $n \geq 1$ where $\alpha_i \in \mathcal{T}$ for each i, $1 \leq i \leq n$. A *list-object database instance* of a schema $S = < \alpha_1, ..., \alpha_n >$ is a sequence $< i_1, ..., i_n >$ where $\forall i, 1 \leq i \leq n$, $i \in Dom(\alpha_i)$. The set of all instances of a schema S is denoted $inst(S)$.

2.2 List-object functions

Let S be a database schema and let α be a list-object type. A *list-object function* f from S to α, denoted $f : S \longrightarrow \alpha$, is a (partial) function from $inst(S)$ to $Dom(\alpha)$.

We are interested in list-object functions that are *generic* and *Turing computable*.

- Genericity is invariance under permutation. A permutation ψ in $\mathcal{U}$ is extended to tuples and lists in the natural way. A list-object function f is *generic* if for all d and extended $\mathcal{U}$ permutations ψ, either $f(\psi(d)) = \psi(f(d))$, or both $f(\psi(d))$ and $f(d)$ are undefined.
- The definition of Turing computability requires an encoding function μ from $\mathcal{U}$ to $\{\mathbf{0}, \mathbf{1}\}^*$. We assume that our Turing machines have alphabets that include **0** and **1**. We can define μ^* to be the natural extension of μ to list-object database instances (see [HS93]). We say that the Turing machine M *computes* f, relative to μ, if for every d for which $f(d)$ is defined, M halts on input $\mu^*(d)$ with output $\mu^*(f(d))$.

Furthermore, if there exists a polynomial p such that, for each input d for which $f(d)$ is defined, M runs within time $p(|\mu^*(d)|)$, then f is a *polynomial-time* (PTIME) generic list-object function.

3 The $\mathcal{L}_p$ language

In this section, we describe the $\mathcal{L}_p$ language. Expressions in $\mathcal{L}_p$ correspond to list-object functions. In fact, in Section 4 we will show that the class of list-object functions captured by $\mathcal{L}_p$ is the class of polynomial-time generic list-object functions. We begin with the syntax of the language, followed by the semantics. We then show how several interesting computations can be expressed in this language.

3.1 Syntax

We will assume that for each $\alpha \in \mathcal{T}$ there is an infinite enumerable set of α-typed variables $\{x_1^\alpha, x_2^\alpha, ...\}$. Expressions in the $\mathcal{L}_p$ language are built from the following constructs:

1. *Variables*:

   If x^α is a variable (of type α) then x^α is an $\mathcal{L}_p$ expression of type α.

2. *Boolean constructs*:

   (a) If ℓ_1 and ℓ_2 are $\mathcal{L}_p$ expressions of the same type then $\ell_1 = \ell_2$ and $\ell_1 \neq \ell_2$ are $\mathcal{L}_p$ expressions of type *boolean*.[1]

   (b) If a and b are $\mathcal{L}_p$ expressions of type *boolean* then so are $\neg a$, $a \wedge b$ and $a \vee b$.

   (c) If c is an $\mathcal{L}_p$ expression of type *boolean* and ℓ_1 and ℓ_2 are $\mathcal{L}_p$ expressions of type α, then *if* c *then* ℓ_1 *else* ℓ_2 is an $\mathcal{L}_p$ expression of type α.

[1] We can represent *boolean* elements by elements of the type ([]), for example *true* by ([]) and *false* by ().

3. *Tuple constructs*:

   (a) If $\ell_1, ..., \ell_n$ are $\mathcal{L}_p$ expressions of type $\alpha_1, ..., \alpha_n$, $n \geq 0$, then $mktup(\ell_1, ..., \ell_n)$ is an $\mathcal{L}_p$ expression of type $[\alpha_1, ..., \alpha_n]$.

   (b) If ℓ is a $\mathcal{L}_p$ expression of type $[\alpha_1, ..., \alpha_n]$, $n \geq 0$, and i such that $1 \leq i \leq n$, then $\pi_i(\ell)$ is an $\mathcal{L}_p$ expression of type α_i.

4. *List constructs*:

   (a) If α is some type then $()^\alpha$ is an $\mathcal{L}_p$ expression of type (α).[2]

   (b) If ℓ is an $\mathcal{L}_p$ expression of type (α) then $first(\ell)$ is an $\mathcal{L}_p$ expression of type α and $rest(\ell)$ is an $\mathcal{L}_p$ expression of type (α).

   (c) If ℓ_1 is an $\mathcal{L}_p$ expression of type α and ℓ_2 is an $\mathcal{L}_p$ expression of type (α) then $cons(\ell_1, \ell_2)$ is an $\mathcal{L}_p$ expression of type (α).

   We will call the $\mathcal{L}_p$ expressions which can be defined from rules in items 1 through 4 only, the *list–trav*-free $\mathcal{L}_p$ expressions. We are now ready to define our final syntax rule.

5. *Iteration construct*:

   First, we need the concept of a lambda $\mathcal{L}_p$ expression. If ℓ is a $\mathcal{L}_p$ expression of type α and the free variables in ℓ are a subset of $\{x_1^{\alpha_1}, ..., x_n^{\alpha_n}\}$, then $\lambda\langle x_1^{\alpha_1}, ..., x_n^{\alpha_n}\rangle\, \ell$ is a lambda $\mathcal{L}_p$ expression of type α.

   Let α_{truc} and α_{output} be types. Let ℓ_S and ℓ_G be $\mathcal{L}_p$ expressions of types (α_S) and (α_G), respectively. Let c, τ, ω, ρ and δ be lambda expressions as described below.

   c is of the form $\lambda\langle x^{\alpha_S}, w_L^{(\alpha_G)}, w_R^{(\alpha_G)}\rangle\, condition$ and of type *boolean*,

   τ is of the form $\lambda\langle x^{\alpha_S}, w_L^{(\alpha_G)}, w_R^{(\alpha_G)}\rangle\, e_{true}$ and of type α_{true},

   ω is of the form $\lambda\langle x^{\alpha_S}, y^{(\alpha_{true})}\rangle\, e_{output}$ and of type (α_{output}),

   ρ is of the form $\lambda\langle x^{\alpha_S}, y^{(\alpha_{true})}\rangle\, e_{recurse}$ and of type (α_S), and

   δ is of the form $\lambda\langle x^{\alpha_S}, z^{(\alpha_G)}\rangle\, e_{default}$ and of type (α_{output}).

   Let, τ and ρ be *list–trav*-free $\mathcal{L}_p$ expressions. Then, $list\text{–}trav_{c,\tau,\omega,\rho,\delta}(\ell_S, \ell_G)$ is an $\mathcal{L}_p$ expression of type (α_{output}).

[2] We will drop the type superscripts when the meaning is clear.

3.2 Semantics

The $\mathcal{L}_p$ language essentially consists of expressions that can be built from the list and tuple manipulation constructs *first*, *rest*, *cons*, π, *mktup*, boolean constructs, and the list iteration construct *list–trav*. From a given list, the function *first* retrieves the first element, and the function *rest*, the list consisting of all but the first element. The *cons* function inserts elements at the beginning of a list. The tuple functions π and *mktup* are used for accessing a given field of a tuple and constructing a tuple, respectively.

The *list–trav* function is the basic iterative construct for traversing lists. It takes two lists, say ℓ_S and ℓ_G, as inputs. The list ℓ_S is traversed in a sequential fashion (hence "S") where only the first element of the list is involved in the computation at a particular step of the iteration. Also, new elements may be prepended to ℓ_S during the computation. The computation terminates when ℓ_S becomes an empty list. The list ℓ_G, on the the other hand, is examined globally (hence "G") at each step and the list may be modified only by deleting elements.

The *list–trav* construct has five parameters - a condition c, and four transformation functions τ, ω, ρ, and δ which determine the actions performed by *list–trav*. These "behavior" parameters are mnemonically named according to the first letters of their descriptions: τ for **t**rue, ω for **o**utput, ρ for **r**ecurse, and δ for **d**efault. We first give the motivation behind the development of the *list–trav* construct, before describing the actual semantics.

Iteration or recursion schemas may be very broadly classified into two categories. The first category consists of simple *map* type of iteration methods which apply a certain computation, say f, to each element in the list, the results of which are accumulated in the result list. This type of recursion is not powerful enough to express many PTIME computations such as transitive closure. The second category contains more general recursion schemes that allow the result of an application of f to be available for further computation. The least fixed point construct, set-reduce and structural recursion are all examples of this type of recursion. As mentioned earlier, the fixed point construct does not lend itself naturally to a setting in which duplicates are allowed. In the case of set-reduce and structural recursion, the type of computation, f, performed within the recursion has to be tightly controlled, when the recursion is performed in a list-based setting in order to prevent exponential blow-up of the size of the intermediate results. The most

obvious culprit is any sort of doubling function such as $\lambda\langle x\rangle\ append(x, x)$. When f involves such a function, the complexity of the results can grow to exponential.

Our strategy has been to separate the two types of recursion by using different parameters in the *list–trav* construct so that one can place restrictions on the parameters involving the second type of (more general) recursion to control the types of computations that can be performed within the recursion. The parameters ω and δ are used in performing the first type of recursion in which the result of each application of ω or δ is simply accumulated in the result list. The parameter ρ provides the more general recursion capability since its result is re-inserted into the list being iterated over. The restriction that *list–trav* cannot be used recursively in the parameters involved in the more general recursion limits the expressiveness of the language to PTIME queries.

Being able to re-insert and iterate over intermediate results provides the more general recursion capability but requires an explicit way to guarantee termination. This is because, unlike a set, a list can grow indefinitely when the same set of elements are added to it repeatedly. Termination is achieved by using a second input list from which elements can be deleted but not inserted and by reducing the size of at least one of the two lists at every step.

The operation of *list–trav* can be explained by a procedural description. This "program" uses typical imperative language constructs and functions *length*, *concat*, and *rightinsert*, such that $concat(\ell_1, \ell_2)$ is the concatenation of ℓ_1 and ℓ_2 (which obviously must be lists of the same type) and $rightinsert(\ell, e)$ returns the list ℓ with the single element e added to the right end. For example, $rightinsert((a,b),c) = (a,b,c)$ and $rightinsert((),()) = (())$. Note that the programming constructs and auxiliary functions are <u>not</u> part of $\mathcal{L}_p$ but are used only to explain the operation of *list–trav*.

```
list–trav_{c,τ,ω,ρ,δ}(ℓ_S, ℓ_G)                                   1
if ℓ_S = () then                                                  2
   return ()                                                      3
else                                                              4
   x ← first(ℓ_S)                                                 5
   w_L ← (); w_R ← ℓ_G  /* used to split ℓ_G into                 6
                           left and right parts */
   y ← ()               /* accumulates values passed              7
                           to ρ and ω */
   newℓ_G ← ()          /* accumulates values passed              8
                           to recursive list–trav */
   for i ← 1 to length(ℓ_G) do                                    9
      if c(x, w_L, w_R) then                                      10
         y ← rightinsert(y, τ(x, w_L, w_R))                       11
      else                                                        12
         newℓ_G ← rightinsert(newℓ_G, first(w_R))                 13
      endif                                                       14
      w_L ← rightinsert(w_L, first(w_R))                          15
      w_R ← rest(w_R)                                             16
   endfor                                                         17
   if y ≠ () then /* c evaluated to true                         18
                     at least once */
      out ← ω(x, y)                                               19
      newℓ_S ← concat(ρ(x, y), rest(ℓ_S))                         20
   else                                                           21
      out ← δ(x, newℓ_G)                                          22
      newℓ_S ← rest(ℓ_S)                                          23
   endif                                                          24
   return concat(out, list–trav(newℓ_S, newℓ_G))                  25
endif                                                             26
```

The key to understanding this algorithm is the effect of the **for** loop (lines 9 through 17). Think of this loop as moving a cursor through ℓ_G. At each step, the cursor is positioned at some element in ℓ_G; w_L is the sublist of ℓ_G to the left of the cursor and w_R is the sublist to the right of the cursor including the element under the cursor (which we will refer to as the cursor element) as shown in Figure 1.[3] Depending upon the test c (line 10), the cursor element ($first(w_R)$ in the program) is either moved to $newℓ_G$ ($false$ case) or transformed and placed on y ($true$ case). After the **for** loop, the list $newℓ_G$ contains only those cursor elements in ℓ_G for which c evaluated to $false$, i.e., the elements that were under the cursor when c evaluated to $true$ are deleted. If c never evaluates to $true$, y is the empty list and $newℓ_G$ is exactly ℓ_G. Note that x stays set to $first(\ell_S)$.

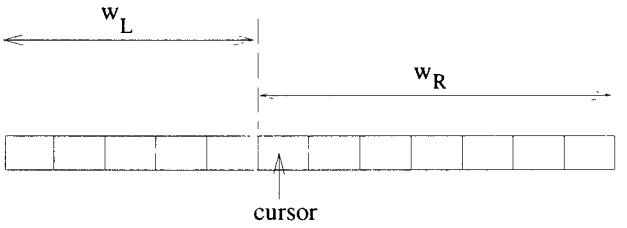

Figure 1:

So, after the **for** loop we have one of two cases:

[3] The two sublists w_L and w_R allow position dependent selections to be performed. For example, if we want to select the last element in ℓ_G we can set c to $\lambda\langle x, w_L, w_R\rangle\ rest(w_R) = ()$.

1. c evaluated to *true* at least once in the **for** loop:

   The list y is then the accumulation of τ applied for every c is *true* case. This list is input to ω and ρ along with x (the first element of ℓ_S). The output list is set to the result of ω, and $new\ell_S$ is set to the result of ρ prepended to $rest(\ell_S)$ (lines 19,20).

2. c evaluated to *false* at every step in the **for** loop (or ℓ_G was empty and the loop was never entered):

   The list y is then the empty list and $new\ell_G$ is ℓ_G. The output list is set to the result of the default function, δ, applied to x and $new\ell_G$, and $new\ell_S$ is set to $rest(\ell_S)$ (lines 22,23).

Finally, the recursive *list–trav* call on $new\ell_S$ and $new\ell_G$ is appended to the output list (line 25).

Example 3.1 Consider the problem of removing duplicates from a list ℓ. The result of duplicate removal from ℓ can be expressed as $list\text{–}trav_{c,\tau,\omega,\rho,\delta}(\ell,\ell)$ where the behavior parameters, denoted collectively as *rdup*, are:

$c:\ \lambda\langle x, w_L, w_R\rangle\, x = first(w_R),\ \ \tau:\ \lambda\langle x, w_L, w_R\rangle\, (),$
$\omega:\ \lambda\langle x, y\rangle\, cons(x, ()),\ \ \rho:\ \lambda\langle x, y\rangle\, (),$ and
$\delta:\ \lambda\langle x, z\rangle\, ()$

For example, if $\ell = (a,b,a,b,c,a)$, then $list\text{–}trav_{rdup}(\ell,\ell) = (a,b,c)$. For this *list–trav* operation, the parameters τ, ρ and δ always return empty lists. At each step of the iteration on ℓ_S, the condition c is evaluated whereby the first element of ℓ_S, say x, is compared with each element in ℓ_G. If at least one element in ℓ_G matches x, then all of the elements in ℓ_G that are equal to x are deleted and x is added to the output list as a result of ω. Note that the variable y in the program is a list containing the result of τ evaluated for each element in ℓ_G satisfying c. So, since τ always returns an empty list in this example, y is a list containing as many empty lists as the number of elements that matched x. If there are no elements in ℓ_G that match x, then x is a duplicate occurrence of some element and x is not added to the output list (since δ returns the empty list). Since ρ always returns an empty list, no elements are ever inserted in ℓ_S. Figure 2 shows the contents of the argument lists ℓ_S and ℓ_G for each recursive step, along with the value of x during each recursion and each *out* value prepended to the final result of the recursive calls (line 25). ■

Example 3.2 As another example, consider the problem of reformatting a list of elements into groups of identical elements. For example, given the list (a,b,a,b,c,a), construct the list $((a,a,a),(b,b),(c))$. This can be expressed as $list\text{–}trav_{group}(\ell,\ell)$, where ℓ is the input list and *group* refers to the following set of behavior parameters.

step	x	ℓ_S	ℓ_G	*out*
0	a	(a,b,a,b,c,a)	(a,b,a,b,c,a)	(a)
1	b	(b,a,b,c,a)	(b,b,c)	(b)
2	a	(a,b,c,a)	(c)	$()$
3	b	(b,c,a)	(c)	$()$
4	c	(c,a)	(c)	(c)
5	a	(a)	$()$	$()$
6		$()$	$()$	

Figure 2:

$c:\ \lambda\langle x, w_L, w_R\rangle\, x = first(w_R),\ \ \tau:\ \lambda\langle x, w_L, w_R\rangle\, x,$
$\omega:\ \lambda\langle x, y\rangle\, cons(y, ()),\ \ \rho:\ \lambda\langle x, y\rangle\, (),$ and $\delta:\ \lambda\langle x, z\rangle\, ().$

The main difference between this *list–trav* and the one in the previous example is that the result of ω is a list containing the list of all occurrences of the element being matched. ■

The ρ argument of *list–trav* in the last two examples always returned an empty list and thus no elements were inserted into the first input list. The results of ω or δ, at any step in the computation, are moved to the output list but not used in the rest of the computation. Many problems, such as transitive closure of a relation, require intermediate results to be available for further computation. The following example illustrates the use of ρ in formulating such queries.

Example 3.3 Given a list of tuples denoting parent-child relationships, the descendants of a given element, say d, can be determined as follows. Let ℓ denote the list containing the parent-child tuples. We first construct the list containing all the tuples of the form $[d,x]$, by applying $list\text{–}trav_{select}((d),\ell)$, where *select* is defined as the parameter functions:

$c:\ \lambda\langle x, w_L, w_R\rangle\, x = \pi_1(first(w_R)),$
$\tau:\ \lambda\langle x, w_L, w_R\rangle\, first(w_R),\ \ \omega:\ \lambda\langle x, y\rangle\, y,$
$\rho:\ \lambda\langle x, y\rangle\, (),$ and $\delta:\ \lambda\langle x, z\rangle\, ().$

Let ℓ' denote the list obtained above. Now, the descendants can be computed by applying $list\text{–}trav_{desc}$ to (ℓ',ℓ) where *desc* is defined as follows:

$c:\ \lambda\langle x, w_L, w_R\rangle\, \pi_2(x) = \pi_1(first(w_R)),$
$\tau:\ \lambda\langle x, w_L, w_R\rangle\, mktup(\pi_1(x), \pi_2(first(w_R))),$
$\omega:\ \lambda\langle x, y\rangle\, cons(x, ()),\ \ \rho:\ \lambda\langle x, y\rangle\, y,$ and
$\delta:\ \lambda\langle x, z\rangle\, cons(x, ()).$

Finally duplicates can be removed as shown in Example 3.1. ■

3.3 Further Examples

We define several interesting operations in terms of the *list–trav* operator. We show how each operation can be simulated by defining the corresponding behavior parameters, c, τ, ω, ρ and δ. For convenience we will denote any $\mathcal{L}_p$ expression that is of the form $\lambda\langle x_1, ..., x_n\rangle\,()$, as ϕ. Observe that when c is set to $false$ the parameters τ, ω and ρ are irrelevant and *list–trav* behaves like a *map* operation applying δ during the traversal.

1. **Singleton Product:** $list\text{–}trav_{sprod}$
   $c : \lambda\langle x, w_L, w_R\rangle\, false,\ \tau : \phi,\ \omega : \phi,\ \rho : \phi,$
   $\delta : \lambda\langle x, z\rangle\, if\ z = ()\ then\ ()$
   $else\ cons(mktup(first(z), x), ())$

   Example: Let $\ell_1 = (x, y, z)$ and $\ell_2 = (a)$, then $list\text{–}trav_{sprod}(\ell_1, \ell_2) = ([a, x], [a, y], [a, z])$.
   Note that since c is always $false$, no elements are ever deleted from ℓ_2 and hence the z variable in δ is always bound to ℓ_2.

2. **Cartesian Product:** $list\text{–}trav_{prod}$
   $c : \lambda\langle x, w_L, w_R\rangle\, false,\ \tau : \phi,\ \omega : \phi,\ \rho : \phi,$
   $\delta : \lambda\langle x, z\rangle\, list\text{–}trav_{sprod}(z, cons(x, ()))$

   Example: Let $\ell_1 = (a, b)$ and $\ell_2 = (x, y, z)$, then $list\text{–}trav_{prod}(\ell_1, \ell_2) =$
   $([a, x], [a, y], [a, z], [b, x], [b, y], [b, z])$.

3. **List Reversal:** $list\text{–}trav_{lrev}$
   $c : \lambda\langle x, w_L, w_R\rangle\, rest(w_R) = (),$
   $\tau : \lambda\langle x, w_L, w_R\rangle\, first(w_R),\ \omega : \lambda\langle x, y\rangle\, y,$
   $\rho : \phi,\ \delta : \phi$

   $list\text{–}trav_{lrev}(\ell, \ell)$ produces a list that is the reverse of ℓ.

4. **Append:**
   $append(l_1, l_2) \stackrel{\text{def}}{=}$
   $if\ (l_1 = ())\ then\ l_2\ else\ list\text{–}trav_{app}(l_1, l_2)$, where $list\text{–}trav_{app}$ is defined as follows:
   $c :\ \lambda\langle x, w_L, w_R\rangle\, w_L = (),$
   $\tau :\ \lambda\langle x, w_L, w_R\rangle\, first(w_R),\ \omega :\ \lambda\langle x, y\rangle\, y,$
   $\rho :\ \lambda\langle x, y\rangle\, cons(x, ()),\ \delta :\ \lambda\langle x, z\rangle\, cons(x, ())$

   Example: $\ell_1 = (a, b, c)$ and $\ell_2 = (d, e, f)$ then $append(\ell_1, \ell_2) = (d, e, f, a, b, c)$.

5. **Flatten:** $list\text{–}trav_{flat}$
   $c : \lambda\langle x, w_L, w_R\rangle\, false,\ \tau : \phi,\ \omega : \phi,\ \rho : \phi,$
   $\delta : \lambda\langle x, z\rangle x$

   Example: Let $\ell = ((a), (b), (b, c, d), (), (a, a), (d))$, then $list\text{–}trav_{flat}(\ell, ()) = (a, b, b, c, d, a, a, d)$.

6. **Length-Comparison:** $list\text{–}trav_{is-longer}$
   $c : \lambda\langle x, w_L, w_R\rangle\, w_L = (),\ \tau : \phi,\ \omega : \phi,\ \rho : \phi,$
   $\delta : \lambda\langle x, z\rangle\, cons(x, ())$

   Given two lists ℓ_1 and ℓ_2, $list\text{–}trav_{is-longer}(\ell_1, \ell_2)$ returns a non-empty list if ℓ_1 is greater in length than ℓ_2 and an empty list otherwise.

7. **Integer Division:** $list\text{–}trav_{div}$
   $c : \lambda\langle x, w_L, w_R\rangle\, list\text{–}trav_{is-longer}(x, w_L) \neq (),$
   $\tau : \lambda\langle x, w_L, w_R\rangle\, first(w_R),$
   $\omega : \lambda\langle x, y\rangle\, if\ y = x\ then\ cons(first(y), ())\ else(),$
   $\rho : \lambda\langle x, y\rangle\, cons(x, ()),\quad \delta : \phi$

   Given two lists ℓ_1 and ℓ_2 where ℓ_1 contains a list containing the unary representation of a number n_1 and ℓ_2 contains the unary representation of n_2, $list\text{–}trav_{div}(\ell_1, \ell_2)$ returns the result of performing the integer division of n_2 by n_1. For example, if $\ell_1 = ((1, 1))$ and $\ell_2 = (1, 1, 1, 1, 1, 1, 1)$, then $list\text{–}trav_{div}(\ell_1, \ell_2) = (1, 1, 1)$. However, if the divisor is 0, i.e., $\ell_1 = (())$, $list\text{–}trav_{div}$ will return 0, i.e., the empty list.

4 Complexity of the $\mathcal{L}_p$ language

In this section, we will show that the $\mathcal{L}_p$ language characterizes the class of PTIME generic list-object functions. We begin with a simple lemma.

Lemma 1 *Let e be an $\mathcal{L}_p$ expression. Then e is a generic list-object function*[4]. (The proof is by simple structural induction on e.)

The proof that $\mathcal{L}_p$ is a subset of the PTIME (generic) list-object functions involves two list measures: *size* and *maxlength*. Let d be a domain element of type α.

$$\begin{aligned} size(d) &= |\,\mu(d)\,|, \text{ if } d \text{ is of type } \mathcal{B} \\ &= 2 + \textstyle\sum_{i=1}^{k} size(t_i), \\ &\quad \text{if } d = (t_1, ..., t_k) \vee \\ &\quad\ \ d = [t_1, ..., t_k],\ k \geq 0 \end{aligned}$$

$$\begin{aligned} maxlength(d) &= |\,\mu(d)\,|, \text{ if } d \text{ is of type } \mathcal{B} \\ &= max(k, max_{1 \leq i \leq k}(maxlength(t_i))), \\ &\quad \text{if } d = (t_1, ..., t_k) \vee \\ &\quad\ \ d = [t_1, ..., t_k],\ k \geq 0 \end{aligned}$$

The measures *size* and *maxlength* can be extended to instances in the obvious way. Obviously $maxlength(d) \leq size(d)$. Because we work with non-recursive data

[4] For simplicity, we overload the notation e to also denote the list-object function defined by the semantics of $\mathcal{L}_p$ for the expression e.

types, the *maxlength* measure is also polynomially related to the *size* measure. In particular, for each $\mathcal{L}_p$ type α, there exist constants $a_\alpha \geq 1$, $b_\alpha \geq 1$ and $c_\alpha \geq 1$ such that for all d in $Dom(\alpha)$, $size(d) \leq a_\alpha(maxlength(d))^{b_\alpha} + c_\alpha$. (The constant b_α is essentially determined by the (fixed) height of type α.)

Proposition 1 *Let e be an $\mathcal{L}_p$ expression. There exists a polynomial p_e that governs the time complexity of e, i.e., for every valid input d_e to e, $Time(e(d_e)) \leq p_e(size(d_e))$.*

Proof: (Sketch) Note that the proposition is clearly valid if e is a *list–trav*-free $\mathcal{L}_p$ expression. Next, we observe the following important property of *list–trav*-free expressions.

Lemma 2 *Let e' be a list–trav-free expression. Then, there exists a constant $k_{e'}$ such that for every valid input $d_{e'}$ to e', if $e'(d_{e'})$ is defined then $maxlength(e'(d_{e'})) \leq maxlength(d_{e'}) + k_{e'}$.* (The intuition behind the proof of this lemma is that each of the subexpressions of e' either decreases the *maxlength* measure or at most adds a constant (independent of $d_{e'}$) to this measure. Notice that *first*, *rest*, and π decrease the *maxlength* while *cons* and *mktup* increase the maxlength by a constant.)

Lemma 2, the semantics of the list-traverse operation, and a simple complexity argument imply the following:

Lemma 3 *Let lt be a list–trav-operation. Then, there exists a polynomial p_{lt} that governs the size of the input arguments to subsequent (recursive) invocations of lt, i.e., for each pair of valid input lists ℓ_S and ℓ_G to lt, the size of the input arguments to subsequent lt operations is bounded from above by $p_{lt}(size(\ell_S, \ell_G))$.* (The intuition behind the proof of this lemma is that the "c evaluated to *true* at least once" case (and hence the concatenation of the result of ρ to $\text{rest}(\ell_S)$) can occur at most $length(\ell_G)$ times and ℓ_G can only decrease in size between recursive invocations of *list–trav*. Consider the state of the variables x and y at line 20 of the *list–trav* algorithm (in this case c evaluated to *true* at least once). As a consequence of Lemma 2, the semantics of *maxlength*, and the fact that ρ and τ are *list–trav*-free, it can be shown that $maxlength(\rho(x,y)) \leq maxlength(maxel(\ell_S), \ell_G) + k$, where k is a constant independent of ℓ_S and ℓ_G, and $maxel(\ell_S)$ is the first element of ℓ_S with the maximum *maxlength* measure. Using these observations and an inductive argument based on the maximum number of times ρ can be invoked, we can show that the *maxlength* of the first argument of any intermediate *list–trav* call is bound by a polynomial on the *maxlength* of the original inputs ℓ_S and ℓ_G. Because of the polynomial relationship between *size* and *maxlength*, the *size* of the first argument can increase only polynomially.)

By using Lemma 3 and appropriate structural induction on e, the proof of the proposition becomes routine. Lemma 3 is truly at the heart of this proof. It is this lemma which exercises the control that limits the growth rate of intermediate data over which *list–trav* operations recurse to a polynomial. ■

Next we establish that each PTIME generic list-object function can be expressed by an $\mathcal{L}_p$ expression. We first establish the following technical lemma.

Lemma 4 *Let M_f be a polynomial time TM computing the list function f in the context of an encoding function μ. Then, there exists an $\mathcal{L}_p$ expression $M_f(\mathcal{L}_p)$ which emulates M_f. Furthermore, the time complexity associated with $M_f(\mathcal{L}_p)$ is polynomially related to that of M_f.*

Proof: (Sketch). The M_f simulation can be expressed in terms of $list\text{–}trav_M(\ell_1, \ell_2)$, where the list ℓ_1 contains the configuration of M_f at each step and the list ℓ_2 serves as the "clock" that controls the number of steps. (Details of the expression $list\text{–}trav_M$ are omitted.) The list ℓ_1 is a singleton list which contains an element of the form $[s, a, (a_1, a_2, ..., a_n), (a_{-1}, a_{-2}, ..., a_{-m})]$. The interpretation of this element is as follows: M_f is in state s with a under the read/write head, $a_1, a_2, ..., a_n$ lie to the right of the read/write head (reading left to right) and $a_{-1}, a_{-2}, ..., a_{-m}$ lie to the left of the read/write head (reading right to left). That is, the tape holds $a_{-m}...a_{-2}a_{-1}a\,a_1a_2...a_n$ with the portions to the left of a_{-m} and to the right of a_n containing the # symbol.

If $p = an^k + b$ is the polynomial that governs the time complexity of M_f, then we can construct an $\mathcal{L}_p$ expression that produces a list ℓ_2 such that $size(\ell_2) = p$.

The action of $list\text{–}trav_M(\ell_1, \ell_2)$ is to simulate M_f's behavior by changing ℓ_1. The list ℓ_1 is output when $list\text{–}trav_M$ terminates, which occurs either when a halt state is reached or when ℓ_2 becomes empty. We can then examine the first field to determine if M_f reached a final state in which case the third field will represent the output of M_f. ■

Proposition 2 *Let f be a PTIME generic list-object function. There exists an $\mathcal{L}_p$ expression e_f such that e_f defines f.*

Proof: (Sketch) We first make the following observations about generic functions.

Lemma 5 *For any domain element d let $\sigma(d)$ denote the set of atomic elements appearing in d. If f is a generic list-object function, then $\sigma(f(d)) \subseteq \sigma(d)$.*

Lemma 6 *Let f be a generic list-object function and let M be a Turing Machine computing f relative to μ (an encoding function from $\mathcal{U}$ to $\{\mathbf{0},\mathbf{1}\}^*$). Consider a different encoding function ν from $\mathcal{U}$ to $\{\mathbf{0},\mathbf{1}\}^*$. M also computes f relative to ν.*

Lemma 5 essentially states that generic functions are also *domain preserving*, i.e., they do not invent new values while Lemma 6 states that their computations cannot depend upon a particular encoding. The proofs of these lemmas are straightforward (see Hull and Su [HS93]). We now give the sketch of the proof for Proposition 2.

Let μ be a fixed encoding of $\mathcal{U}$. Since f is a polynomial time generic list-object function, there exists a polynomial TM M_f that computes f. This means that for each legal input argument d to f we have that $M_f(\mu^*(d)) = \mu^*(f(d))$. Since $\mathcal{L}_p$ expressions can involve the (basic) type $\mathcal{B}$, and therefore also its associated enumerable domain $\mathcal{U}$, we need to consider encoding $\mathcal{U}$-elements into words over $\{\mathbf{0},\mathbf{1}\}^*$ according to the encoding function μ. Unfortunately, the encoding function μ is not associated with an $\mathcal{L}_p$ expression, and therefore we can not hope to accomplish this encoding directly in $\mathcal{L}_p$.

However, since f is a generic list-object function, by Lemma 6 it will suffice to use a suitable encoding function μ' which depends on the given input d. This is accomplished by following a strategy similar to that in Hull and Su [HS93]. We first scan the flattened version of the input d for all the unique atomic elements and then build a *dictionary* encoding these, using lists of lists such as $(())$, $((),())$, etc., to represent the TM alphabet symbols. So, we can effectively compute the behavior of f on d by first applying $M_f(\mathcal{L}_p)$ to d encoded according to this dictionary and then mapping back the result. Lemma 5 guarantees that the dictionary is sufficient to unmap the result. ■

5 Increasing the complexity of $\mathcal{L}_p$

In the definition of the *list–trav* function, we did not allow the use of *list–trav* within τ and ρ. This restriction is very crucial to the $\mathcal{L}_p \subseteq P$ result. At each step in the *list–trav* iteration, the result of ρ is concatenated to the first argument of *list–trav*. The growth rate of this argument is limited to a polynomial factor by the aforementioned restriction. Without this restriction, one can express computations that are beyond PTIME as illustrated by the following example.

Let ℓ_1 be the list $((a))$ and ℓ_2 a list containing n number of elements. Then, $\mathit{list\text{–}trav}_{exp}(\ell_1,\ell_2)$, defined below, will produce a list containing 2^n elements. The use of $\mathit{list\text{–}trav}_{app}$ (defined in Section 3.3) in ρ causes doubling of the first element of ℓ_1 at every step.

$c : \lambda\langle x, w_{\mathrm{L}}, w_{\mathrm{R}}\rangle\, w_{\mathrm{L}} = (),\ \ \tau : \phi,\ \ \omega : \phi,$
$\rho : \lambda\langle x, y\rangle\, cons(\mathit{list\text{–}trav}_{app}(x,x),()),$ and $\delta : \lambda\langle x, z\rangle\, x$

List types that are supported by $\mathcal{L}_p$ do not allow heterogeneous lists. If the language were extended to support *recursive* types that allowed heterogeneous lists, i.e., lists with elements of different types, then $\mathcal{L}_p$ would no longer be within PTIME. For example, it would be possible to compute $cons(x,x)$ in such a setting. Thus, $cons((a),(a))$ would return $((a),a)$. It would then be possible to express an exponential function as follows:

$c : \lambda\langle x, w_{\mathrm{L}}, w_{\mathrm{R}}\rangle\, w_{\mathrm{L}} = (),\ \ \tau : \phi,\ \ \omega : \phi,$
$\rho : \lambda\langle x, y\rangle\, cons(cons(x,x),()),$ and $\delta : \lambda\langle x, z\rangle\, x$

6 Conclusions

We have presented a language for querying databases supporting list-based complex objects and shown that it expresses precisely the PTIME generic functions on these objects. The key to limiting the expressive power of this language was to use a tightly controlled recursion scheme. There are many interesting possibilities for further research. We would like to investigate further syntactic restrictions on $\mathcal{L}_p$, that would allow the characterization of other interesting classes of queries, such as, LOGSPACE. We are also interested in seeing how the current framework can be modified to obtain a characterization for *bag-generic* queries. Traversal over ordered sets has been used to obtain a characterization of the PTIME queries over flat relations by many researchers. It would be interesting to see if iteration over lists can be used to obtain a similar characterization over bags.

The technique used here to control the iteration scheme relies on separating the simple *map* type of recursion, wherein intermediate results are not available for further processing, from the fixpoint type of recursion by using different parameters in the *list–trav* construct and then placing a simple restriction on the parameters involving the latter type of recursion. A disadvantage of this approach is that several parameters are required making the *list–trav* construct fairly complex. We are investigating ways of simplifying the iteration scheme by using only the latter type of recursion. In our opinion, this would require more syntactic restrictions on the types of computations that can be performed within the recursion and also on the number of levels of recursion to remain within PTIME. Finally, we are interested in comparing the results presented here to other characterizations of polynomial time functions, such as Bellantoni and Cook's [BC92] recursion-theoretic characterization and Leivant and Marion's [LM93] lambda calculus characterization.

Acknowledgements

The authors would like to thank Jan Van den Bussche, Gerry Allwein and the referees for helpful comments.

References

[AB87] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. In *Proceedings of the International Workshop on Theory and Applications of Nested Relations and Complex Objects*, Darmstadt, 1987.

[AV90] S. Abiteboul and V. Vianu. Procedural languages for database queries and updates. *Journal of Computer and System Sciences*, 41(2):181–229, 1990.

[AV91] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proceedings of the 23rd ACM Symposium on Theory of Computing*, pages 209–219, 1991.

[BC92] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. In *Proceedings of the 24th ACM Symposium on Theory of Computing*, pages 283–293, 1992.

[BTBN91] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query langauge. In *Proceedings of the 3rd International Workshop on Database Programming Languages*, pages 9–19, Nafplion, Greece, August 1991. Morgan Kaufmann.

[BTBW92] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In *LNCS 646: Proceedings of the International Conference on Database Theory*, pages 140–154. Springer-verlag, October 1992.

[CH80] A. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[CH82] A. Chandra and D. Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25:99–128, 1982.

[Cha81] A. Chandra. Programming primitives for database languages. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 50–62, May 1981.

[CSV93] L. S. Colby, L. V. Saxton, and D. Van Gucht. A query language for list-based complex objects. Technical Report 395, Indiana University, December 1993.

[CSV94] L. S. Colby, L. V. Saxton, and D. Van Gucht. Concepts for modeling and querying list-structured data. *Information Processing and Management*, 1994. To appear.

[GM93] S. Grumbach and T. Milo. Towards tractable algebras for bags. In *Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 49–58, Washington, DC, May 1993.

[GV88] M. Gyssens and D. Van Gucht. The powerset algebra as a result of adding programming constructs to the nested relational algebra. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 225–232, Chicago, IL, June 1988.

[GW92] S. Ginsburg and X. Wang. Pattern matching by rs-operations: Towards a unified approach to querying sequenced data. In *Proceedings of the eleventh ACM*

SIGACT-SIGMOD Symposium on Principles of Database Systems, pages 293–300, San Diego, California, 1992.

[HKM93] G. G. Hillebrand, P. C. Kanellakis, and H. G. Mairson. Database query languages embedded in the typed lambda calculus. In *LICS*, 1993.

[HS89] R. Hull and J. Su. Untyped sets, invention, and computable queries. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 347–359, 1989.

[HS91] R. Hull and J. Su. On the expressive power of database queries with intermediate types. *Journal of Computer and System Sciences*, 43(1):219–267, 1991.

[HS93] R. Hull and J. Su. Algebraic and calculus query languages for recursively typed complex objects. *Journal of Computer and System Sciences*, 47:121–156, 1993.

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

[IPS91a] N. Immerman, S. Patnaik, and D. Stemple. The expressiveness of a family of finite set languages. Technical Report 91-96, Computer and Information Science Department, University of Massachusetts, 1991.

[IPS91b] N. Immerman, S. Patnaik, and D. Stemple. The expressiveness of a family of finite set languages. In *Proceedings of the tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 37–52, 1991.

[KV93] G. M. Kuper and M. Y. Vardi. On the complexity of queries in the logical data model. *Theoretical Computer Science*, 116:33–57, 1993.

[LM93] D. Leivant and J-Y. Marion. Lambda calculus characterizations of poly-time. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, 1993. To appear in Fundamenta Informaticae.

[LW93] L. Libkin and L. Wong. Some properties of query languages for bags. In *Proceedings of the Fourth International Workshop on Database Programming Languages*, 1993.

[MV93] D. Maier and B. Vance. A call to order. In *Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–16, Washingtion, DC, May 1993.

[PSV92] D. S. Parker, E. Simon, and P. Valduriez. SVP - a model capturing sets, streams and parallelism. In *Proceedings of the 18th VLDB Conference*, pages 115–126, Vancouver, Canada, 1992.

[Suc93] D. Suciu. Fixpoints and bounded fixpoints for complex objects. In *Proceedings of the Fourth International Workshop on Database Programming Languages*, 1993.

[Tri91] P. Trinder. Comprehensions, a query notation for DBPLs. In *Proceedings of the 3rd International Workshop on Database Programming Languages*, pages 55–68, Nafplion, Greece, August 1991. Morgan Kaufmann.

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