A Query Language for List-Based Complex Objects

Latha S. Colby[*] Lawrence V. Saxton[†] Dirk Van Gucht [‡]

May 1994 (revised)

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.

[*]Data Parallel Systems Inc., 4617 Morningside Dr., Bloomington, IN, 47408; email: colby@dpsi.com

[†]University of Regina, Dept. of Comp. Science, Regina, Saskatchewan S4S 0A2, Canada, email: saxton@cs.uregina.ca

[‡]Indiana University, Comp. Science Dept., Bloomington, IN 47405-4101, email: vgucht@cs.indiana.edu.

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, data types such as *arrays*, *multisets* (also called bags), *finite-lists*, *streams* etc., were added (e.g. [9, 18, 19]).

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., [2, 3, 8, 7, 17, 21]). The earliest significant result in this area, relevant to the research presented here, was the theorem by Immerman [17] and Vardi [21] 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 are relations in turn), when the lfp construct is added to the nested relational algebra, the language is equivalent to the nested algebra + powerset [10]. Similar languages were studied in a more general setting of set-based (non-recursive) complex objects by Abiteboul and Beeri [1] and Hull and Su [12, 14]. These languages were shown to express exactly all the *elementary queries* by Hull and Su [13]. There have also been several interesting efforts to apply functional programming paradigms, in particular *list comprehension* [20], *structural recursion* [4, 5], and *typed lambda calculus* [11] to the design of set-based query languages. 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. Recently, Immerman, Patnaik and Stemple presented an elegant, functionally oriented language for sets called SRL (the set-reduce language) in [15, 16]. 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 and Hull and Su etc. 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 [6], 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, there too difficulties exist. 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[14] 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. The next section contains some preliminaries regarding list-object types and functions. The syntax and semantics of the $\mathcal{L}_p$ language are described in Section 3. The results regarding the complexity of $\mathcal{L}_p$ are established in Section 4.

2 List-object types and functions

In this section, we establish some preliminaries regarding the types of objects and classes of computations on these objects on which our investigations are based. 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.[1] We then define list-object functions. We will be interested in list-object functions that are *computable* (Section 2.2) and *generic* (Section 2.3).

2.1 Types and Domains

List-object types have much in common with the *non-recursive* complex object types studied by Abiteboul and Beeri [1], and Hull and Su [14]. 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.

The set of list-object types $\mathcal{T}$ is defined recursively from the basic type $\mathcal{B}$, and tuple and list constructors and is the smallest set determined as follows:

1. $\mathcal{B} \in \mathcal{T}$.
2. If $\alpha_1, ..., \alpha_n \in \mathcal{T}$, $n \geq 0$, then $[\alpha_1, ..., \alpha_n] \in \mathcal{T}$.
3. If $\alpha \in \mathcal{T}$ then $(\alpha) \in \mathcal{T}$.

Let $\mathcal{U}$ be an enumerable set of (basic) domain elements. Let $Dom(\alpha)$ denote the set of domain elements of type α. Dom is inductively defined as follows:

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

[1]Thus, we will *not* consider heterogeneous lists. A (heterogeneous) list l over $\mathcal{A}$ (some base domain of atoms) is an object $(l_1, ..., l_n)$, $n \geq 0$, such that for each i, $1 \leq i \leq n$, either l_i is an element of $\mathcal{A}$ or l_i is an (heterogeneous) list over $\mathcal{A}$.

The set of all domain elements, i.e. $\bigcup_{\alpha \in \mathcal{T}} Dom(\alpha)$, will be denoted by $\mathcal{I}$.

A *list-object database schema* is a sequence $S =< \alpha_1, ..., \alpha_n >$, $n \geq 1$ where $\alpha_i \in \mathcal{T}$ for each i, $1 \leq i \leq n$. A *list-object database instance* of a schema S 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 $S =< \alpha_1, ..., \alpha_n >$ 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 will only be interested in *computable* list-object functions. To make this notion precise, we adapt some material from Hull and Su [14]. A deterministic *Turing machine* (TM) is a sextuple $M = (K, \Sigma, , , \delta, q_s, q_h)$; K is a finite set of states, Σ is the set of TM working symbols which we assume to be $\{\mathbf{0}, \mathbf{1}, (,), \llbracket, \rrbracket, \|, \#\}$ (we assume that Σ is disjoint from $\mathcal{U}$), $, = \Sigma - \{\#\}$ is the set of input alphabet symbols, δ is the TM transition function (i.e., δ is a function from $K \times \Sigma$ to $K \times \Sigma \times \{left, right\}$), q_s is the starting state and q_h is the halting state. An *instantaneous description* (ID) of M is a quadruple $(q, \alpha, \sigma, \beta)$ where q is the current state, σ the symbol in the tape square where the head is, α and β are two strings of symbols representing the left and the right part of the tape relative to the head.

Let μ be a one-to-one mapping from $\mathcal{U}$ to $\{\mathtt{0}, \mathtt{1}\}^*$. We define $\mu^* : \mathcal{I} \rightarrow , ^*$ as follows:

$$\begin{aligned} \mu^*(d) &= \mu(d), \text{ if } d \in \mathcal{U}; \\ &= \llbracket\ \mu^*(d_1) \| ... \| \mu^*(d_n)\ \rrbracket, \text{ if } d = [d_1, ..., d_n]; \\ &= (\ \mu^*(d_1) \| ... \| \mu^*(d_k)\ ), \text{ if } d = (d_1, ..., d_k). \end{aligned}$$

Intuitively, $\mu^*(d)$ is a sequential enumeration of the object d where the atomic objects are mapped according to μ; the list and tuple end markers and $\|$ serve as delimiters.

Let $f : S \longrightarrow \alpha$, be a list-object function. We say that the Turing machine M *computes* f, relative to μ, if for every valid input $d =< d_1, ..., d_n >$ to f for which $f(d)$ is defined, M halts in state q_h on input $\mu^*(d_1)\mu^*(d_2)...\mu^*(d_n)$ 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) list-object function.

2.3 Generic list-object functions

We next establish the notion of a *generic* list-object function. Generic list functions, as opposed to arbitrary list-object functions, are not allowed to interpret the internal representation of list atoms, i.e., the internal details of the $\mathcal{U}$ elements as determined via μ. So for example, the list-object function

$$reverse :< (\mathcal{B}) > \longrightarrow (\mathcal{B})$$

which reverses a list, is a generic list-object function because list members need not be interpreted. However, the list object function

$$sort :< (\mathcal{B}) > \longrightarrow (\mathcal{B})$$

which sort lists lexicographically according to μ (we assume that $\mathbf{0} < \mathbf{1}$) is not generic.

To define the concept of generic list function, we need some preliminaries. Let ψ be a permutation of the universe $\mathcal{U}$. ψ can be inductively extended to tuple and list domains and finally to database instances as follows. If $[a_1, ..., a_n] \in Dom([\alpha_1, ..., \alpha_n])$ then $\psi([a_1, ..., a_n]) = [\psi(a_1), ..., \psi(a_n)]$. If $(a_1, ..., a_n) \in Dom(list(\alpha))$ then $\psi(a_1, ..., a_n) = (\psi(a_1), ..., \psi(a_n))$. If $< d_1, ..., d_n > \in inst(< \alpha_1, ..., \alpha_n >)$ then $\psi < d_1, ..., d_n > = < \psi(d_1), ..., \psi(d_n) >$.

Let $f : S \longrightarrow \alpha$ be a list-object function. f is a *generic* list-object function if for all d and for all extended $\mathcal{U}$ permutations ψ, either $f(\psi(d)) = \psi(f(d))$, or both $f(\psi(d))$ and $f(d)$ are undefined.

3 The $\mathcal{L}_p$ language

In this section we describe the $\mathcal{L}_p$ language. Expressions in $\mathcal{L}_p$ corresponds 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 an informal description of the semantics (with examples), and then the formal semantics.

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. $Free(x^\alpha) = \{x^\alpha\}$.
2. *Boolean constructs*:
   (a) If l_1 and l_2 are $\mathcal{L}_p$ expressions of the same type then $l_1 = l_2$ and $l_1 \neq l_2$ are $\mathcal{L}_p$ expressions of type *boolean*.[2] $Free(l_1 = l_2) = Free(l_1 \neq l_2) = Free(l_1) \cup Free(l_2)$.

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

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

(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. $Free(\textit{if } c \textit{ then } l_1 \textit{ else } l_2) = Free(c) \cup Free(l_1) \cup Free(l_2)$.

3. *Tuple constructs*:

(a) If $l_1, ..., l_n$ are $\mathcal{L}_p$ expressions of type $\alpha_1, ..., \alpha_n$, $n \geq 0$, then $mktup(l_1, ..., l_n)$ is an $\mathcal{L}_p$ expression of type $[\alpha_1, ..., \alpha_n]$. $Free(mktup(l_1, ..., l_n)) = Free(l_1) \cup ... \cup Free(l_n)$.

(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 $\pi_i(l)$ is an $\mathcal{L}_p$ expression of type α_i. $Free(\pi_i(l)) = Free(l)$.

4. *List constructs*:

(a) If α is some type then $()^\alpha$ is an $\mathcal{L}_p$ expression of type (α).[3] $Free(\ ()^\alpha\ ) = \{\}$.

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

(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 $cons(l_1, l_2)$ is an $\mathcal{L}_p$ expression of type (α). $Free(cons(l_1, l_2)) = Free(l_1) \cup Free(l_2)$.

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 $Free(l) \subseteq \{x_1^{\alpha_1}, ..., x_n^{\alpha_n}\}$, then $\lambda\langle x_1^{\alpha_1}, ..., x_n^{\alpha_n}\rangle\, l$ is a lambda $\mathcal{L}_p$ expression of type α. $Free(\langle x_1^{\alpha_1}, ..., x_n^{\alpha_n}\rangle\, l) = \{x_1^{\alpha_1}, ..., x_n^{\alpha_n}\}$.

Let α_{true} and α_{output} be types. Let l_1 and l_2 be $\mathcal{L}_p$ expressions of types (α_1) and (α_2), respectively. Let c, τ, ω, ρ and δ be lambda expressions as described below.

c is of the form $\lambda\langle x^{\alpha_1}, y_l^{(\alpha_2)}, y_r^{(\alpha_2)}\rangle\, condition$ and of type *boolean*,

τ is of the form $\lambda\langle x^{\alpha_1}, y_l^{(\alpha_2)}, y_r^{(\alpha_2)}\rangle\, e_{true}$ and of type α_{true},

ω is of the form $\lambda\langle x^{\alpha_1}, y^{(\alpha_{true})}\rangle\, e_{output}$ and of type (α_{output}),

ρ is of the form $\lambda\langle x^{\alpha_1}, y^{(\alpha_{true})}\rangle\, e_{recurse}$ and of type (α_1), and

δ is of the form $\lambda\langle x^{\alpha_1}, z^{(\alpha_2)}\rangle\, e_{default}$ and of type (α_{output}).

Let τ and ρ be *list–trav*-free $\mathcal{L}_p$ expressions. Then, $list\text{–}trav_{c,\tau,\omega,\rho,\delta}(l_1, l_2)$ is an $\mathcal{L}_p$ expression of type (α_{output}).

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

3.2 Semantics

3.2.1 Informal Description

The $\mathcal{L}_p$ language essentially consists of expressions that can be built from list and tuple manipulation constructs (*first*, *rest*, *cons*, π, *mktup*), and boolean constructs and the list iteration construct *list–trav*. The function *first* retrieves from a given list, 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 h_1 and h_2, as inputs. The list h_1 is traversed in a sequential fashion 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 inserted into h_1 during the computation. The computation terminates when h_1 becomes an empty list. The list h_2, on the the other hand, is examined globally 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. 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. These parameters determine the type of actions performed by *list–trav* which can be described as follows: Let l_{answer} denote the result of *list–trav*(h_1, h_2). This list can be visualized as consisting of an l_{out} part which is the list that is output at the first step and a l'_{answer} part which is the list generated at the next recursive call. Let x denote the first, (i.e., left most), element of h_1. At each stage of the operation, the condition c is evaluated and the lists h_1 and h_2 are modified as follows: For each element t_i in h_2, the inputs to c are x and the two sublists of h_2, $(t_1, ..., t_{i-1})$ and $(t_i, ..., t_k)$, where $h_2 = (t_1, ..., t_k)$. The inputs to τ are the same as those of c. For each element t_i that satisfies c, i.e., $c(x, (t_1, ..., t_{i-1}), (t_i, ..., t_k))$ is true, $\tau(x, (t_1, ..., t_{i-1}), (t_i, ..., t_k))$ is evaluated and ω and ρ are applied to the list containing the result of these τ's. The inputs to ω and ρ are x and the aforementioned list. The result of ω is the l_{out} output list. The result of ρ is concatenated to (the left of) the list $rest(h_1)$. The element x is deleted (this can be avoided by including x in the list returned by the τ's). The transformation τ is in some sense a local transformation, while ω and ρ are global transformations. The list h_2 is modified by deleting each element t_i in h_2 for which $c(x, (t_1, ..., t_{i-1}), (t_i, ..., t_n))$ is true.

If no elements in h_2 satisfy c, i.e., $\forall i, c(x, (t_1, ..., t_{i-1}), (t_i, ..., t_k))$ is false , then h_1 becomes $rest(h_1)$ and h_2 is not modified. The default function δ is evaluated on x and h_2 to obtain the list l_{out}.

Finally, *list–trav* is invoked on the modified h_1 and h_2 and the result l'_{answer} is appended to l_{out}.

Example 3.1 Consider the problem of removing duplicates from a list l. This can be expressed as $list\text{–}trav_{c,\tau,\omega,\rho,\delta}(l,l)$ where the behavior parameters c, τ, etc., are as follows:

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

For example, if $l = (a, b, a, b, c, a)$, then $list\text{–}trav_{rdup}(l,l) = (a, b, c)$, where $rdup$ denotes the above set of parameters. For this $list\text{–}trav$ operation, the parameters τ, ρ and δ always return empty lists and are hence essentially no-ops. The exact sequence of operations can be explained as follows. Let l_{out} denote the list that is output at each stage and let l_1 and l_2 denote the two lists that $list\text{–}trav$ is operating on. The contents of these three lists after each step is shown in Figure 1.

Step	l_{out}	l_1	l_2
1	(a)	(a, b, a, b, c, a)	(a, b, a, b, c, a)
2	(b)	(b, a, b, c, a)	(b, b, c)
3	$()$	(a, b, c, a)	(c)
4	$()$	(b, c, a)	(c)
5	(c)	(c, a)	(c)
6	$()$	(a)	$()$
7		$()$	$()$

Figure 1:

At each step of the computation, the condition c is evaluated whereby the first element of l_1, say x, is compared with all the elements in l_2. The elements in l_2 that are equal to x are deleted and x is added to the output list as a result of ω. If there are no elements in l_2 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). Note that since ρ always returns an empty list, no elements are ever inserted in l_1. ■

Example 3.2 As another example, consider the problem of reformatting a list of elements into groups of similar 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 $list\text{–}trav_{group}(l,l)$, where l is the input list and $group$ refers to the following set of behavior parameters.

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

The main difference between this *list–trav* and the one in the previous example is that the result of ω is a list containing 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 result of ω or δ, at any given step in the computation, is moved to the output list and is not used in the rest of the computation. Many problems, such as transitive closure of a relation, require the intermediate results to be available for further computation. The following example illustrates the use of ρ in expressing the solution to the transitive closure problem.

Example 3.3 Given a list of tuples denoting a binary relation, the transitive closure of the relation can be expressed in terms of the following *list–trav* operations. We first define $list\text{–}trav_{group_i}$ that takes a list of tuples and groups together tuples that agree on the value of the i-th attribute. (Note that i is not really a parameter. We are using it as a notational convenience).

$c : \lambda\langle x, y_l, y_r\rangle\, \pi_i(x) = \pi_i(first(y_r))$
$\tau : \lambda\langle x, y_l, y_r\rangle\, first(y_r)$
$\omega : \lambda\langle x, y\rangle\, cons(y, ())$
$\rho : \phi$
$\delta : \phi$

For example, if $l = ([a,b],[b,c],[a,c],[b,a][c,a])$, then $list\text{–}trav_{group_1}(l,l)$
$= (([a,b],[a,c]),([b,c],[b,a]),([c,a]))$

Next, we define $list\text{–}trav_{desc}$ which takes as inputs a list of parent-child tuples of a particular parent and the original binary relation list (i.e., all the parent-child tuples) and computes the list of all the descendants of the parent in the first list. For example, if $r = ([a,b],[b,c],[d,a],[d,e],[e,f],[g,h])$ is the original binary relation containing all the parent-child tuples and l is the list $([d,a],[d,e])$, then $list\text{–}trav_{desc}(l,r) = ([d,a],[d,b],[d,c],[d,e],[d,f])$. The behavior parameters in $list\text{–}trav_{desc}$ are as follows:

$c : \lambda\langle x, y_l, y_r\rangle\, \pi_2(x) = \pi_1(first(y_r))$
$\tau : \lambda\langle x, y_l, y_r\rangle\, mktup(\pi_1(x), \pi_2(first(y_r)))$
$\omega : \lambda\langle x, y\rangle\, cons(x, ())$
$\rho : \lambda\langle x, y\rangle\, y$
$\delta : \lambda\langle x, z\rangle\, cons(x, ())$

Finally, we define the function $list\text{–}trav_{trans}$ which takes as input a list of lists, where each list contains the parent-child tuples of a particular parent, and the original binary relation and produces a list of all the descendants. The first list is constructed by applying $list\text{–}trav_{group_1}$ to the binary relation. Duplicates in the result of $list\text{–}trav_{trans}$ can be removed by applying an $list\text{–}trav_{rdup}$ similar to the one described in Example 3.1. For example, if $r = ([a,b],[b,c],[d,a],[d,e],[e,f],[g,h])$ and $l = list\text{–}trav_{group_1}(r,r)$, then $list\text{–}trav_{rdup}(l',l')$ where

$l' = list\text{–}trav_{trans}(l, r)$ is the transitive closure of r. The behavior parameters in $list\text{–}trav_{trans}$ are as follows:

$c : \lambda\langle x, y_l, y_r\rangle\, false$
$\tau : \lambda\langle x, y_l, y_r\rangle\, \phi$
$\omega : \lambda\langle x, y\rangle\, \phi$
$\rho : \lambda\langle x, y\rangle\, \phi$
$\delta : \lambda\langle x, z\rangle\, list\text{–}trav_{desc}(x, z)$ ■

3.2.2 Formal Semantics

Let $\mathcal{X}$ denote the set of all the finite subsets of the set of all variables in $\mathcal{L}_p$. An *assignment* σ, denoted $\sigma : \mathcal{X} \longrightarrow \mathcal{I}$ is a relation mapping variables to domain elements (of the appropriate type). We denote the mapping of a variable x to a domain element a by $(x \leftarrow a)$ and the set of all variables in the domain of an assignment σ by $var(\sigma)$. Formally, $var(\sigma) = \{x \mid (\exists a \in \mathcal{I}) \wedge (x \leftarrow a) \in \sigma\}$.

For any finite subset of variables $X = \{x_1^{\alpha_1}, ...x_n^{\alpha_n}\}$, let σ_X denote the subset of σ that contains the mappings for the variables in X. In other words, $\sigma_X = \{(x \leftarrow a) \mid (\sigma(x) = a) \wedge (x \in X)\}$.

If X is a finite set of variables then we will denote the family of mappings defined over X by $\mathcal{S}_X$, i.e., $\mathcal{S}_X = \{\sigma \mid var(\sigma) = X\}$.

If l is a $\mathcal{L}_p$ expression of type α with $Free(l) = \{x_1^{\alpha_1}, ..., x_n^{\alpha_n}\}$, $n \geq 0$, then the semantics of l, denoted $[\![l]\!]$, is a partial function from the set of assignments $\mathcal{S}_{Free(l)}$ to the set of domain elements $\mathcal{I}$. For an assignment $\sigma \in \mathcal{S}_{Free(l)}$, we will inductively define $[\![l]\!]\sigma$. In this inductive definition, we will adopt the standard convention that a function call is undefined if one of its arguments is undefined. We will use the term $false$ to denote the empty list () and the term $true$ to denote the list ([]) of type ([]).

We can now specify the semantics of the various $\mathcal{L}_p$ expressions.

1. *Variables*:
   $[\![x^\alpha]\!]\sigma = \sigma(x^\alpha)$.
2. *Boolean constructs*:
   (a) $[\![l_1 = l_2]\!]\sigma$ is $true$ if $[\![l_1]\!]\sigma_{Free(l_1)} = [\![l_2]\!]\sigma_{Free(l_2)}$, and is $false$ otherwise. $[\![l_1 \neq l_2]\!]$ is defined analogously.
   (b) $[\![\neg a]\!]\sigma$ is $false$ if $[\![a]\!]\sigma = true$, and is $true$ if $[\![a]\!]\sigma = false$. $[\![a \wedge b]\!]$ and $[\![a \vee b]\!]$ are defined analogously.
   (c) $[\![\textit{if } c \textit{ then } l_1 \textit{ else } l_2]\!]\sigma$ is undefined if $[\![c]\!]\sigma_{Free(c)}$ is different from $true$ and $false$, is $[\![l_1]\!]\sigma_{Free(l_1)}$ if $[\![c]\!]\sigma_{Free(c)} = true$, and is $[\![l_2]\!]\sigma_{Free(l_2)}$ if $[\![c]\!]\sigma_{Free(c)} = false$.

3. *Tuple constructs*:

   (a) $[\![mktuple(l_1, ..., l_n)]\!]\sigma$ is $[\ [\![l_1]\!]\sigma_{Free(l_1)}, ..., [\![l_n]\!]\sigma_{Free(l_n)}\ ]$.

   (b) $[\![\ \pi_i(l)\ ]\!]\sigma$ is the i-th component of $[\![l]\!]\sigma$, when $[\![l]\!]\sigma$ is of the form $[e_1, ..., e_m]$, $1 \leq i \leq m$. If $[\![l]\!]\sigma = []$ then $[\![\ \pi_i(l)\ ]\!]\sigma$ is undefined.

4. *List constructs*:

   (a) $[\![\ ()^\alpha\ ]\!]\sigma$ is equal to $()$. (Note that by definition σ must be the empty set).

   (b) $[\![first(l)]\!]\sigma$ is $first([\![l]\!]\sigma)$, where
   $$\begin{aligned} first(h) &= t_1,\ \text{if } h = (t_1, ..., t_n),\ n > 0 \\ &\quad \text{is undefined, otherwise} \end{aligned}$$
   $[\![rest(l)]\!]\sigma$ is $rest([\![l]\!]\sigma)$, where
   $$\begin{aligned} rest(h) &= (t_2, ..., t_n),\ \text{if } h = (t_1, ..., t_n),\ n > 0 \\ &= (),\ \text{if } h = () \end{aligned}$$

   (c) $[\![cons(l_1, l_2)]\!]\sigma$ is $cons([\![l_1]\!]\sigma_{Free(l_1)}, [\![l_2]\!]\sigma_{Free(l_2)})$, where
   $$cons(d, h) = (d, t_1, ..., t_n),\ \text{if } h = (t_1, ..., t_n),\ n \geq 0$$

5. *Iteration construct*:
   First, we specify the semantics of a parameterized $\mathcal{L}_p$ expression.

   $[\![\lambda\langle x_1^{\alpha_1}, ..., x_n^{\alpha_n}\rangle\, l\ ]\!]\sigma = [\![l]\!]\sigma_{Free(l)}$. (Note that $Free(l) \subset \{x_1^{\alpha_1}, ..., x_n^{\alpha_n}\}$.)

   Then, $[\![list\text{–}trav_{c,\tau,\omega,\rho,\delta}(l_1, l_2)]\!]\sigma$ is equal to $list\text{–}trav_{c,\tau,\omega,\rho,\delta}([\![l_1]\!]\sigma_{Free(l_1)}, [\![l_2]\!]\sigma_{Free(l_2)})$.

We now give the semantics of the *list–trav* function. We first define the (multi-argument) function *concat* which takes a finite number of lists of the same type and returns a list which is the concatenation of these lists. Note that this function is not a part of the $\mathcal{L}_p$ language. It is only used in the definition of *list–trav*.

Let $h_1, ..., h_k$, $k \geq 0$, be domain elements of type (α).

$$\begin{aligned} concat(h_1, ..., h_k) &= (t_{11}, ..., t_{1m_1}, ..., t_{k1}, ..., t_{km_k}),\ \text{if} \\ &\quad h_1 = (t_{11}, ..., t_{1m_1}),\ m_1 \geq 0, ..., h_k = (t_{k1}, ..., t_{km_k}),\ m_k \geq 0 \end{aligned}$$

In addition, we will use the following notation. If h is a list of the form $(t_1, ..., t_k)$, $k \geq 0$, and i such that $1 \leq i \leq k$, then $h|_i$ denotes the sublist $(t_1, ..., t_i)$ and $h|^i$ denotes the sublist $(t_i, ..., t_k)$. Furthermore, $h|_0 = ()$ and $h|^{k+1} = ()$.

Let h_1 and h_2 be $\mathcal{L}_p$ domain elements of type (α_1) and (α_2), respectively. Let $\lambda\langle x, y_l, y_r\rangle$ be the parameter lists for c and τ, $\lambda\langle x, y\rangle$ the parameter lists for ω and ρ and $\lambda\langle x, z\rangle$ the list for δ.

To define $list\text{–}trav_{c,\tau,\omega,\rho,\delta}(h_1, h_2)$, we consider four cases:

(a) *First-argument-empty case*:
If $h_1 = ()$ then $list\text{–}trav_{c,\tau,\omega,\rho,\delta}(h_1, h_2)$ is equal to $()$;

(b) *Condition-false case*:
h_1 is a non-empty list, $h_2 = (t_1, ..., t_k)$ and for *all* i, $0 \leq i \leq k$, $[\![c]\!]\{(x \leftarrow first(h_1)), (y_l \leftarrow h_2|_{i-1}), (y_r \leftarrow h_2|^i)\} = false$ (notice that in particular h_2 could be the empty list). Then $list\text{–}trav_{c,\tau,\omega,\rho,\delta}(h_1, h_2)$ is
$concat([\![\delta]\!]\{(x \leftarrow first(h_1)), (z \leftarrow h_2)\},\ list\text{–}trav_{c,\tau,\omega,\rho,\delta}(rest(h_1), h_2))$;

(c) *Condition-undefined case*:
Both h_1 and h_2 are non-empty lists, $h_2 = (t_1, ..., t_k)$ and for *some* i, $1 \leq i \leq k$, $[\![c]\!]\{(x \leftarrow first(h_1)), (y_l \leftarrow h_2|_{i-1}), (y_r \leftarrow h_2|^i)\}$ is undefined. Then $list\text{–}trav_{c,\tau,\omega,\rho,\delta}(h_1, h_2)$ is undefined;

(d) *Condition-true case*:
Both h_1 and h_2 are non-empty lists, $h_2 = (t_1, ..., t_k)$ and
for *all* i, $1 \leq i \leq k$, $[\![c]\!]\{(x \leftarrow first(h_1)), (y_l \leftarrow h_2|_{i-1}), (y_r \leftarrow h_2|^i)\}$ is defined,
and for *some* i, $1 \leq i \leq k$, $[\![c]\!]\{(x \leftarrow first(h_1)), (y_l \leftarrow h_2|_{i-1}), (y_r \leftarrow h_2|^i)\} = true$.

Define:

$$\begin{array}{rcl} h_{output} & = & [\![\omega]\!]\{(x \leftarrow first(h_1)), (y \leftarrow concat(t'_1, ..., t'_k))\}, \\ h_{recurse} & = & [\![\rho]\!]\{(x \leftarrow first(h_1)), (y \leftarrow concat(t'_1, ..., t'_k))\}, \text{where for } i\ (1 \leq i \leq k), \\ t'_i & = & (),\ \text{if } [\![c]\!]\{(x \leftarrow first(h_1)), (y_l \leftarrow h_2|_{i-1}), (y_r \leftarrow h_2|^i)\} = false, \\ & = & ([\![\tau]\!]\{(x \leftarrow first(h_1)), (y_l \leftarrow h_2|_{i-1}), (y_r \leftarrow h_2|^i)\}),\ \text{otherwise} \\ \text{and } h'_2 & = & concat(t''_1, ..., t''_k),\ \text{where for } i\ (1 \leq i \leq k), \\ t''_i & = & (t_i),\ \text{if } [\![c]\!]\{(x \leftarrow first(h_1)), (y_l \leftarrow h_2|_{i-1}), (y_r \leftarrow h_2|^i)\} = false, \\ & = & (),\ \text{otherwise} \end{array}$$

Then $list\text{–}trav_{c,\tau,\omega,\rho,\delta}(h_1, h_2)$ is
$concat(h_{output}, list\text{–}trav_{c,\tau,\omega,\rho,\delta}(concat(h_{recurse}, rest(h_1)), h'_2))$.

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.

1. **Singleton Product:** $list\text{–}trav_{sprod}$

   $c:\ \lambda\langle x, y_l, y_r\rangle\ false$

   $\tau:\ \phi$

   $\omega:\ \phi$

   $\rho:\ \phi$

   $\delta:\ \lambda\langle x, z\rangle\ cons(mktup(first(z), x), ())$

   Example: Let $l_1 = (x, y, z)$ and $l_2 = (a)$, then $list\text{–}trav_{sprod}(l_1, l_2) = ([a, x], [a, y], [a, z])$.

2. **Cartesian Product:** $list\text{–}trav_{prod}$

   $c:\ \lambda\langle x, y_l, y_r\rangle\, false$

   $\tau:\ \phi$

   $\omega:\ \phi$

   $\rho:\ \phi$

   $\delta:\ \lambda\langle x, z\rangle\ \ list\text{–}trav_{sprod}(z, cons(x, ()))$

   Example: Let $l_1 = (a, b)$ and $l_2 = (x, y, z)$, then $list\text{–}trav_{prod}(l_1, l_2) =$ $([a, x], [a, y], [a, z], [b, x], [b, y], [b, z])$.

3. **List Reversal:** $list\text{–}trav_{lrev}$

   $c:\ \lambda\langle x, y_l, y_r\rangle\, rest(y_r) = ()$

   $\tau:\ \lambda\langle x, y_l, y_r\rangle\, first(y_r)$

   $\omega:\ \lambda\langle x, y\rangle\, y$

   $\rho:\ \phi$

   $\delta:\ \phi$

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

4. **Membership** $list\text{–}trav_{mem}$

   $c:\ \lambda\langle x, y_l, y_r\rangle\, x = first(y_r)$

   $\tau:\ \phi$

   $\omega:\ \lambda\langle x, y\rangle\, cons(x, ())$

   $\rho:\ \phi$

   $\delta:\ \phi$

   Example: Let $l_1 = (a)$, then $list\text{–}trav_{mem}(l_1, l_2) = (a)$ if $a \in l_2$ and $()$, otherwise.

5. **Projection** $list\text{–}trav_{\pi_i}$

   $c:\ \lambda\langle x, y_l, y_r\rangle\, y_l = ()$

   $\tau:\ \phi$

   $\omega:\ \lambda\langle x, y\rangle\, cons(\pi_i(x), ())$

   $\rho:\ \phi$

   $\delta:\ \phi$

Example: Let $l = ([a,b],[c,d],[c,e])$, then $list\text{–}trav_{\pi_2}(l,l) = (b,d,e)$.

An alternate way to represent $list\text{–}trav_{\pi_i}$ would be to set c to $\lambda\langle x, y_l, y_r\rangle\, true$, τ to $\lambda\langle x, y_l, y_r\rangle\ cons(\pi_i(first(y_r)),())$, ρ to $\lambda\langle x,y\rangle\, y$ and the rest to ϕ.

6. **Append:**
   $append(l_1,l_2) \overset{\text{def}}{=}$
   $if\ (l_1 = ())\ then\ l_2\ else\ list\text{–}trav_{append}(l_1,l_2)$, where $list\text{–}trav_{append}$ is defined as follows:

   $c:\ \lambda\langle x, y_l, y_r\rangle\, y_l = ()$

   $\tau:\ \lambda\langle x, y_l, y_r\rangle\, first(y_r)$

   $\omega:\ \lambda\langle x, y\rangle\, y$

   $\rho:\ \lambda\langle x, y\rangle\, cons(x,())$

   $\delta:\ \lambda\langle x, z\rangle\, cons(x,())$

   Example: $l_1 = (a,b,c)$ and $l_2 = (d,e,f)$ then $list\text{–}trav_{append}(l_1,l_2) = (d,e,f,a,b,c)$.

7. **Flatten:** $list\text{–}trav_{flat}$

   $c:\ \lambda\langle x, y_l, y_r\rangle\, y_l = ()$

   $\tau:\ \phi$

   $\omega:\ \lambda\langle x, y\rangle\, x$

   $\rho:\ \phi$

   $\delta:\ \phi$

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

8. **Length-Comparison:** $list\text{–}trav_{is-longer}$

   $c:\ \lambda\langle x, y_l, y_r\rangle\, y_l = ()$

   $\tau:\ \phi$

   $\omega:\ \phi$

   $\rho:\ \phi$

   $\delta:\ \lambda\langle x, z\rangle\, cons(x,())$

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

9. **Division:** $list\text{–}trav_{div}$

   $c:\ \lambda\langle x, y_l, y_r\rangle\, list\text{–}trav_{is-longer}(x, y_l) \neq ()$

   $\tau:\ \lambda\langle x, y_l, y_r\rangle\, first(y_r)$

   $\omega:\ \lambda\langle x, y\rangle\, if\ y = x\ then\ cons(first(y), ())\ else\ ()$

   $\rho:\ \lambda\langle x, y\rangle\, cons(x, ())$

   $\delta:\ \phi$

   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, $list\text{–}trav_{div}(l_1, l_2)$ returns the result of performing the integer division of n_2 by n_1. For example, if $l_1 = ((1, 1))$ and $l_2 = (1, 1, 1, 1, 1, 1, 1)$, then $list\text{–}trav_{div}(l_1, l_2) = (1, 1, 1)$. However, if the divisor is 0, i.e., $l_1 = (())$, $list\text{–}trav_{div}$ 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 polynomial time generic list-object functions. In Section 4.1, we show some properties of $\mathcal{L}_p$ expressions. Section 4.2 establishes the result that the class of functions represented by $\mathcal{L}_p$ is contained in the class of polynomial time functions. In Section 4.3, we show the converse, i.e., that the class of polynomial time functions is contained in the class of functions represented by $\mathcal{L}_p$. These two results together prove that $\mathcal{L}_p$ captures precisely the class of polynomial time functions. In Section 4.4 we present some modifications to $\mathcal{L}_p$ that extend its computational power beyond PTIME.

4.1 Properties of $\mathcal{L}_p$ expressions

We first establish that the result of a *list–trav*-operation can be determined in a finite amount of time.

Lemma 1 *The result of a list–trav-operation* $list\text{–}trav_{c,\tau,\omega,\rho,\delta}(h_1, h_2)$ *can be determined in a finite amount of time.*

Proof:

We will prove by induction on the length of h_2 that the *list–trav*-operation can be done in a finite number of steps. If the length of h_2 is 0 then we are in the condition-false case. If the length of h_1 is also 0 then we are done. So assume the length of h_1 is positive. The semantics of the *list–trav*-operation then dictates that we compute
$concat([\![\delta]\!]\{(x \leftarrow first(h_1)), (z \leftarrow h_2)\},\ list\text{–}trav_{c,\tau,\omega,\rho,\delta}(rest(h_1), h_2))$.

The crucial observation is that the same *list–trav*-operation is called on $(rest(h_1), h_2)$. So nothing is changed to h_2 and the computation skips over the first element of h_1. Clearly, this patterns of compuation persists until either we run into an undefined situation (and so the computation halts) or until the first argument of the *list–trav*-operation becomes empty. In the latter situation we have entered the first-argument-empty case of the *list–trav*-operation and the computation terminates by returning the empty list to the issueing *list–trav*-operation.

Next, assume that the length of h_2 positive. If the length of h_1 is 0 we are done. Otherwise there are three cases. If we are in the condition-undefined case, the computation terminates. If we are in the condition-true case (and we don't run into an undefined sub-computation in which case the computation halts), the *list–trav*-operation semantics indicates that we compute $list\text{–}trav_{c,\tau,\omega,\rho,\delta}(concat(h_{recurse}, rest(h_1)), h'_2)$. The important observation here is that the length of the second argument h'_2 is strictly smaller than the length of h_2. By the induction hypothesis on this length, we can conclude that the computation terminates after a finite number of steps. The only remaining case is the condition-false case. In that situation, the length of the second argument does not shrink, however the computation skips over the first element of the first argument of the *list–trav*-operation. So, in this case, we need to perform the same analysis as just described. In the worst case, the induction hypothesis can not be invoked because we run into successive condition-false cases. If this happens, however, after a finite number of steps, the first argument of the *list–trav*-operations will become the empty list and so the computation terminates on account of the first-argument-empty case. ■

Next we establish that the functions associated with $\mathcal{L}_p$ expressions are generic, computable list-object functions. Now, strictly speaking, $\mathcal{L}_p$ expressions are functions defined over assignments, whereas list-object functions are defined over database schemas. Nevertheless, the association between $\mathcal{L}_p$ expressions and list-object functions should be clear.

Lemma 2 *If l is an $\mathcal{L}_p$ expression then $[\![l]\!]$ is a generic, computable list-object function.*

Proof: By simple structural induction on the definition of $\mathcal{L}_p$ expressions. ■

So, we have established that $\mathcal{L}_p$ expressions correspond to computable generic list-object functions. Next, we will show that the $\mathcal{L}_p$ expressions actually characterize the polynomial time generic list-object functions.

4.2 $\mathcal{L}_p \subseteq \mathbf{P}$

We begin with some definitions. Let d be an $\mathcal{L}_p$ domain element of type α.

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

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

Let α be a type.

$$\begin{aligned} height(\alpha) &= 1 \text{ if } \alpha = \mathcal{B} \\ &= 1 + max_{1 \leq i \leq k}(height(\alpha_i)) \text{ if } \alpha = [\alpha_1, ..., \alpha_k], \ k \geq 0 \\ &= 1 + height(\alpha') \text{ if } \alpha = (\alpha') \end{aligned}$$

Lemma 3 *Let α be an $\mathcal{L}_p$ type. Then:*

1. *There exist constants $a_\alpha \geq 1$, $b_\alpha \geq 1$ and $c_\alpha \geq 1$ such that for all d in $Dom(\alpha)$*

$$size(d) \leq a_\alpha (maxlength(d))^{b_\alpha} + c_\alpha$$

2. *For all d in $Dom(\alpha)$*

$$maxlength(d) \leq size(d)$$

In other words, size and maxlength are polynomially related measures.

Proof:

1. Set $a_\alpha = 2 * height(\alpha)$, $b_\alpha = height(\alpha)$, and $c_\alpha = 1$.
2. Obvious.

■

4.2.1 Complexity of *list–trav*-free expressions

We recall that an $\mathcal{L}_p$ expression l is called *list–trav*-free if it is generated by the syntax rules in items 1-4 (Section 3.1) only. We first extend the definitions of *size* and *maxlength* to assignments. Formally, if $\sigma = \{(x_1 \leftarrow d_1), ..., (x_n \leftarrow d_n)\}$, then $size(\sigma) = size(d_1) + \cdots + size(d_n)$ and $maxlength(\sigma) = max(n, max_{1 \leq i \leq n} maxlength(d_i))$.

Lemma 4 *Let l be an list–trav-free expression. There exists a polynomial p_l that governs the time complexity of $[\![l]\!]$, i.e., for every valid input d_l to $[\![l]\!]$, $Time([\![l]\!] d_l) \leq p_l(size(d_l))$.*

Proof: The proof of Lemma 4 is by structural induction on *list–trav*-free expressions. ∎

Lemma 5 *Let l be an list–trav-free expression. There exists a constant k_l such that for every valid input d_l to $[\![l]\!]$, if $[\![l]\!]d_l$ is defined then $maxlength([\![l]\!]d_l) \leq maxlength(d_l) + k_l$.*

Proof: The proof is by simple structural induction on the definition of $[\![l]\!]d_l$. The intuition of the proof is that each of the *list–trav*-free subexpressions of l either decreases the *maxlength* measure or adds a constant (independent of d_l) to this measure. ∎

Lemma 6 *Let lt be the list–trav-operation $list\text{–}trav_{c,\tau,\omega,\rho,\delta}$. 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 h_1 and h_2 to lt, the size of the input arguments to subsequent lt operations is bounded from above by $p_{lt}(size(h_1, h_2))$.*

Proof: Since the second argument to subsequent lt calls can only shrink, the only thing we have to consider is the growth-rate of the first argument to lt operations. The first argument can increase due to condition-true cases in the semantics of lt. For convenience, we repeat the semantics of the condition-true case.

Let $\langle x, y_l, y_r\rangle$ be the parameter lists for c and τ, $\langle x, y\rangle$ the parameter lists for ω and ρ and $\langle x, z\rangle$ the list for δ.

Condition-true case:
Both h_1 and h_2 are non-empty lists, $h_2 = (t_1, ..., t_k)$ and for *all* i, $1 \leq i \leq k$,
$[\![c]\!]\{(x \leftarrow first(h_1)), (y_l \leftarrow h_2|_{i-1}), (y_r \leftarrow h_2|^{i})\}$ is defined, and for *some* i, $1 \leq i \leq k$,
$[\![c]\!]\{(x \leftarrow first(h_1)), (y_l \leftarrow h_2|_{i-1}), (y_r \leftarrow h_2|^{i})\} = true$.

Define:

$$\begin{array}{rcl}
h_{output} & = & [\![\omega]\!]\{(x \leftarrow first(h_1)), (y \leftarrow concat(t'_1, ..., t'_k))\}, \\
h_{recurse} & = & [\![\rho]\!]\{(x \leftarrow first(h_1)), (y \leftarrow concat(t'_1, ..., t'_k))\}, \text{where for } i\ (1 \leq i \leq k), \\
t'_i & = & (),\ \text{if } [\![c]\!]\{(x \leftarrow first(h_1)), (y_l \leftarrow h_2|_{i-1}), (y_r \leftarrow h_2|^{i})\} = false, \\
 & = & ([\![\tau]\!]\{(x \leftarrow first(h_1)), (y_l \leftarrow h_2|_{i-1}), (y_r \leftarrow h_2|^{i})\}),\ \text{otherwise} \\
\text{and } h'_2 & = & concat(t''_1, ..., t''_k),\ \text{where for } i\ (1 \leq i \leq k), \\
t''_i & = & (t_i),\ \text{if } [\![c]\!]\{(x \leftarrow first(h_1)), (y_l \leftarrow h_2|_{i-1}), (y_r \leftarrow h_2|^{i})\} = false, \\
 & = & (),\ \text{otherwise}
\end{array}$$

Then $list\text{–}trav_{c,\tau,\omega,\rho,\delta}(h_1, h_2)$ is
$concat(h_{output}, list\text{–}trav_{c,\tau,\omega,\rho,\delta}(concat(h_{recurse}, rest(h_1)), h'_2))$.

By Lemma 5, we have for each i, $1 \leq i \leq k$, $maxlength(t'_i) \leq maxlength(first(h_1), h_2) + k_\tau$.

Since ρ is also a *list–trav*-free expression, we again have by Lemma 5,

$$maxlength(h_{recurse}) \leq maxlength(first(h_1), concat(t'_1, ..., t'_k)) + k_\rho$$

Now, $maxlength(first(h_1), concat(t'_1, ..., t'_k))$

$$\begin{aligned}
&\leq max(maxlength(first(h_1)), maxlength(concat(t'_1, ..., t'_k))) \\
&\leq max(maxlength(first(h_1)), k, maxlength(t'_1, ..., t'_k)) \\
&\qquad \text{(since each } t'_i \text{ is a singleton list by defn.)} \\
&\leq max(maxlength(first(h_1)), maxlength(first(h_1), h_2) + k_\tau) \\
&\qquad \text{(since } k \leq maxlength(h_2)\text{)} \\
&\leq maxlength(first(h_1), h_2) + k_\tau
\end{aligned}$$

Thus, we obtain the following:

Lemma 7

$$maxlength(h_{recurse}) \leq maxlength(first(h_1), h_2) + k_\tau + k_\rho$$

We now establish the following Lemma.

Lemma 8 *Consider lt = list–trav$_{c,\tau,\omega,\rho,\delta}$. There exists a $k \geq 0$ such that if for inputs h_1 and h_2, $lt(h_1, h_2)$ involves $n \geq 0$ condition-true cases then for each $h \neq ()$ which appears as a first argument in a list–trav call resulting from the call $lt(h_1, h_2)$[4], $maxlength(maxel(h)) \leq maxlength(maxel(h_1), h_2) + nk$. Here, maxel is a function which extracts, from a non-empty list, the first element with maximum maxlength measure.*

Proof: We prove Lemma 8 by induction on the number $p \geq 1$ of *list–trav* calls resulting from $lt(h_1, h_2)$.

Basis: $p = 1$. In this case $n = 0$ and the only possibility for h is h_1. If $h_1 \neq ()$, the inequality trivially follows.

Induction: We assume that the lemma holds for *list–trav* calls which result in at most $p-1 \geq 1$ subsequent *list–trav* calls. Assume that $lt(h_1, h_2)$ involves p *list–trav* calls. We consider two cases.

Case 1: The original *list–trav* call, i.e., $lt(h_1, h_2)$ is a condition-*false* case. In this case, the second *list–trav* call is $lt(rest(h_1), h_2)$. By induction we know that for each $h \neq ()$ which appears as a first argument in a *list–trav* call resulting from the call $lt(rest(h_1), h_2)$,

$$maxlength(maxel(h)) \leq maxlength(maxel(rest(h_1)), h_2) + nk$$

Since $maxlength(maxel(rest(h_1))) \leq maxlength(maxel(h_1))$ we have that for such h, the inequality in the lemma holds. The only other value for h is h_1. Clearly, the inequality holds trivially for this value of h.

Case 2: The original *list–trav* call, i.e., $lt(h_1, h_2)$ is a condition-*true* case. In this case, the second lt call is $lt(concat(h_{recurse}, rest(h_1)), h'_2)$, where $h_{recurse}$ and h'_2 are as defined previously.

[4]We include the original call also as such a call.

By induction we know that for each $h \neq ()$ which appears as a first argument in a *list–trav* call resulting from the call $lt(concat(h_{recurse}, rest(h_1)), h'_2)$,

$$maxlength(maxel(h)) \leq maxlength(maxel(concat(h_{recurse}, rest(h_1))), h'_2) + (n-1)k$$

where $k = k_\tau + k_\rho$ (k_τ and k_ρ are the constants appearing in Lemma 7). It follows from Lemma 7 and the semantics of *maxlength* and *maxel* that $maxlength(maxel(concat(h_{recurse}, rest(h_1))), h'_2) \leq maxlength(maxel(h_1), h_2) + k$. Therefore,

$$maxlength(maxel(h)) \leq maxlength(maxel(h_1), h_2) + nk$$

Finally, when $h = h_1$, the inequality holds trivially. ■

Now, let $h_{recurse_i}$ denote the list that is added to the first argument as a result of the ith condition-true case. By Lemma 7, $maxlength(h_{recurse_i}) \leq maxlength(first(h'_1), rest(h_2)) + k$, where h'_1 is the first argument of lt from which $h_{recurse_i}$ is derived and $k = k_\tau + k_\rho$. So, if $lt(h_1, h_2)$ involves n condition-*true* cases,

$$\begin{aligned} maxlength(h_{recurse_i}) \quad & \leq maxlength(maxel(h'_1), rest(h_2)) + k \\ & \leq max(maxlength(maxel(h'_1)), maxlength(rest(h_2))) + k \\ & \leq max(maxlength(maxel(h_1), h_2) + (i-1)k, rest(h_2)) + k \\ & \quad \text{by Lemma 8, since there were } i-1 \text{ condition-true calls} \\ & \quad \text{between the original } h_1 \text{ and } h'_1 \\ & \leq maxlength(maxel(h_1), h_2) + i.k \\ & \leq maxlength(maxel(h_1), h_2) + n.k \end{aligned}$$

Therefore, each condition-*true* case in $lt(h_1, h_2)$ has the potential of adding $maxlength(maxel(h_1), h_2) + nk$ to the maxlength of subsequent *list–trav* calls, where n is the number of condition-*true* cases in $lt(h_1, h_2)$. Since $n \leq length(h_2)$, it follows that

$$maxlength(h) \leq maxlength(h_1) + length(h_2)(maxlength(maxel(h_1), h_2) + length(h_2)k)$$

where $h \neq ()$ is the first argument of any subsequent *list–trav* call. It thus follows that if h and g are first and second arguments of subsequent *list–trav* call to $lt(h_1, h_2)$ that

$$maxlength(h, g) \leq maxlength(h_1, h_2)(1 + (k+1)maxlength(h_1, h_2))$$

If we now invoke Lemma 3 it follows that the *size*-measure of the first argument of *list–trav* is bounded from above by a polynomial in $size(h_1, h_2)$. ■

We are now ready for the main result of this section:

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

Proof: The proof is by induction on the number of occurrences of *list–trav*-expressions in l.

The basis of the induction follows from Lemma 4.

For the induction step, consider the last *list–trav* call, say $list\text{–}trav_{c,\tau,\omega,\rho,\delta}(l_1, l_2)$, of l. By the induction hypothesis, we can assume that there are polynomials p_c, p_τ, p_ω, p_ρ, and p_δ, associated with c, τ, ω, ρ, and δ, respectively.

We now set out to determine the time complexity of $list\text{–}trav_{c,\tau,\omega,\rho,\delta}(l_1, l_2)$. Since l_1 and l_2 are $\mathcal{L}_p$ expressions with fewer *list–trav*-expressions than l, we can assume by the induction hypothesis that there are polynomials p_{l_1} and p_{l_2} that govern the time complexity of l_1 and l_2, respectively.

Given these polynomials, we have to prove that there exists a polynomial $p_{list\text{–}trav_{c,\tau,\omega,\rho,\delta}}$ such that for each valid input pair of list h_1 and h_2 to the *list–trav*-traverse operation $list\text{–}trav_{c,\tau,\omega,\rho,\delta}$

$$Time(list\text{–}trav_{c,\tau,\omega,\rho,\delta}(h_1, h_2)) \leq p_{list\text{–}trav_{c,\tau,\omega,\rho,\delta}}(size(h_1, h_2))$$

We first need to determine an upperbound on the number of times $list\text{–}trav_{c,\tau,\omega,\rho,\delta}$ can be called recursively. Getting this upperbound is principaly determined by the number of times the condition-true and condition-false cases in the semantics of $list\text{–}trav_{c,\tau,\omega,\rho,\delta}$ can be occur. As previously observed (Lemma 6) the condition-true case can occur at most $length(h_2)$ times. The number of occurrences of condition-false case, on the other hand, is determined by the growth-rate of the first argument to $list\text{–}trav_{c,\tau,\omega,\rho,\delta}$. By Lemma 6, we know that this growth-rate is bounded from above by a polynomial in $size(h_1, h_2)$. Hence the number of recursive calls of $list\text{–}trav_{c,\tau,\omega,\rho,\delta}$ is bounded by a polynomial in $size(h_1, h_2)$.

Now each $list\text{–}trav_{c,\tau,\omega,\rho,\delta}$ call involves some extra work. If there is a fixed polynomial in $size(h_1, h_2)$ which governs this work for each such call, we are done.

Each $list\text{–}trav_{c,\tau,\omega,\rho,\delta}$ call involves verifying *condition* on elements and lists generated through the computation from from h_1 and h_2. By Lemma 6 we know that these elements and sublists are polynomially bounded in $size(h_1, h_2)$. By the induction hypothesis, there is a polynomial p_c that governs the time complexity of condition c, i.e., each c condition check is polynomialy time-bounded in $size(h_1, h_2)$.

In the condition-false case, the extra work is output related and is governed by the polynomial p_δ associated with δ and the-growth rate of the first argument to $list\text{–}trav_{c,\tau,\omega,\rho,\delta}$ calls. So this work is also polynomially bounded in $size(h_1, h_2)$.

Finally, in the condition-true case, the extra work is determined by the polynomials p_τ, p_ρ, p_ω and p_δ associated with τ, ρ, ω and δ, repectively, and the growth-rate of the first argument to $list\text{–}trav_{c,\tau,\omega,\rho,\delta}$ calls. So again this work is polynomially bounded in $size(h_1, h_2)$.

Therefore, we have established that each $list\text{–}trav_{c,\tau,\omega,\rho,\delta}$ call involves work which is governed by a (fixed) polynomial in $size(h_1, h_2)$. Since there are at most a (fixed) polynomial number of recursive $list\text{–}trav_{c,\tau,\omega,\rho,\delta}$ calls, again measured in $size(h_1, h_2)$, the overall complexity of $list\text{–}trav_{c,\tau,\omega,\rho,\delta}(h_1, h_2)$ is bounded from above by a polynomial in $size(h_1, h_2)$, as was to be established.

Now returning to the original expression l, we might have some further $\mathcal{L}_p$ expressions after the last *list–trav* call. Since these expressions are necessarily *list–trav*-free they entail further polynomial work in $size(h_1, h_2)$ (Lemma 4).

Since h_1 and h_2 were taken to be arbitrary input lists to $list\text{–}trav_{c,\tau,\omega,\rho,\delta}$ (the last *list–trav*-call in l) we can conclude by the induction hypothesis that $[\![l]\!]$ is polynomially bounded.

■

4.3 $\mathcal{L}_p \supseteq \mathbf{P}$

Let f be a polynomial time generic list-object function. We will construct a $\mathcal{L}_p$ expression l_f such that $[\![l_f]\!] = f$.

Let μ be a fixed encoding of $\mathcal{U}$ and let M_f be a polynomial TM that computes f (see Section 2.2).

A central difficulty in this construction is the mismatch that exists between the objects manipulated by TM's, i.e., words over the **finite** alphabet $\{\mathbf{0},\ \mathbf{1},\ \boldsymbol{(},\ \boldsymbol{)},\ [\![,\ ]\!],\ |,\ \mathbf{\#}\}$ and objects manipulated by $\mathcal{L}_p$ expressions. Since $\mathcal{L}_p$ expressions can involve the (basic) type $\mathcal{B}$, and therefore also its associated **infinite** 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 function, it will suffice to map $\mathcal{L}_p$ objects into their corresponding *canonical representations*, which *can* be input to an $\mathcal{L}_p$ expression $M_f(\mathcal{L}_p)$ that emulates M_f. The canonical representation mapping has the property that if d_1 and d_2 are isomorphic $\mathcal{L}_p$ objects (see Section 2.3) then their canonical representations are identical. The genericity of f guarantees that we can effectively compute the behavior of f on d by first applying $M_f(\mathcal{L}_p)$ to d's canonical representation. If, in addition, we maintain a *dictionary* which allows us to map back and forth between d and its canonical representation, and if, at the end of the $M_f(d)$ emulation, we transform back according to this dictionary, genericity guarantees that we get the desired result.

We begin by showing how an arbitrary $\mathcal{L}_p$ object can be "TM-represented" by a domain element of type $((\mathcal{B}))$. Consider the finite alphabet $\{\mathbf{0},\ \mathbf{1},\ \boldsymbol{(},\ \boldsymbol{)},\ [\![,\ ]\!],\ |,\ \mathbf{\#}\}$. We will code these alphabet symbols by lists of type $((\mathcal{B}))$ as follows:

symbol	code
0	(())
1	((),())
(	((),(),())
)	((),(),(),())
⟦	((),(),(),(),())
⟧	((),(),(),(),(),())
‖	((),(),(),(),(),(),())
#	((),(),(),(),(),(),(),())

Now a word over $\{\mathbf{0}, \mathbf{1}\}$ is coded by a list of type $((\mathcal{B}))$. For example the word **1001** is coded by the list $(((), ()), (()), (()), ((), ()))$.

And, a word over the full alphabet is coded correspondingly. For example the word

(⟦ 1 0 ‖ 0 ⟧ ‖ ⟦ 0 ‖ 1 1 1 ⟧)

is coded by the list (**(**, **⟦**, **1**, **0**, ‖, **0**, **⟧**, ‖, **⟦**, **0**, ‖, **1**, **1**, **1**, **⟧**, **)**)

For simplicity we have used the original symbols to represent their corresponding codes.

Remark: Given a type α, we can construct an $\mathcal{L}_p$ expression which decides, upon input of a domain element e of type $((\mathcal{B}))$, whether e corresponds to a domain element of α.

Canonical representations

Next we will establish how an input d to f can be transformed by an $\mathcal{L}_p$ expression into a *canonical* representation. The canonical representation mapping is such that if d_1 and d_2 are isomorphic inputs to f, then their canonical representations are identical.

We first give an outline of the general algorithm for encoding an $\mathcal{L}_p$ domain element into a flat sequence of atomic (basic domain) elements that can be input to a TM. We then give a sketch of the algorithm for decoding the output of a TM into the equivalent $\mathcal{L}_p$ element. Note that since the encoding/decoding procedure depends on the type of the $\mathcal{L}_p$ domain element, the exact algorithm for encoding (decoding) will vary depending upon the type of the input. For the sake of simplicity we assume that the Turing Machine symbols are coded into themselves rather than into their representation into objects of type $((\mathcal{B}))$.

Encoding

The encoding procedure is a three step process. The first step creates a dictionary representing a mapping between the atomic elements occurring in the input list and lists representing numbers. Numbers are encoded in a unary representation. For example, the number 4 is represented by the list (**1**,**1**,**1**,**1**). Encodings in other representations, e.g. binary, are also possible; however, for the sake of simplicity, we have chosen the unary representation for this discussion. Note that the binary representation of a number can be constructed from the unary representation by using the remainders obtained on repeated divisions by 2 (see the integer division example in Section 3). The second step creates a flattened version of the input list with the appropriate list and tuple delimiters. The final step replaces each atomic element in the flattened list with its unary or binary representation. We will refer to these steps as Step 1, Step 2 and Step 3.

As an example, consider the list $([a,b],[c,a],[b,d])$. The set of all the atomic elements in this list is $\{a,b,c,d\}$. We can now assign a bijective mapping from this set to the set of numbers $\{1,2,3,4\}$. One such mapping may be represented by the list $([a,(\mathbf{1})],[b,(\mathbf{1},\mathbf{1})],[c,(\mathbf{1},\mathbf{1},\mathbf{1})],$ $[d,(\mathbf{1},\mathbf{1},\mathbf{1},\mathbf{1})])$, where the second component of each tuple is the unary representation of a number between 1 and 4. The first step in the encoding process will generate such a mapping.

The second step will generate a flattened version of the input list with the appropriate list and tuple delimiters. So, the flattened version of the example list will be

$$(\llparenthesis,\llbracket,a,\|,b,\rrbracket,|,\llbracket,c,\|,a,\rrbracket,|,\llbracket,b,\|,d,\rrbracket,\rrparenthesis)$$

In the final step, each of the atomic elements in the above list (excluding the special symbols - the delimiters, etc.,) will be replaced by the corresponding number from the mapping list generated in Step 1. So, the final encoded list of the example list will be

$$(\llparenthesis,\llbracket,\mathbf{1},\|,\mathbf{1},\mathbf{1},\rrbracket,|,\llbracket,\mathbf{1},\mathbf{1},\mathbf{1},\|,\mathbf{1},\rrbracket,|,\llbracket,\mathbf{1},\mathbf{1},\|,\mathbf{1},\mathbf{1},\mathbf{1},\mathbf{1},\rrbracket,\rrparenthesis)$$

Step 1

The input list is flattened and all the duplicates are removed to get the list of all atomic elements in the input list. This can be done by applying the appropriate "flatten" at each step. For example, if the input list, say l, is of type (α) where α is the type $[\alpha_1,\alpha_2]$, then the list of atomic elements, say s, can be obtained as follows:

$s = \mathit{list\text{–}trav}_{rdup}(l',l')$ where $l' = \mathit{list\text{–}trav}_{flat_{(\alpha)}}(l,l)$ and $\mathit{list\text{–}trav}_{flat_{(\alpha)}}$ is defined as follows:

$c : \lambda\langle x, y_l, y_r\rangle\, y_l = ()$
$\tau : \phi$
$\omega : \lambda\langle x, y\rangle\, \mathit{flat}_\alpha(x)$
$\rho : \phi$
$\delta : \phi$

where $\mathit{flat}_\alpha(x) = \mathit{concat}(\mathit{flat}_{\alpha_1}(\pi_1(x)), \mathit{flat}_{\alpha_2}(\pi_2(x)))$ and flat_{α_1} and flat_{α_2} are defined in a similar fashion. So, if $l = ([a,b],[c,a],[b,d])$ then the final list containing the atomic elements in l will be (a,b,c,d).

The *concat* operation can be defined as follows: $\mathit{concat}(l_1,l_2) = \mathit{list\text{–}trav}_{append}(l_2,l_1)$ (the operation $\mathit{list\text{–}trav}_{append}$ is defined in Section 3.3).

Next, we construct a list containing tuples of the form $[x, y]$ where x is an atomic element and y is a list containing the corresponding unary representation. Let l_1 be the list $(())$ and l_2 a list containing n number of $\mathbf{1}$'s, where n is the number of atomic elements in the list constructed earlier (the list l_1 can be constructed easily from the list of atomic elements). For example, if the list of atomic elements is (a, b, c, d), then $l_2 = (\mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{1})$. Then, $list\text{–}trav_{number}(l_1, l_2)$ will contain a list of lists containing the unary representation of the numbers 1 through n, where $list\text{–}trav_{number}$ is as follows:

$c : \lambda\langle x, y_l, y_r\rangle\, true$
$\tau : \lambda\langle x, y_l, y_r\rangle\, cons(\mathbf{1}, y_l)$
$\omega : \lambda\langle x, y\rangle\, y$
$\rho : \phi$
$\delta : \phi$

For example, $list\text{–}trav_{number}((()), (\mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{1}))$ will produce the list $((\mathbf{1}), (\mathbf{1}, \mathbf{1}), (\mathbf{1}, \mathbf{1}, \mathbf{1}), (\mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{1}))$.

The list containing tuples as described above can be easily constructed from the list of atomic elements and the list of numbers generated above. So, if (a, b, c, d) is the list of atomic elements, then the final result of Step 1 will be the list $([a, (\mathbf{1})], [b, (\mathbf{1}, \mathbf{1})], [c, (\mathbf{1}, \mathbf{1}, \mathbf{1})], [d, (\mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{1})])$.

<u>Step 2</u>

Let l' be a list of type $((\mathcal{B}))$ containing n empty lists, where n is the size of the original input list l. The list l' can be constructed easily from l. Then, the transformed list l'' is obtained as follows:

$l'' = cons(\boldsymbol{(}, list\text{–}trav_{trans_{(\alpha)}}(l, l'))$, where $list\text{–}trav_{trans_{(\alpha)}}$ is defined as follows:

$c : \lambda\langle x, y_l, y_r\rangle\, y_l = ()$
$\tau : \lambda\langle x, y_l, y_r\rangle\ \ if\ rest(y_r) \neq ()\ then\ \|\ else\ \boldsymbol{)}$
$\omega : \lambda\langle x, y\rangle\, concat(trans_\alpha(x), y)$
$\rho : \phi$
$\delta : \phi$

where $trans_\alpha(x) = concat((\llbracket\,), concat(trans_{\alpha_1}(\pi_1(x)), concat((\|), concat(trans_{\alpha_2}(\pi_2(x)), (\rrbracket)))))$ and $trans_{\alpha_1}$ and $trans_{\alpha_2}$ are defined analogously.

Thus, if $l = ([a, b], [c, a], [b, d])$, then the transformed list will be $(\boldsymbol{(}, \llbracket, a, \|, b, \rrbracket, \|, \llbracket, c, \|, a, \rrbracket, \|, \llbracket, b, \|, d, \rrbracket, \|, \boldsymbol{)})$

<u>Step 3</u>

The last step involves replacing each atomic element in the list produced in Step 2 by the unary representation. Let $list\text{–}trav_{lookup}$ be defined as follows:

$c : \lambda\langle x, y_l, y_r\rangle\, \pi_1(first(y_r)) = x$
$\tau : \lambda\langle x, y_l, y_r\rangle\, \pi_2(first(y_r))$

$\omega : \lambda\langle x, y\rangle\, first(y)$
$\rho : \phi$
$\delta : \phi$

Then, $list\text{–}trav_{lookup}(x, s)$, where x is an atomic element and s is the mapping list produced in Step 1, will return the unary number that x is mapped to.

Let $list\text{–}trav_{substitute}$ be defined as follows:

$c : \lambda\langle x, y_l, y_r\rangle\, false$
$\tau : \phi$
$\omega : \phi$
$\rho : \phi$
$\delta : \lambda\langle x, z\rangle\ \ if\ list\text{–}trav_{mem}(x, l') = ()\ then\ list\text{–}trav_{lookup}(cons(x, ()), z)\ else\ cons(x, ())$

Decoding

The decoding process is essentially a two step process where the first step involves replacing the numbers with the atomic elements that they represent (the reverse of Step 3 in the encoding process) and then recreating the complex object represented by the list (the reverse of Step 2 in the encoding process).

Step 1

In order to replace the numbers with the corresponding atomic elements, we will need a lookup function, similar to the one given for the encoding process, that takes a list representing a unary number and the mapping list and returns the corresponding element. But first, we must group together all the **1**'s representing a number. Let $list\text{–}trav_{regroup}$ be defined as follows:

```
$c : \lambda\langle x, y_l, y_r\rangle\, y_l = ()$
$\tau : \lambda\langle x, y_l, y_r\rangle\, first(y_r)$
$\omega : \lambda\langle x, y\rangle\ \ if\ \ first(y) = \mathbf{1}\ then\ ()\ else$
          $if\ x = ()\ then\ y\ else\ cons(x, cons(y, ()))$
$\rho : \lambda\langle x, y\rangle\ \ if\ \ first(y) = \mathbf{1}\ then\ cons(cons(first(y), x), ())\ else\ cons((), ())$
$\delta : \phi$
```

$list\text{–}trav_{regroup}((()), l')$ will produce a list where every element in l' that is not a **1**, is replaced by a singleton list containing that element and each sequence of **1**'s is replaced by a list containing the **1**'s. Let this list be denoted as l''. We can now apply $list\text{–}trav_{decode}(l'', s)$, where s is the mapping list and $list\text{–}trav_{decode}$ is defined as follows:

$c : \lambda\langle x, y_l, y_r\rangle\, false$
$\tau : \phi$
$\omega : \phi$
$\rho : \phi$

$\delta : \lambda\langle x, z\rangle \;\; if \;\; first(x) = \mathbf{1} \; then \; list\text{–}trav_{revlookup}(cons(x, ()), z) \;\; else \; x$

where $list\text{–}trav_{revlookup}$ takes a list containing (the unary representation) of a number and the mapping list and returns a list containing the corresponding atomic element.

Step 2

Next, the list produced in Step 1, say h, has to be transformed into the structured complex object that it represents. Like the encoding process, the exact transformation operations that are applied in this step will depend upon the type of the complex object that the list represents. For example, if the type of the output of the function (being computed by the TM) is $\mathcal{B}$, then the corresponding list generated by Step 1 of the decoding process, i.e., h will be of the form (a), where $a \in \mathcal{B}$ and the corresponding transformation is simply $first(h)$. If the output represents a $\mathcal{L}_p$ expression of type $(\mathcal{B})$, i.e., h is of the form $(\pmb{(}, a_1, \|, ..., \|, a_n, \pmb{)})$. We can perform $list\text{–}trav_{unflat_1}(l, h)$ to get the corresponding $\mathcal{L}_p$ expression $(a_1, ..., a_n)$ where l is any non-empty list and $list\text{–}trav_{unflat_1}$ is defined below.

$c : \lambda\langle x, y_l, y_r\rangle \; first(y_r) \neq \pmb{(} \wedge first(y_r) \neq \pmb{)} \wedge first(y_r) \neq \|$
$\tau : \lambda\langle x, y_l, y_r\rangle \; first(y_r)$
$\omega : \lambda\langle x, y\rangle \; y$
$\rho : \phi$
$\delta : \phi$

Now, suppose that the output is of type $((\mathcal{B}))$. For example, if the output represents the list $((a, b), (c))$, then h will be $(\pmb{(}, \pmb{(}, a, |, b, \pmb{)}, \|, \pmb{(}, c, \pmb{)}, \pmb{)})$.

Let $list\text{–}trav_{balanced}$ be defined as follows:

$c : \lambda\langle x, y_l, y_r\rangle \; y_l = () \wedge (((first(y_l) \neq \pmb{(}) \vee (first(x) = \pmb{)})) \wedge ((first(y_l) \neq \pmb{[\![}) \vee (first(x) = \pmb{]\!]})))$
$\tau : \lambda\langle x, y_l, y_r\rangle \; first(y_r)$
$\omega : \phi$
$\rho : \lambda\langle x, y\rangle \;\; if \;\; first(y) = \pmb{(} \vee first(y) = \pmb{[\![} \;\; then \; cons(rest(x), ())$
$\qquad\quad else \; if \;\; first(y) = \pmb{)} \vee first(y) = \pmb{]\!]} \;\; then \; cons(cons(first(y), x), ()) \; else \; cons(x, ())$
$\delta : \lambda\langle x, z\rangle \;\; if \;\; z = () \wedge x = () \; then \; (()) \; else \; ()$.

So, if $l_1 = (())$ and l_2 is a list of symbols, then $list\text{–}trav_{balanced}(l_1, l_2)$ will return true, i.e., $(())$, if l_2 is the reverse of the list representing an encoding of a $\mathcal{L}_p$ object and false otherwise. For example, if $l_2 = (\pmb{]\!]}, b, |, a, \pmb{[\![})$, then the reverse of l_2 represents the object $[a, b]$ and so $list\text{–}trav_{balanced}$ will return true.

Let h' be the list h without the two outermost parentheses. This can be easily obtained by applying the $first$ and $list\text{–}trav_{lrev}$ (list reverse) operators. We can now, apply $list\text{–}trav_{unflat_2}(l, h')$, where l is a list containing as may empty lists as the number of elements in h' and $list\text{–}trav_{unflat_2}$ is defined below.

$c : \lambda\langle x, y_l, y_r\rangle\, y_l = () \wedge ((x = ()) \vee (list\text{–}trav_{balanced}((()), x) = ()))$ (i.e., x is not balanced)
$\tau : \lambda\langle x, y_l, y_r\rangle\, first(y_r)$
$\omega : \phi$
$\rho : \lambda\langle x, y\rangle\ \ if\ \ first(y) = \mathbf{|} \wedge x = ()\ \ then\ ()\ else\ cons(cons(first(y), x), ())$
$\delta : \lambda\langle x, z\rangle\ \ cons(list\text{–}trav_{lrev}(x, x), ())$

So, if h' were the list $(\mathbf{(}, a, \mathbf{|}, b, \mathbf{)}, \mathbf{|}, \mathbf{(}, c, \mathbf{)})$, then the result of $list\text{–}trav_{unflat_2}((()), h')$ will be the list $((\mathbf{(}, a, \mathbf{|}, b, \mathbf{)}), (\mathbf{(}, c, \mathbf{)}))$. We can now apply a traversal operation that takes this list and applies $list\text{–}trav_{unflat_1}$ to each element of the list.

We are now ready to state and prove the main results of this section.

TM emulation

Our first result is that, with respect to the just described representation method, there exists an $\mathcal{L}_p$ expression $M_f(\mathcal{L}_p)$ which emulates M_f (recall that M_f is a TM which computes f).

Theorem 1 *Let M_f be a polynomial time TM computing the list function f in the context of an encoding function μ. Then, (i) there exists an $\mathcal{L}_p$ expression $M_f(\mathcal{L}_p)$ of type $((\mathcal{B})) \to ((\mathcal{B}))$ which emulates M_f, and (ii) the time complexity associated with $M_f(\mathcal{L}_p)$ is polynomially related to that of M_f.*

Proof: For simplicity, we will denote the codes of the TM alphabet symbols by the alphabet symbols themselves.

For the simulation of M_f in $\mathcal{L}_p$, we first describe the singleton list l_1 which represents a configuration of M_f. If l_1 contains $[s, a, (a_1, a_2, ..., a_n), (a_{-1}, a_{-2}, ..., a_{-m})]$ then 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.

We will now construct the $\mathcal{L}_p$ expression $M_f(\mathcal{L}_p)$ that emulates M_f. For convenience, we will denote the expression as a sequence of assignment statements, instead of a series of compositions.

We can first determine the size of the input (it is straightforward to construct an $\mathcal{L}_p$ expression to do this). Let n be this size and let p be the polynomial $an^k + b$ (for some natural number constants a, b, and k) which governs the time complexity of M_f. Given constants a, b, and k, is easy to write an $\mathcal{L}_p$ expression which, given a list of n (n arbitary) elements, produces a list of $an^k + b$ elements. We can define a function $list\text{–}trav_{mult}$ that takes two lists of m and n elements and constructs a list containing $m * n$ elements (see the cartesian product example in Section 3). Let j represent a list containing n elements. By repeated applications of $list\text{–}trav_{mult}$ one can construct a list containing n^k elements.

$x_1 = list\text{–}trav_{mult}(j, j)$
$x_2 = list\text{–}trav_{mult}(x_1, j)$
$\vdots$
$x_{k-1} = list\text{–}trav_{mult}(x_{k-2}, j)$

The list x_{k-1} contains n^k number of symbols. We can generate a list containing an^k symbols by applying $list\text{–}trav_{append}$ repeatedly (a times). Finally, we can take some symbol such as **0** and *cons* it repeatedly (b times) to obtain a list containing the appropriate polynomial number of symbols.

Now, let $list\text{–}trav_{M_f}(l_1, l_2)$ be as follows:

```
$c :\ \lambda\langle x, y_l, y_r\rangle\, y_l = () \vee \pi_1(x) \neq q_h$

$\tau :\ \lambda\langle x, y_l, y_r\rangle\ if\ \pi_1(x) = s_1 \wedge \pi_2(x) = a_1$
        $then\ \delta_{s_1,a_1}$
        $else\ if\ \pi_1(x) = s_2 \wedge \pi_2(x) = a_1$
        $then\ \delta_{s_2,a_1}$
        $else\ ...$
        $\vdots$
        $then\ \delta_{s_m,a_n}$

$\omega\ :\ \phi$

$\rho\ :\ \lambda\langle x, y\rangle\, y$

$\delta\ :\ \lambda\langle x, z\rangle\, x$
```

where, $Q = \{s_1, ..., s_m\}$, $S = \{a_1, ..., a_n\}$ and for each pair $s \in Q$ and $a \in S$, if $f(s, a) = s'$, $g(s, a) = a'$ and $d(s, a) = R$, then $\delta_{s,a}$ is as follows:

```
$if\ \pi_3(x) \neq ()$
then $mktup(f_1(x), f_2(x), f_3(x), f_4(x))$
    where $f_1(x) = s'$,
          $f_2(x) = first(\pi_3(x))$,
          $f_3(x) = rest(\pi_3(x))$
      and $f_4(x) = cons(a', \pi_4(x))$.
else $mktup(f_1(x), f_2(x), f_3(x), f_4(x))$
    where $f_1(x) = s'$,
          $f_2(x) = \texttt{#}$,
          $f_3(x) = ()$
      and $f_4(x) = cons(a', \pi_4(x))$.
```

For $d(s,a) = L$, we change $f_2(x) = first(\pi_4(x))$, $f_3(x) = cons(a', \pi_3(x))$ and $f_4(x) = rest(\pi_4(x))$ and the condition $(\pi_3(x) \neq ())$ by $(\pi_4(x) \neq ())$.

Now, $l_M = list\text{–}trav_M(l_1, l_2)$, where l_1 is the initial input and l_2 is the list containing the required polynomial number of elements. It is obvious that for each step of M_f, this program maintains a proper encoding of the configuration in l_1 and decrements the length of l_2 by 1. As long as M_f halts in $p(n)$ steps, the program has enough symbols in l_2 to maintain the emulation and it will stop whenever M halts.

δ causes the contents of l_1 to be output. We can examine the first field to determine if M_f reached a final state. If so, the third field represents the output of M_f. It is straightforward to write the $\mathcal{L}_p$ expression required to perform this last step.

We will now prove the second part of Theorem 1 which states that the time complexity of $M_f(\mathcal{L}_p)$ and M_f are polynomially related.

If d is an $\mathcal{L}_p$ domain element, then let $struct\text{–}flat(d)$ denote the flattened structure with the appropriate delimiters. For example, if $d = ([a,b])$ then $struct\text{–}flat(d) = (\boldsymbol{(}, [\![, a, \|, b,]\!], \boldsymbol{(})$. Now, the encoding process described earlier transforms an input d into $c(\mu^*(\psi(d)))$ where c is the mapping of the TM alphabet into the lists of type $((\mathcal{B}))$ and the mapping ψ is defined as follows. First we compute in $\mathcal{L}_p$ the flat list $flat(d) = (a_0, ..., a_{n-1})$, $(n \geq 0)$, which lists all the distinct elements of $\mathcal{U}$ that occur in d. Given μ, let $(a^i_0, ..., a^i_{n-1})$ be the elements in $\mathcal{U}$ such that $\mu(a^i_j) = j$, $0 \leq j \leq n-1$. The mapping ψ is such that $\psi(a_j) = a^i_j$, $0 \leq j \leq n-1$. Let $encode_f$ denote the $\mathcal{L}_p$ expression that performs the encoding $c(\mu^*(\psi(d)))$ on an input d.

For any given polynomial-time generic list-object function f, we know that

$$Time(M_f(\mu^*(d))) \leq p_{M_f}(\mid \mu^*(d) \mid)$$

and

$$Time(M_f(\mathcal{L}_p)(encode_f(d))) \leq p_{M_f(\mathcal{L}_p)}(size(encode_f(d)))$$

where p_{M_f} and $p_{M_f(\mathcal{L}_p)}$ are some polynomials.

Now, since $\mid \mu^*(\psi(d)) \mid \leq \mid \mu^*(d) \mid$ and $size(c(e)) \leq k \mid e \mid$ where e is a flat list of TM alphabet symbols and k is some constant, we have

$$size(encode_f(d)) \leq k \mid \mu^*(\psi(d)) \mid \leq k \mid \mu^*(d) \mid$$

Therefore,

$$p_{M_f(\mathcal{L}_p)}(size(encode_f(d))) \leq p_{M_f(\mathcal{L}_p)}(k \mid \mu^*(d) \mid)$$

Hence $Time(M_f(\mu^*(d)))$ and $Time(M_f(\mathcal{L}_p)(encode_f(d)))$ are polynomially related. ■

Proposition 2 *Let f be a polynomial time generic list-object function. There exists a $\mathcal{L}_p$ expression l_f such that $[\![l_f]\!] = f$.*

Proof: In this construction, let μ be a fixed encoding of $\mathcal{U}$. Since f is a polynomial time generic list 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 $M_f(\mu^*(d)) = \mu^*(f(d))$.

Now let $M_f(\mathcal{L}_p)$ be the $\mathcal{L}_p$ expression which simulates M_f as established in Theorem 1. Now

- $encode_f$ is an $\mathcal{L}_p$ expression which transforms d into the (flat) list $c(\mu(\psi(d)))$ as explained earlier.
- We know that $M_f(\mathcal{L}_p)(encode_f(d)) = M_f(\mathcal{L}_p)(c(\mu(\psi(d)))) = c(\mu(f(\psi(d))))$ by the properties of $M_f(\mathcal{L}_p)$. Because f is generic, it follows that $M_f(\mathcal{L}_p)(encode_f(d)) = c(\mu(\psi(f(d))))$.
- Now $decode_f$ transforms back $c(\mu(\psi(d)))$ according to the mapping $\psi^{-1}\mu^{-1}c^{-1}$. Therefore
$$decode_f(M_f(\mathcal{L}_p)(encode_f(d))) = decode_f(c(\mu(\psi(f(d))))) = f(d)$$

Therefore, if we choose for l_f the $\mathcal{L}_p$ expression $decode_f(M_f(\mathcal{L}_p)(encode_f(.)))$ we have established the proposition. ■

4.4 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, $list\text{–}trav_{exp}(l_1, l_2)$ will produce a list containing 2^n elements, where $list\text{–}trav_{exp}$ is defined as follows:

$c : \lambda\langle x, y_l, y_r\rangle\, (y_l = ())$
$\tau : \phi$
$\omega : \phi$
$\rho : \lambda\langle x, y\rangle\, cons(list\text{–}trav_{append}(x, x), ())$
$\delta : \lambda\langle x, z\rangle\, x$

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 the $list\text{–}trav_{exp}$ function as follows:

$c : \lambda\langle x, y_l, y_r\rangle\, (y_l = ())$
$\tau : \phi$
$\omega : \phi$
$\rho : \lambda\langle x, y\rangle\, cons(cons(x, x), ())$
$\delta : \lambda\langle x, z\rangle\, x$

5 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.

Acknowledgements

We are very grateful to Ed Robertson and Jan Van den Bussche for insightful remarks. We also thank Gerry Allwein for helpful comments.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

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

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

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

[9] 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, Washingtion, DC, May 1993.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

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