Query Languages for Sequence Databases: Termination and Complexity*

Giansalvatore Mecca
Università della Basilicata (Italy)
mecca@dia.uniroma3.it

Anthony J. Bonner
University of Toronto (Canada)
bonner@cs.toronto.edu

Abstract

This paper develops a query language for sequence databases, such as genome databases and text databases. Unlike relational data, queries over sequential data can easily produce infinite answer sets, since the universe of sequences is infinite, even for a finite alphabet. The challenge is to develop query languages that are both highly expressive and finite. This paper develops such a language as a subset of a logic for string databases called *Sequence Datalog*. The main idea is to use safe recursion to control and limit unsafe recursion. The main results are the definition of a finite form of recursion, called *domain–bounded recursion*, and a characterization of its complexity and expressive power. Although finite, the resulting class of programs is highly expressive, since its data complexity is complete for the elementary functions.

1 Introduction

It is widely accepted that relational databases do not provide enough support for many of today's advanced applications. In some cases, object-oriented databases are the right solution. However, in other cases, such as genome databases and text databases, there is still a need for more flexibility in data representation and manipulation. The problem of extending relational databases with string manipulation features has recently motivated several research proposals [4, 8, 5, 6, 7]. In fact, sequences represent a particularly interesting domain for query languages. In contrast to sets, computations over sequences can easily become infinite, even when the underlying alphabet is finite. This is because repetitions of symbols are allowed, so that the number of possible sequences over any finite alphabet is infinite. The researcher thus faces an interesting challenge: on the one hand, the language should provide powerful primitives for restructuring sequences; on the other hand, the expressive power of the language should be carefully limited, to avoid infinite computations.

In [3], we developed a logic called *Sequence Datalog* for querying sequence databases. A safe subset of the logic was defined, based on a new computational model called *Generalized Sequence Transducers*. These machines are a simple yet powerful device for computing sequence mappings. In [3], we showed how networks of these machines could be expressed in Sequence Datalog. Moreover, any Sequence Datalog program constructed in this way is guaranteed to be safe and finite. In this paper, we take a different approach: instead of computational definitions, we develop *syntactic restrictions* that guarantee finiteness and safety. This provides an alternate view of finite computations in the logic. The main idea is to use structural recursion (which is guaranteed to terminate) to limit the construction of new sequences. The

*Preliminary portions of this paper appear in the *Proceedings of the Fifth International Workshop on Database Programming Languages (DBPL'95).* The first author was partially supported by MURST and Consiglio Nazionale delle Ricerche (CNR). The second author was partially supported by an operating grant from the Natural Sciences and Engineering Research Council of Canada (NSERC).

first result is a syntactically defined class of Sequence Datalog programs that guarantees finiteness and safety. We call these programs *domain–bounded programs*. The second result is a characterization of their complexity and expressive power. We prove that domain–bounded programs can express any sequence mapping with hyper-exponential time complexity. Thus, although finite, these programs are still highly expressive.

In the paper, we use bounded term-size to study complexity. Bounded terms have been extensively studied in the framework of logic programming (see, for example, [11]). Our work differs from earlier works in this field in several respects. First, we study complexity (upper and lower bounds), not just termination. Second, we derive results on expressive completeness. Third, we are concerned with *bottom-up* execution (in the deductive-database tradition), not *top-down* execution (in the logic-programming tradition). This difference is significant, since programs that terminate in a top-down execution model, may not terminate in a bottom-up model; and vice-versa. For instance, one of our results shows that the least fixpoint of a program is finite, which is not at all the same thing as saying that top-down execution terminates. Fourth, the syntax and semantics of our language is different from that of classical logic programs. Semantically, all function terms in Sequence Datalog are *interpreted* as sequences. Thus, a program in our language can build sequences, but cannot build more complex structures, such as nested lists. In addition, Sequence Datalog has a richer syntax than the $[Head|Tail]$ list constructor of Prolog. We show that this syntax leads naturally to methods for reasoning about sequence length, and to syntactic restrictions for limiting program complexity.

Our results rely on the ability of Sequence Datalog to distinguish syntactically between two types of recursion — safe and unsafe. In bottom-up execution, recursion through construction of new sequences is inherently unsafe since it can create longer sequences, which can make the active domain grow indefinitely. On the other hand, structural recursion over existing sequences is inherently safe, since it only creates shorter sequences, so that growth in the active domain is bounded. In Sequence Datalog, constructive recursion is performed using constructive terms, of the form $X \bullet Y$, and structural recursion is performed using indexed terms, of the form $X[n_1{:}n_2]$, as shown in the following sections. In this framework, bottom-up evaluation is based on a novel active-domain semantics (which introduces the notion of *extended active domain* of a sequences database). This semantics is crucial to our finiteness and complexity results.

The paper is organized as follows. We first give an overview of the syntax and semantics of Sequence Datalog, in Section 2. Then, in Section 3 we informally introduce the notion of *domain–bounded recursion* with the help of several examples. The rest of the paper is devoted to the formal definition of this form of recursion, and to the study of its complexity and expressibility. Sections 4.1 and 4.2 develop a simple form of reasoning on sequence terms that allows us to "compare" their lengths. These ideas are then used in Section 4.3 as a basis for defining the notion of domain–bounded program. Finally, complexity and expressibility results are given in Section 5.

2 Overview of Sequence Datalog

Sequence Datalog is an extension of Datalog for manipulating sequences. It uses a simple data model that extends the relational model by allowing tuples of *sequences* in relations, instead of just tuples of constant symbols. Let Σ be a countable set of symbols, called the *alphabet*. Σ^* denotes the set of all possible sequences over Σ, including the *empty sequence*, ϵ. $\sigma_1\sigma_2$ denotes the concatenation of two sequences, $\sigma_1, \sigma_2 \in \Sigma^*$. $\text{LEN}(\sigma)$ denotes the length of sequence σ, and $\sigma(i)$ denotes its i-th element. With an abuse of notation, we blur the distinction between elements of the alphabet and 1-ary sequences. A *relation* of arity k over Σ is a finite subset of the k-fold cartesian product of Σ^* with itself. A *database* over Σ is a finite set of relations over Σ. We assign a distinct predicate symbol, r, of appropriate arity to each relation in a database.

This section provides an informal overview of the syntax and semantics of Sequence Datalog. A formal development can be found in [3].

2.1 Syntax

To manipulate sequences, *Sequence Datalog* has two interpreted function symbols for constructing complex terms, one for concatenating sequences and one for extracting subsequences. Intuitively, if X and Y are sequences, and I and J are integers, then the term $X \bullet Y$ denotes the concatenation of X and Y, and the term $X[I : J]$ denotes the subsequence of X extending from position I to position J. For example, the following rule extracts all prefixes of sequences in relation R: $\quad prefix(X[1{:}N]) \leftarrow R(X)$. For each sequence X in R, this rule says that a prefix of X is any subsequence starting with the first element and ending with the N-th element, so long as N is no longer than the length of X. The following rule constructs all possible concatenations of sequences in relation R: $\quad answer(X \bullet Y) \leftarrow R(X),\ R(Y)$. This rule takes any pair of sequences, X and Y, in relation R, concatenates them, and stores the result in $answer$.

To be more precise, the language of terms uses three countable, disjoint sets: a set of constant symbols, $a, b, c, \ldots$, called the *alphabet* and denoted Σ; a set of variables, $R, S, T, \ldots$, called *sequence variables* and denoted V_Σ; and another set of variables, $I, J, K, \ldots$, called *index variables* and denoted V_I. A constant sequence (or *sequence*, for short) is an element of Σ^*. From these sets, we construct two kinds of terms as follows:

- *index terms* are built from integers, index variables, and the special symbol *end*, by combining them recursively using the binary connectives $+$ and $-$. Thus, if N and M are index variables, then 3, $N+3$, $N-M$, $end-5$ and $end-5+M$ are all index terms.
- *sequence terms* are built from constant sequences, sequence variables and index terms, by combining them recursively into *indexed terms* and *constructive terms*, as follows: (*i*) If s is a sequence variable and n_1, n_2 are *index terms*, then $s[n_1{:}n_2]$ is an *indexed sequence term*. n_1 and n_2 are called the *indexes* of s. As a shorthand, each sequence term of the form $s[n_i{:}n_i]$ is written $s[n_i]$. (*ii*) If s_1, s_2 are *sequence terms*, then $s_1 \bullet s_2$ is a *constructive sequence term*. Thus, if S_1 and S_2 are sequence variables, *ccgt* is a sequence and N is an index variable, then $S_1[4]$, $S_1[1{:}N]$, and $ccgt \bullet S_1[1{:}end-1] \bullet S_2$ are all sequence terms.

A substitution, θ, is a mapping that associates a sequence with each sequence variable in V_Σ, and an integer with each index variable in V_I. Substitutions can be extended to partial mappings on sequence and index *terms* in a straightforward way. Because these terms are interpreted, the result of a substitution is either a sequence or an integer. For example, if n_1 and n_2 are index terms, then $\theta(n_1 \pm n_2) = \theta(n_1) \pm \theta(n_2)$. Similarly, if $s[n_1{:}n_2]$ is a sequence term, then $\theta(s[n_1{:}n_2])$ is defined iff $1 \leq \theta(n_1) \leq \theta(n_2) + 1 \leq \mathrm{LEN}(\theta(s)) + 1$. In particular, $\theta(s[n_1{:}n_2])$ is the contiguous subsequence of $\theta(s)$ extending from position $\theta(n_1)$ to position $\theta(n_2)$. However, there are some subtleties when the index terms take on "fringe" values, as shown in the following examples; note how terms such as $s[n+1{:}n]$ are conveniently interpreted as the empty sequence, ϵ.

$\theta(uvwx[3:5])$ is undefined	$\theta(uvwx[3:4]) = wx$	$\theta(uvwx[3:3]) = w$
$\theta(uvwx[3:2]) = \epsilon$	$\theta(uvwx[3:1])$ is undefined	$\theta(uvwx[3:0])$ is undefined

If the special index term *end* appears in the sequence term $s[n_1{:}n_2]$, then *end* is interpreted as the length of $\theta(s)$. Thus, $\theta(s[n{:}end])$ is a suffix of $\theta(s)$. Finally, $\theta(s_1 \bullet s_2)$ is interpreted as the concatenation of $\theta(s_1)$ and $\theta(s_2)$, *e.g.*, $\theta(abc \bullet def) = abcdef$.

As in most logics, the language of formulas for *SequenceDatalog* includes a countable set of predicate symbols, $p, q, r, \ldots$, where each predicate symbol has an associated arity. If p is a predicate symbol of arity n, and $s_1, \ldots, s_n$ are sequence terms, then $p(s_1, \ldots, s_n)$ is an atom. Moreover, if s_1 and s_2 are sequence terms, then $s_1 = s_2$ and $s_1 \neq s_2$ are also atoms. From atoms, we build *rules, facts* and *clauses* in the usual way [9]. The head and body of a clause, γ, are denoted $\mathrm{HEAD}(\gamma)$ and $\mathrm{BODY}(\gamma)$, respectively. A clause that contains a constructive term in its head is called a *constructive clause.* A Sequence Datalog *program* is a set of Sequence Datalog rules in which constructive terms may appear in rule heads, but not in rule bodies.

We say that a variable, X, is *guarded* in a clause if X occurs in the body of the clause as an argument of some predicate. Otherwise, we say that X is *unguarded.* For example, X is guarded in $p(X[1]) \leftarrow q(X)$, whereas it is unguarded in $p(X) \leftarrow q(X[1])$. A rule is *guarded* if all variables in the rule are guarded. A program is *guarded* if all rules of the program are guarded.[1]

2.2 Semantics

The formal semantics of Sequence Datalog is developed in [3]. Here, we review the main ideas. As in classical logic programming [9], each Sequence Datalog program, P, has an associated T-operator that maps databases to databases. Each application of the T-operator may create new atoms, which may contain new sequences. The T-operator is monotonic and continuous and has a least fixpoint [3]. If the least fixpoint is finite, we say that P has a *finite semantics.*

The universe of sequences over the alphabet, Σ, is infinite. Thus, to keep the semantics of programs finite, we do not evaluate rules over the entire universe, Σ^*. Instead, we introduce a new active domain for sequence databases, called the *extended active domain.* This domain contains all the sequences occurring in the database, plus all their *subsequences.*[2] Substitutions range over this domain when rules are evaluated. Note that the size of the *extended domain* is at most quadratic in the size of the database domain. In fact, the number of different contiguous subsequences of a given sequence of length k is at most $\sum_{i=0}^{2} \binom{k}{i}$, that is, $\frac{k(k+1)}{2} + 1$.

[1] As it will become clear in the following, because of the active–domain semantics, variables in Sequence Datalog clauses need not be guarded. However, in this paper we will mainly refer to guarded programs.

[2] In this paper, we always refer to *contiguous subsequences*, that is, subsequences specified by a start and end position in some other sequence. Thus, *bcd* is a contiguous subsequence of *abcde*, whereas *bd* is not.

The extended active domain is not fixed during query evaluation. Instead, whenever a new sequence is created (by the concatenation operator, $\bullet$), the new sequence—and its subsequences—are added to the extended active domain. The fixpoint theory of Sequence Datalog provides a declarative semantics for this apparently procedural notion [3]. In the fixpoint theory, the extended active domain of the least fixpoint is larger than the extended active domain of the database. For the database, the domain consists of the sequences in the database and all their subsequences. For the least fixpoint, the domain consists of the sequences in the database and any new sequences created during rule evaluation, and all their subsequences. The concatenated sequences (and their subsequences) form the extended active domain of the least fixpoint.

To be more precise, we define the fixpoint semantics of a program, P, over a database, DB, as follows:

- The *extended active domain* of a database, DB, with respect to a program, P, is denoted $\mathcal{D}^{ext}_{P,\mathrm{DB}}$. It is the union of the following three sets: (i) the *active domain* of the database and the program, that is, the set of sequences occurring in DB and P; (ii) all the contiguous subsequences of the sequences in the active domain; and (iii) the set of integers $\{0, 1, 2, \ldots, l_0 + 1\}$, where l_0 is the maximum length of a sequence in the active domain.

- The operator $T_{P,\mathrm{DB}}$ associated with program P and database DB maps interpretations — i.e., sets of ground atomic formulas — to interpretations. In particular, if I is an interpretation, then $T_{P,\mathrm{DB}}(I)$ is the following interpretation [3]:

$$\{\theta(\textsc{head}(\gamma)) \mid \theta(\textsc{body}(\gamma)) \subseteq I \text{ \textit{for some clause} } \gamma \in P \cup \mathrm{DB} \text{ \textit{and some substitution} } \theta \text{ \textit{based on} } \mathcal{D}^{ext}_{I} \text{ and defined at } \gamma\}$$

  The *least fixpoint* [9] of the *operator* $T_{P,\mathrm{DB}}$ is computed in a bottom-up fashion, by starting at the database, DB, and applying the operator repeatedly until a fixpoint is reached [9].

- At each step, if an inferred fact contains a new sequence (*i.e.*, a sequence not currently in the *extended active domain*), then it is added to the active domain. Thus, as the bottom-up computation proceeds, the extended active domain may expand. At each step of the computation, substitutions range over the current value of the extended active domain.

Note that the least fixpoint can be an infinite set. In this case, we say that the semantics of P over DB is *infinite*; otherwise, it is *finite*.

3 Domain Bounded Recursion by Examples

As discussed in the previous sections, computations over sequences may become infinite even when the underlying alphabet is finite. We are interested in studying *finite* programs, that is, programs that have a finite semantics (*i.e.* a finite least fixpoint) over every input database.

As is typical with powerful logics, the finiteness property for Sequence Datalog programs is in general undecidable [3]. Thus, one of our aims is to develop subsets of the logic that are finite. We first note that a necessary condition for infiniteness is the generation of sequences of unbounded length. To do this, programs must use recursion through construction. That is, newly computed sequences must be used recursively to construct more new sequences. This kind of computation is closely related to a particular form of constructive rule, which we call *recursive constructive rules*. In such rules, the predicate in the head depends on itself. To formalize this concept, we use the notion of a *predicate dependency graph* of a Sequence Datalog program.[3] This notion, and several others, are closely related:

- Predicate p is a *constructive predicate* in program P if P contains a constructive rule for p, that is, a rule with a constructive term (a term containing $\bullet$) in its head. Note that constructive predicates cause new sequences to be added to the domain during query evaluation.
- Predicate p *depends on* predicate q in program P if P contains a rule in which p is the predicate symbol in the head and q is a predicate symbol in the body. If the rule is constructive, then p *depends constructively* on q.[4]
- The *predicate dependency graph*, PDG_P, of program P is a directed graph representing the binary relation *"depends on"* over the predicates of P. An edge (p, q) in this graph is a *constructive edge* if p depends constructively on q.
- Predicate p is *recursive with respect to construction* in program P if the predicate dependency graph for P contains a cycle passing through p with a constructive edge.

The simplest way to enforce finiteness in the presence of constructive rules is to disallow recursion through construction. This means forbidding programs whose predicate dependency graph contains cycles with constructive edges. However, we have shown [3] that the resulting language, called Stratified Sequence Datalog, has rather limited expressive power, due to its limited ability to restructure sequences. Many natural, simple and low-complexity restructurings—such as reversing a sequence or computing its complement—require constructive recursion, and cannot be expressed in Stratified Sequence Datalog. For these operations, the number of concatenations depends on the database. In contrast, in Stratified Sequence Datalog, the number of concatenations is fixed for each program and is independent of the database.

In this section we show how it is possible to increase expressiveness while preserving finiteness. The examples develop the idea that constructive recursion (which is unsafe) can be limited and controlled by structural recursion (which is always safe). This is the main idea of this paper, and the basis for the syntactic restrictions developed in the next sections. Consider, for example, the next program. It restructures the sequences in the database, producing new sequences longer than any in the database. The first version of the program does it in a straightforward way, but has an infinite semantics. The second version solves the same problem, but with a finite semantics.

Example 3.1 **[Infinite Semantics]** Suppose R is a unary relation containing a set of sequences. For each sequence X in R, we want the sequence obtained by repeating each symbol in X twice. For example, given the sequence $abcd$, we want the sequence $aabbccdd$. We call these sequences *echo* sequences. The easiest way to define echo sequences is with the following program:

[3] Our notion of predicate dependency graph is a variant of the one introduced in [1] for stratified negation.

[4] For clarity's sake, this paper supposes that only guarded variables occur in programs. With some added complexity, the definitions can be adjusted to the more general case.

$$
\begin{array}{llll}
\gamma_1: & answer(X,Y) & \leftarrow & R(X), echo(X,Y). \\
\gamma_2: & echo(\epsilon,\epsilon) & \leftarrow & true. \\
\gamma_3: & echo(X, X[1] \bullet X[1] \bullet Z) & \leftarrow & echo(X[2{:}end], Z).
\end{array}
$$

The first rule retrieves every sequence in relation R and its echo, by invoking the predicate $echo(X,Y)$. The last two rules specify what an echo sequence is. For every sequence, X, in the extended active domain, these rules generate its echo sequence, Y. Starting with $X = \epsilon$ and $Y = \epsilon$, they recursively concatenate single characters onto X while concatenating two copies of the same character onto Y. As new sequences are generated, they are added to the active domain, which expands indefinitely.

This program has an infinite semantics over every database that contains a non-empty sequence. This is because the rules defining $echo(X,Y)$ recursively generate longer and longer sequences without bound. For example, suppose the input database contains only one tuple, $\{R(aa)\}$. Its extended active domain consists of the sequences ϵ, a, aa. During a bottom-up computation of the least fixpoint, whenever rule γ_3 is fired, the inferred facts and the extended domain both grow. The least fixpoint of the T operator is therefore infinite, and its extended active domain is the set of all sequences made of a's. Note that the query answer consists of a single atom, $answer(aa, aaaa)$. Thus, although the least fixpoint is infinite, the query answer is not. The next program expresses the query in such a way that both the answer and the least fixpoint are finite.

$$
\begin{array}{llll}
\gamma_{1'}: & answer(X,Y) & \leftarrow & R(X),\ echo'(X,Y). \\
\gamma_{2'}: & echo'(\epsilon,\epsilon) & \leftarrow & true. \\
\gamma_{3'}: & echo'(X[1{:}N+1], Z \bullet X[N+1] \bullet X[N+1]) & \leftarrow & R(X),\ echo'(X[1{:}N], Z).
\end{array}
$$

In this program, the sequences in relation R act as input for the third rule, which defines the predicate $echo'(X,Y)$. This rule recursively scans each input sequence, X, while constructing an output sequence, Y. For each character in the input sequence, two copies of the character are appended to the output sequence. The rule computes the echo of every prefix of every sequence in R. The first rule then retrieves the echoes of the sequences in R.

Like rules γ_1–γ_3 above, the program made of rules $\gamma_{1'}$–$\gamma_{3'}$ involves constructive recursion. However, in the latter case, the least fixpoint is finite. This is because constructive recursion does not go on indefinitely, but terminates as soon as the input sequences have been scanned. In essence, growing terms of upwardly bounded length are used to guarantee termination: these terms "grow" at each recursive evaluation of the rule, and recursion stops when the upper bound has been reached. In this way, structural recursion over the first argument controls and limits constructive recursion over the second argument. $\diamondsuit$

As shown by Example 3.1, in some patterns of recursion the length of newly constructed sequences can be bounded above by using structural recursion to control constructive recursion in such a way that the recursive construction of new sequences proceeds up to a certain length and then stops. In these cases, the length of constructed sequences is bounded above by the size of the active domain of the database, that is, by the sum of the lengths of all sequences in the database. Recursion therefore stops after a finite amount of time, depending on the size of the domain. We call these forms of recursion *domain–bounded recursion*.

The notion of domain–bounded recursion can also be used as a basis for optimizing rule-evaluation.

In fact, it suggests a rule-rewriting technique that in some cases may significantly improve the evaluation of programs, as shown in the following example.

Example 3.2 **[Complement]** Suppose R is a base relation storing a set of binary sequences, and we wish to compute the complement of all the sequences in R. There are two ways of expressing this transformation in Sequence Datalog; both are finite, but the second has lower complexity.

The first solution uses a standard logic-programming approach involving tail recursion:

$$
\begin{array}{lll}
\gamma_1: & answer(Y) & \leftarrow \quad R(X), \quad complement(X, Y). \\
\gamma_2: & complement(\epsilon, \epsilon) & \leftarrow \quad true. \\
\gamma_3: & complement(X,\ Y \bullet Z) & \leftarrow \quad compl(X[1],\ Y),\ complement(X[2{:}end],\ Z).
\end{array}
$$

Here, we assume that the atoms $compl(0, 1)$ and $compl(1, 0)$ are in the database. In this case, predicate *complement* constructs the complement of every sequence in the extended active domain of the database. Recursion starts with the empty sequence, and proceeds until the complement of every sequence in the extended active domain has been generated. Since sequences do not grow as a result of evaluating predicate *complement*—the complementary sequence has the same length as the original one—the active domain semantics prevents the generation of sequences of unbounded length, and recursion stops after a finite amount of time.

Although they have a finite semantics, rules γ_1 to γ_3 are still highly inefficient, since they compute the complement of every sequence in the extended active domain of the database, not just the complement of sequences in relation R. Since the extended active domain has polynomial (quadratic) size wrt the database domain, a polynomial number of unnecessary complement sequences are computed. This inefficiency can be avoided by expressing the query in another, more-efficient way, again using domain–bounded recursion:

$$
\begin{array}{lll}
\gamma_{1'}: & answer(Y) & \leftarrow \quad R(X), \quad complement'(X, Y). \\
\gamma_{2'}: & complement'(\epsilon, \epsilon) & \leftarrow \quad true. \\
\gamma_{3'}: & complement'(X[1{:}N+1], Z \bullet Y) & \leftarrow \quad R(X),\ compl(X[N+1], Y),\ complement'(X[1{:}N], Z).
\end{array}
$$

In these rules, each sequence in relation R is scanned from beginning to end, and in the process, the complementary sequence is constructed one symbol at a time. Note that since the complement is generated only for sequences in relation R, rule evaluation requires only a linear number of database accesses (linear in the sum of the lengths of all sequences in R). $\diamondsuit$

The next sections formally define the notion of domain–bounded recursion.

4 Domain Bounded Recursion: Definition

4.1 Reasoning about Length

To determine if a program is finite, we need to reason about the lengths of any new sequences created by the program. If these lengths can be bounded, then the program is finite. This section develops a simple formalism for comparing the "lengths" of two sequence terms. As a first step, we develop the notion of the *symbolic length* of a sequence term. This is an arithmetic expression in which symbols and numbers can appear. For example, if X is a sequence variable, then its symbolic length is the symbol

L_X. Likewise, if $X \bullet Y$ is a sequence term, then its symbolic length is $L_X + L_Y$. The symbolic length of a constant sequence (*e.g.*, *actg*) is its actual length (*e.g.*, 4). Such expressions allow us to reason about the lengths of partially specified sequences. The reasoning is tractable because we are dealing with just a tiny subset of arithmetic.

To reason about sequence terms such as $X[N : M]$, we need to reason about the index terms N and M. We therefore introduce the notion of the *symbolic value* of an index term. Like symbolic lengths, symbolic values are arithmetic expressions in which numbers and symbols can appear. For example, if N is an index variable, then its symbolic value is the symbol V_N. In general, the symbolic value of an index term depends on the sequence term in which it is embedded. For example, in the sequence term $X[N : end]$, the index term *end* represents the last position in the sequence X. Thus, in the term $acgt[2 : end]$, the symbolic value of *end* is 4, while in the term $actgactg[2 : end]$, its symbolic value is 8. The following definition makes these ideas precise.

Definition 1 (Symbolic Length and Value) *The* symbolic length *of a sequence term, s, is an arithmetic expression, denoted $L(s)$. The* symbolic value *of an index term, n, in the context of s is also an arithmetic expression, denoted $V(n, s)$. These expressions are built from integers, two binary connectives ($+$ and $-$), and a collection of symbols. They are constructed in a mutually recursive fashion as follows:*

– *Symbolic Lengths:*

- *If s is a constant sequence in Σ^*, then $L(s)$ is the length of s. e.g.,* $L(acgt) = 4$.
- *If X is a sequence variable in V_Σ, then $L(X)$ is the symbol L_X.*
- $L(s_1 \bullet s_2) = L(s_1) + L(s_2)$. *e.g.,* $L(X \bullet atcg) = L_X + 4$.
- $L(s[n_1{:}n_2]) = V(n_2, s) - V(n_1, s) + 1$.

– *Symbolic Values:*

- *If n is an integer, then $V(n, s) = n$.*
- *If N is an index variable in V_I, then $V(N, s)$ is the symbol V_N.*
- $V(end, s) = L(s)$. *e.g.,* $V(end, X) = L_X$ $\quad V(end, atcgatcg) = 8$.
- $V(n_1 \pm n_2, s) = V(n_1, s) \pm V(n_2, s)$. *e.g.,* $V(N + 3, X) = V_N + 3$; $V(end - N, X) = L_X - V_N$.

Here are some sequence terms and their symbolic lengths:

$$L(X[3{:}N]) = V_N - 3 + 1; \qquad L(atcgatcgatcg[3 + N{:}end - M]) = (12 - V_M) - (3 + V_N) + 1$$
$$L(X[3{:}end]) = L_X - 3 + 1; \qquad L(X[3{:}N] \bullet Y[N{:}end]) = (V_N - 3 + 1) + (L_Y - V_N + 1)$$

Symbolic lengths can be manipulated in a variety of ways. For instance, we can add and subtract two symbolic lengths to obtain another symbolic length. In some situations, we can also evaluate a symbolic length to obtain an integer. For example, if a symbolic length contains only integers and no symbols, then it can be evaluated in the normal way. Even if a symbolic length contains symbols, these symbols may cancel out, so the expression can be evaluated; *e.g.*, the value of $L_X + 4 - L_X - 2$ is 2. This gives two well-defined situations in which symbolic lengths can be evaluated. This idea gives us a mechanism with which to compare two symbolic lengths. We say that two sequence terms s_1, s_2 are *comparable* if the symbolic expression $L(s_1) - L(s_2)$ can be evaluated,[5] to yield an integer, k. If $k > 0$ then s_1 is *longer* than s_2. If $k = 0$ then s_1 is *the same length* as s_2.

For example, the sequence term $s_1 = actg \bullet Y$ is longer than the sequence term $s_2 = Y$. To see this, note that $L(s_1) = 4 + L_Y$ and $L(s_2) = L_Y$, so $L(s_1) - L(s_2)$ evaluates to 4, a positive integer. Similarly, the term $s_3 = X[5{:}N]$ is longer than the term $s_4 = X[5{:}N-3]$. In this case, $L(s_3) = V_N - 5 + 1$ and $L(s_4) = V_N - 3 - 5 + 1$, so $L(s_3) - L(s_4)$ evaluates to 3, a positive integer.

On the other hand, the terms $s_5 = S[1{:}N]$ and $s_6 = S[1{:}M]$ are incomparable. To see this, note that $L(s_5) - L(s_6)$ reduces to $V_N - V_M$, which cannot be evaluated. Similarly, the terms $s_7 = X \bullet Y$ and $s_8 = Y$ are incomparable. In this case, $L(s_7) - L(s_8)$ reduces to L_X, which cannot be evaluated.

4.2 Constrained Variables and Growing Attributes

Another notion that we need is *constrained variables.* Intuitively, we need to infer when a variable ranges over a fixed domain that does not grow during query evaluation. For example, in the rule $p(X[1:3]) \leftarrow q(X)$, the variable X is constrained, since it is guarded by q, so that X is forced to range over sequences in relation q (recall that we say that a variable, X, is guarded in a clause if X occurs in the body of the clause as an argument of some predicate). However, in the rule $p(X) \leftarrow q(X[1:3])$, variable X is *not* constrained. To see this, note that X is unguarded; suppose the database contains the fact $q(abc)$. Then the index term $X[1:3]$ can take on the value abc, which means that X can be any sequence that has abc as a *prefix.* Thus, X can range over an infinite domain, including sequences of unbounded length. These ideas motivate the following definition of *constrained* variable . In this definition, and throughout the paper, we use the notation (p, i) to refer to the i^{th} attribute (or argument) of predicate p.

Definition 2 (Constrained Variables) *We say that a sequence variable S is* constrained *by predicate p in rule γ if at least one of the following holds: (i) variable S is guarded by predicate p in the body of γ; (ii) the body of γ contains an equality atom of the form $S = S_1[N_1{:}N_2]$ where S_1 is guarded by p in the body of γ.*

There are some cases in which it is easy to see that an attribute of a predicate "grows" during the bottom-up computation. For example, consider the following rules, where q is a base predicate:

$$\begin{array}{lcl} p(X, \epsilon) & \leftarrow & q(X). \\ p(X, X[1:N+1]) & \leftarrow & q(X),\ p(X, X[1:N]). \end{array}$$

Here, $p(X, Y)$ is true iff X is a sequence in q and Y is a prefix of X. To see this, note that if X is a sequence in q, then $p(X, \epsilon)$ is true, by the first rule. Then, using the second rule, $p(X, X[1:1])$ is true, then $p(X, X[1:2])$ is true, then $p(X, X[1:3])$, and so on up to $p(X, X)$. After this, $X[1:N]$ is undefined, so recursion stops. The rules thus scan each sequence in q from beginning to end, which is a canonical example of structural recursion. There are two points to observe here. (i) In both rules, variable X is *constrained* by the predicate q. (ii) The second attribute of p *grows* with each bottom-up application of the rules. The notion of "growth" can be made precise by comparing the symbolic lengths of terms in

[5] Sometimes, two symbolic lengths can be intuitively compared even if their difference cannot be evaluated. For example, we can easily conclude that every instantiation of the term $s_1 = X \bullet Y \bullet a$ is longer than the corresponding instantiation of the term $s_2 = X$, even though their difference $L(s_1) - L(s_2) = L_Y + 1$ cannot be evaluated. However, we shall ignore this possibility, since taking it into account would complicate the definitions and the theoretical development without increasing the expressibility of the formalism.

the head and body of a rule. In this case, $X[1:N+1]$ is longer than $X[1:N]$. The following definition generalizes this idea.

Definition 3 (Growing attributes) *Suppose predicate p occurs in the head and body of a rule. Suppose the sequence term in attribute (p,k) in the head is not a constructive term. Attribute (p,k)* grows *in the rule if the sequence term in attribute (p,k) in the head is longer than the sequence term in attribute (p,k) in the body. In addition, attribute (p,k)* does not shrink *in the rule if the sequence term in attribute (p,k) in the head is longer than or the same length as the sequence term in attribute (p,k) in the body.*

4.3 Domain–Bounded Programs

We have now developed the concepts needed to define Domain–Bounded Recursion. The idea is to allow recursion through construction, but in a controlled and limited way. The result is a class of Sequence Datalog programs that we call *domain–bounded programs.*

The first restriction we impose on Sequence Datalog is that recursive constructive rules be linear. Recall that a rule is *linear* iff the predicate in the head is mutually recursive with the predicate of at most one atom in the body [2]. Actually, in order to keep the technical development as simple as possible, we require more than mere linearity: we disallow *mutual* recursion through construction. We call this *simple recursion* through construction. The rulebases in Example 3.1, defining the predicates *echo* and *echo*$'$, are both simple recursive. This property of a program can easily be checked in polynomial time (polynomial in the number of rules).

It is important to note that: (i) mutual recursion and non-linear recursion are still allowed. However, they are not allowed in constructive rules. We thus have all the power of classical Datalog at our disposal (since Datalog is a subset of Sequence Datalog); (ii) abolishing mutual recursion through construction does not limit our expressive power, since mutual recursion can always be reduced to non-mutual recursion; (iii) the definition of domain–bounded recursion and the expressibility results can be extended to programs with mutual recursion, but non-mutual recursion makes the treatment much simpler and elegant. In fact, in simple recursive rules, the predicate symbol in the head must also occur once in the body of the rule. This means that if p is the predicate in the head, then for every attribute (p,i), we can try to compare the length of the term in attribute (p,i) in the body with the length of the corresponding term in the head. In particular, we are interested in attributes that grow from body to head, according to Definition 3.

Definition 4 (Set of growing attributes) *We say that a predicate p has a* set of growing attributes*, $\{(p,i_1),\ (p,i_2),\ldots,\ (p,i_n)\}$, if, for each simple recursive rule γ such that p occurs in* $\text{HEAD}(\gamma)$ *it is the case that: (i) at least one attribute (p,i_k) grows in γ; (ii) none of the other attributes (p,i_j) in the set shrinks.*

We can now define the notion of *domain–bounded program.*

Definition 5 (Domain–bounded program) *A program is* domain–bounded *if every recursive constructive rule, γ, in the program satisfies the following conditions: (i) the rule is simple recursive; (ii)*

the predicate in the head, p, has a set of growing attributes; (iii) *every variable associated with a growing attribute is constrained by some predicate q different from p in γ.*

5 Complexity and Expressibility

In this section we prove a number of results: (i) that domain–bounded programs are finite, (ii) that their data complexity is complete for elementary time, and (iii) that they express exactly the class of elementary sequence functions [10], that is, the class of sequence functions with hyper-exponential time complexity. Thus, although finite, domain–bounded recursion is highly expressive. As a simple illustration, the following example shows that domain–bounded programs can generate sequences of exponential length.

Example 5.1 **[Long sequences]** The following program is domain–bounded:

$$
\begin{array}{lcl}
doubling(\epsilon, 1) & \leftarrow & true. \\
doubling(X[1{:}N+1], Y \bullet Y) & \leftarrow & R(X),\ doubling(X[1{:}N], Y).
\end{array}
$$

Given a sequence, σ, of length n in predicate R, the predicate *doubling* computes a sequence of 1's of length 2^n by doubling the length of a unit sequence n times. $\diamondsuit$

This paper addresses the complexity and expressibility of domain–bounded programs in terms of sequence functions. While a *sequence query* is a partial mapping from the set of databases over Σ to itself, a *sequence function* [4] is a partial mapping from Σ^* to itself. A sequence function is *computable* if it is partial recursive. Sequence functions can be thought of as queries from a database, $\{input(\sigma_{in})\}$, containing a single sequence tuple, to a database, $\{output(\sigma_{out})\}$, containing a single sequence tuple. Given a sequence function, f, the *complexity* of f is defined in the usual way, as the complexity of computing $f(\sigma)$, measured with respect to the length of the sequence σ. A query language, L, is said to *express* a complexity class, $\textsc{c}$, of sequence functions if: *(i)* each sequence function expressible in L has complexity in $\textsc{c}$ and conversely, *(ii)* each sequence function with complexity in $\textsc{c}$ can be expressed in L.

The class of *elementary sequence functions*, $\mathcal{E}$, is defined in terms of the *hyper-exponential functions*, $hyp_i(n)$. These latter functions are defined recursively as follows: (i) $hyp_1(n) = n$; (ii) $hyp_{i+1}(n) = 2^{hyp_i(n)}$, for $i > 1$. hyp_i is called the hyper-exponential function of level i. The set of *elementary sequence functions* is the set of sequence functions that have hyper-exponential time complexity, that is, the set of sequence functions: $\mathcal{E} = \bigcup_{i \geq 1} \text{DTIME}[hyp_i(\mathcal{O}(n))]$

The following theorem is the main result of this section. An immediate corollary is that the data complexity of Sequence Datalog programs is complete for elementary time.

Theorem 1 (Finiteness and Expressibility) *Every domain–bounded program P is finite. Moreover, Domain–bounded programs express the class $\mathcal{E}$, of elementary sequence functions.*

Proof Sketch: (Finiteness) Let us consider a domain–bounded program, P, and its predicate dependency graph, $\textsc{pdg}_P$. If there are k strongly connected components in the graph, then we can linearize the components by a topological sort; *i.e.*, we can assign a distinct integer $i \in \{1, \ldots, k\}$ to each component so that if there is an arc from component i to component j then $i < j$. Let us call $\mathcal{N}_1, \ldots, \mathcal{N}_k$ the linearized components. The linearization induces a stratification on the program P, where the i^{th} stratum consists of those rules in P that define predicates in $\mathcal{N}_i$. Let P_i denote the i^{th} stratum of P. The P_i are disjoint, and $P = P_1 \cup P_2 \cup \cdots \cup P_k$.

Because P is guarded, the extent of a predicate defined in P_i depends only on the rules in $P_1 \cup \cdots \cup P_i$. We can therefore apply the rules in P in a bottom-up fashion, one stratum at a time. Formally, given a database DB, we define a sequence of minimal models $\mathcal{M}_1, \ldots, \mathcal{M}_k$, $\mathcal{M}_{k+1}$ as follows: (i) $\mathcal{M}_1 = \text{DB}$; (ii) $\mathcal{M}_{i+1} = T_{P_i,\mathcal{M}_i} \uparrow \omega$ *for* $1 \leq i \leq k$. Moreover, $\mathcal{M}_{k+1}$ is the minimal model of P and DB.

Our first goal is to bound the size of the extended domain of each $\mathcal{M}_i$. To do this, note that size of the extended domain is $\mathcal{O}(d^{l_i})$, where d is the alphabet size and l_i is the length of the longest sequence in $\mathcal{M}_i$. Thus, it is enough to bound l_i. We do this by induction on i. In particular, we show that $l_i = hyp_i(\mathcal{O}(n))$, where $n = l_1$ is the length of the longest sequence in the database, DB.

– *(Base case)* Follows immediately, since $hyp_1(n) = n = l_1$, by definition.

– *(Inductive case)* Suppose that $l_i = hyp_i(\mathcal{O}(n))$. To show that $l_{i+1} = hyp_{i+1}(\mathcal{O}(n))$, we distinguish two cases: *(i)* $\mathcal{N}_i$ does not contain predicates involved in constructive loops, that is, there is no simple constructive rule in P_i; *(ii)* $\mathcal{N}_i$ is a singleton component having a constructive loop, that is, there is at least one simple constructive rule in P_i.

First consider case (i). Since there is no recursion through construction, $l_{i+1} = \mathcal{O}(l_i) = hyp_i(\mathcal{O}(n))$. Hence $l_{i+1} = hyp_{i+1}(\mathcal{O}(n))$.

Now consider case (ii), in which $\mathcal{N}_i$ is a singleton component involved in a constructive loop. P_i thus defines a single recursive predicate. Let h be the number of growing attributes for this predicate. By the definition of domain-bounded programs, (i) each application of a recursive rule makes one of the growing attributes grow and none shrink, and (ii) all the variables associated with a growing attribute are constrained by some predicate defined in a lower stratum. Condition (ii) means that these variables can bind only to sequences in $\mathcal{M}_i$, and so their lengths are bounded above l_i. Combined with condition (i), this means that the bottom-up computation for P_i must saturate after at most hl_i steps. Now, let $s_1 \bullet s_2 \bullet \ldots \bullet s_m$ be a constructive term in the head of a rule in P_i with a maximal number of $\bullet$ operators. The length of the computed sequences therefore grows by at most a factor of m each time the rules in P_i are fired. The maximum length of a computed sequence is therefore $\mathcal{O}(m^{hl_i}) = 2^{\mathcal{O}(l_i)}$. Thus, if l_i is hyper-exponential, then so is l_{i+1}. In particular, if $l_i = hyp_i(\mathcal{O}(n))$ then $l_{i+1} = hyp_{i+1}(\mathcal{O}(n))$.

This shows that l_{k+1} is of hyper-exponential size. It follows that $\mathcal{M}_{k+1}$, the minimal model of P, is finite and of hyper-exponential size.

(Expressibility) The upper complexity bound follows immediately from the fact that the size of the least fixpoint of a domain program P is hyper-exponential in the size of the database; thus, the minimum model can be computed in hyper-exponential time with respect to the size of the input database.

We now prove that domain–bounded programs are expressively complete for the elementary sequence functions; that is, any elementary sequence function can be expressed by one of these programs. More specifically, given a sequence function, f, and an input sequence, σ_{in}, we encode the input sequence as a unary tuple in predicate *input*, and compute the output sequence, $\sigma_{out} = f(\sigma_{in})$, in predicate *output*.

Since f is an elementary function, there is a Turing machine M_f which runs in hyper-exponential time and computes f. We will prove the result by simulating the computation of M_f over σ_{in}. The crux in the proof is the construction of a unary hyper-exponential counter that will be used during the simulation to mark time and space; if n is the length of the input sequence, and Turing machine M_f runs in time $hyp_i(n)$, we need a counter from 1 to $hyp_i(n)$. To do this, we use a technique similar to the one in Example 5.1 to produce a sequence of length 2^n, and then re-apply to this new sequence the same rules i-times to produce a sequence of length $hyp_i(n)$. Once this counter has been built, we can use the counter to mark tape cells and keep track of time, and easily simulate the computation of M_f over σ_{in} using quite standard techniques. It is easy to show that the sequence in predicate *output* is the output of function f on input σ_{in}. This completes the proof. $\diamondsuit$

References

[1] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kauffman, Los Altos, 1988.

[2] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *ACM SIGMOD International Conf. on Management of Data (SIGMOD'86), Washington, D.C.*, pages 16–52, 1986.

[3] A. J. Bonner and G. Mecca. Sequences, Datalog and Transducers. *Journal of Computing and System Sciences*, Special Issue on PODS'95(57):234–259, 3 1998. http://www.difa.unibas.it/users/gmecca.

[4] L. S. Colby, E. L. Robertson, L. V. Saxton, and D. Van Gucht. A query language for list-based complex objects. In *Thirteenth ACM SIGMOD Intern. Symposium on Principles of Database Systems (PODS'94)*, pages 179–189, 1994.

[5] S. Ginsburg and X. Wang. Pattern matching by RS-operations: towards a unified approach to querying sequence data. In *Eleventh ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS'92)*, pages 293–300, 1992.

[6] G. Grahne, M. Nykanen, and E. Ukkonen. Reasoning about strings in databases. In *Thirteenth ACM SIGMOD Intern. Symposium on Principles of Database Systems (PODS'94)*, pages 303–312, 1994.

[7] G. Grahne and E. Waller. How to make SQL stand for String Query Language ? In *Seventh Intern. Workshop on Database Programming Languages (DBPL'99), Kinloch Rannoch, Scotland*, 1999.

[8] S. Grumbach and T. Milo. An algebra for POMSETS. In *Fifth International Conference on Data Base Theory, (ICDT'95), Prague, Lecture Notes in Computer Science, Springer-Verlag*, pages 191–207, 1995.

[9] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[10] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[11] K. Sohn and A. Van Gelder. Termination detection in logic programs using argument sizes. In *Tenth ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS'91)*, pages 216–226, 1991.