Finite Query Languages for Sequence Databases

Giansalvatore Mecca*

Universitá di Roma "La Sapienza"
Dipartimento di Informatica e Sistemistica
Via Salaria, 113 — 00198 Roma, Italy

mecca@infokit.dis.uniroma1.it

Anthony J. Bonner†

University of Toronto
Department of Computer Science
Toronto, Ontario — Canada M5S 1A4

bonner@db.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. It is a subset of a recently developed logic called *Sequence Datalog* [22]. Sequence Datalog distinguishes syntactically between *subsequence extraction* and *sequence construction*. Extraction creates sequences of bounded length, and leads to *safe* recursion; while construction can create sequences of arbitrary length, and leads to *unsafe* recursion. In this paper, we develop syntactic restrictions for *Sequence Datalog* that allow sequence construction but preserve finiteness. The main idea is to use safe recursion to control and limit unsafe recursion. The main results are a finite language, called *Weakly Constructive Sequence Datalog*, and a characterization of its complexity and expressive power. Although finite, the new language is highly expressive, since its data complexity is complete for the elementary functions.

Appears in *Proceedings of the Fifth International Workshop on Database Programming Languages (DBPL)*, September 6–8, 1995, Gubbio, Umbria, Italy.

This and related papers are available at the following URL:
http://www.cs.toronto.edu/~bonner/papers.html#sequence-datalog

*Research partially supported by MURST and Consiglio Nazionale delle Ricerche (CNR).

†Research partially supported by an operating grant from the Natural Sciences and Engineering Research Council of Canada (NSERC).

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 [3] are the right solution. However, in other cases, such as genome databases [12] and text databases [14], there is still a need for more flexibility in data representation and manipulation. In these applications, much of the data has an inherently sequential structure. This has several implications for database management systems. First, a DBMS should provide a *sequence* type; that is, it should be able to manipulate sequences of unbounded length over a fixed alphabet. Second, the query languages provided to the user must have powerful primitives and operators for analyzing and restructuring sequences.

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 [22], we developed a logic called *Sequence Datalog* for querying sequence databases. Two safe subsets of the logic were defined, based on a new computational model called *Generalized Sequence Transducers* [20]. These machines are a simple yet powerful device for computing sequence mappings. In [22], 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 first result is a syntactically defined subset of Sequence Datalog that guarantees finiteness and safety. We call this subset *Weakly Constructive Sequence Datalog*. The second result is a characterization of its complexity and expressive power. We prove that it can express any sequence mapping with hyper-exponential time complexity. Thus, although Weakly Constructive Sequence Datalog is finite, it is still highly expressive.

1.1 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. This section provides an informal overview of the syntax and semantics of Sequence Datalog. A formal development can be found in [20].

To manipulate sequences, $SequenceDatalog$ 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. 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 term 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:
  - 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 $s[n_i]$.
  - 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, 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.

The semantics of terms is formalized in [20]. Constructive sequence terms have a semantics of concatenation; *e.g.*, $abc \bullet def = abcdef$. Indexed sequence terms have a semantics of subsequence extraction; *e.g.*, $abcdef[2{:}5] = bcde$ and $abcdef[4{:}end] = def$. However, there are some subtleties when the index terms take on "fringe" values, as illustrated by the following examples, where ϵ denotes the empty sequence:

$$
\begin{array}{ll}
uvwxy[3:6] \text{ is undefined} & pqrst[3:end+1] \text{ is undefined} \\
uvwxy[3:5] = wxy & pqrst[3:end] = rst \\
uvwxy[3:4] = wx & pqrst[3:end-1] = rs \\
uvwxy[3:3] = w & pqrst[3:end-2] = r \\
uvwxy[3:2] = \epsilon & pqrst[3:end-3] = \epsilon \\
uvwxy[3:1] \text{ is undefined} & pqrst[3:end-4] \text{ is undefined}
\end{array}
$$

Note that $s[n+1, n] = \epsilon$, while $s[n+i, n]$ is undefined for $i > 1$.

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 [19]. The head and body of a clause, γ, are denoted $\textsc{head}(\gamma)$ and $\textsc{body}(\gamma)$, 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 $p(X) \leftarrow q(X[1])$. Because of the active domain semantics, variables in Sequence Datalog clauses need not be guarded.

The semantics of rules is formalized in [20] in terms of least fixpoints. As in classical logic programming [28, 2, 19], each Sequence Datalog program, P, has an associated operator, T_P, that maps databases to databases. Each application of T_P may create new atoms, which may contain new sequences. The operator T_P is monotonic and continuous and has a least fixpoint [20]. This paper develops conditions under which the least fixpoint is finite. In such cases, we say that P has a *finite semantics*.

Example 1.1 The following rule extracts all prefixes of sequences in relation R:

$$prefix(X[1{:}N]) \quad \leftarrow \quad R(X).$$

For each sequence, X, in R, this rules 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. $\square$

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*.[1] Substitutions range over this domain when rules are evaluated.[2]

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. 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.

Example 1.2 The following rule constructs all possible concatenations of sequences in relation R:

$$answer(X \bullet Y) \quad \leftarrow \quad R(X),\ R(Y).$$

This rule takes any pair of sequences, X and Y, in relation R, concatenates them, and stores the result in $answer$, thereby adding new sequences to the extended active domain. The concatenated sequences (and their subsequences) form the extended active domain of the least fixpoint. □

Compared to Datalog with function symbols, or Prolog, two differences are apparent. The first is that Sequence Datalog has no uninterpreted function symbols, so it is not possible to build arbitrarily nested structures. On the other hand, Sequence Datalog has a richer syntax than the $[Head|Tail]$ list constructor of Prolog. This richer syntax is motivated by a natural distinction between two types of recursion, one safe and the other unsafe. 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 fact, it is bounded by the set of all subsequences of the active domain, which we call the extended active domain. Typically, languages for list manipulation do not discriminate between these two types of recursion. Sequence Datalog does: constructive recursion is performed using constructive terms, of the form $X \bullet Y$, while structural recursion is performed using indexed terms, of the form $X[n_1{:}n_2]$.

1.2 Examples

This section illustrates how sequences are manipulated in Sequence Datalog. 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 Section 4.

Example 1.3 **[Pattern Matching]** Suppose we are interested in sequences of the form $a^n b^n c^n$ in relation R. The query $answer(X)$ retrieves all such sequences, where the predicate $answer$ is defined by the following rules:

[1] 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.

[2] 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, $\frac{k(k+1)}{2} + 1$.

$$\begin{array}{rcl} answer(X) & \leftarrow & R(X),\ abc_n(X[1{:}N_1], X[N_1+1{:}N_2], X[N_2+1{:}end]). \\ abc_n(\epsilon,\epsilon,\epsilon) & \leftarrow & true. \\ abc_n(X,Y,Z) & \leftarrow & X[1]=a,\ Y[1]=b,\ Z[1]=c,\ abc_n(X[2{:}end], Y[2{:}end], Z[2{:}end]). \end{array}$$

The formula $answer(X)$ is true iff X is a sequence in R and it is possible to split X in three parts such that abc_n is true. Predicate abc_n is true for every triple of sequences of the form (a^n, b^n, c^n) in the extended active domain of the database. □

In Example 1.3, the semantics of the rulebase is trivially finite for every database, since the rules contain no constructive terms. Thus, any sequence in the least fixpoint is a subsequence of a sequence in the database. In contrast, the next two examples restructure the sequences in the database, producing new sequences longer than any in the database. Example 1.4 does this in a straightforward way, but has an infinite semantics. Example 1.5 solves the same problem, but with a finite semantics.

Example 1.4 **[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:

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

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. □

The program in Example 1.4 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. The table below shows how the inferred facts and the extended domain both grow during a bottom up computation of the least fixpoint. Each row in the table is the result of one additional application of the T operator. In each row, the inferred facts contain one more echo entry, and the extended active domain contains one more sequence, consisting entirely of a's. 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, $answer(aa, aaaa)$, which is computed during the fourth step. Thus, although the least fixpoint is infinite, the query answer is not. The next example expresses the query in such a way that both the answer and the least fixpoint are finite.

step	inferred facts	extended domain
0	$R(aa)$	ϵ, a, aa
1	$R(aa),\ echo(\epsilon, \epsilon)$	ϵ, a, aa
2	$R(aa),\ echo(\epsilon, \epsilon),\ echo(a, aa)$	ϵ, a, aa
3	$R(aa),\ echo(\epsilon, \epsilon),\ echo(a, aa),\ echo(aa, aaaa)$	$\epsilon, a, aa, aaa, aaaa$
4	$R(aa),\ echo(\epsilon, \epsilon),\ echo(a, aa),\ echo(aa, aaaa)$ $echo(aaa, aaaaaa),\ answer(aa, aaaa)$	$\epsilon, a, aa, aaa, aaaa$ $aaaaa, aaaaaa$
5	$R(aa),\ echo(\epsilon, \epsilon),\ echo(a, aa),\ echo(aa, aaaa)$ $echo(aaa, aaaaaa),\ echo(aaaa, aaaaaaaa),$ $answer(aa, aaaa)$	$\epsilon, a, aa, aaa, aaaa$ $aaaaa, aaaaaa$ $aaaaaaa, aaaaaaaa$
...	...	...

Example 1.5 **[Finite Semantics]** The following program is another way of expressing the query in Example 1.4.

$$
\begin{aligned}
answer(X, Y) &\leftarrow R(X),\ echo(X, Y). \\
echo(\epsilon, \epsilon) &\leftarrow true. \\
echo(X[1{:}N+1], Z \bullet X[N+1] \bullet X[N+1]) &\leftarrow R(X),\ echo(X[1{:}N], Z).
\end{aligned}
$$

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 Example 1.4, the program in Example 1.5 involves constructive recursion. However, in Example 1.5, 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 this way, structural recursion over the first argument controls and limits constructive recursion over the second argument. The bottom-up computation is similar to the one in the table above. The main difference is that no more *echo* facts are inferred after the fourth step, whence the least fixpoint is reached and the computation stops.

2 Preliminary Definitions

This section introduces technical definitions used in the rest of the paper, including *sequence database*, *sequence query* and *sequence function*.

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.

We say that a sequence, σ', of length k is a *contiguous subsequence* of sequence σ if for some integer, $i \geq 0$, $\sigma'(j) = \sigma(i + j)$ for $j = 1, \ldots, k$. Note that for each sequence of length k over Σ, there are at most $\frac{k(k+1)}{2} + 1$ different contiguous subsequences (including the empty sequence). For example, the contiguous subsequences of the sequence abc are: $\epsilon, a, b, c, ab, bc, abc$.

We now describe an extension of the relational model, in the spirit of [13, 15]. The model allows for tuples containing sequences of elements, instead of just constant symbols. 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.

A *sequence query* is a partial mapping from the set of databases over Σ to itself. Given a sequence query, Q, and a database, DB, the result of evaluating Q over DB is denoted $Q(\text{DB})$. Similarly, a *sequence function* [11] is a partial mapping from Σ^* to itself. A sequence function is *computable* if it is partial recursive. Usually, a notion of *genericity* [10] is introduced for queries. The notion can be extended to sequence queries in a natural way. We say that a sequence query Q is *computable* [10] if it is generic and 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. Expressibility results formulated in terms of sequence functions are especially meaningful for sequence query languages, since they provide a clear characterization of the power of the language to manipulate sequences. A sequence query language cannot express complex queries over sequence databases if it cannot express complex sequence functions. In short, function expressibility is necessary for query expressibility.

This paper addresses the complexity of sequence functions, and the data complexity [29] of sequence queries. 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. Given a sequence query, Q, a database, DB, and a suitable encoding of DB as a Turing machine tape, the *data complexity* of Q is the complexity of computing an encoding of $Q(\text{DB})$, measured with respect to the size of DB. A query language, L, is *complete* in the complexity class C if: (i) each query expressible in L has complexity in C; (ii) there is a query, Q, expressible in L such that computing $Q(\text{DB})$ is a complete problem for the complexity class C.

This paper also addresses the ability of query languages to express sequence functions. A query language, L, is said to *express* a complexity class, C, of sequence functions if: *(i)* each sequence function expressible in L has complexity in C and conversely, *(ii)* each sequence function with complexity in C can be expressed in L.

3 The Finiteness Problem

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 over every input database. However, as is typical of powerful logics, the *finiteness property* for Sequence Datalog programs is in general undecidable [22]. Thus, our aim is to develop *subsets* of the logic that are finite. In [22], we took what might be called a "semantic" approach, defining finite subsets in terms of abstract computing devices, called *generalized sequence transducers*. In this paper, we take a syntactic approach, defining finite subsets in terms of syntactic restrictions.

We first note that a necessary condition for infiniteness is the generation of sequences of unbounded length, as in Example 1.4. 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 introduce the notion of a *predicate dependency graph* of a Sequence Datalog program. 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.[3]

- 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.

Example 3.1 Consider the following two Sequence Datalog programs, P_1 and P_2:

$$P_1 : \left\{ \begin{array}{rcl} p(X) & \leftarrow & r(X,Y),\ q(Y[2{:}end])). \\ q(X) & \leftarrow & r(X,Y),\ p(Y[1{:}N]). \\ r(X \bullet Y, Y \bullet X) & \leftarrow & a(X,Y). \end{array} \right.$$

$$P_2 : \left\{ \begin{array}{rcl} p(X) & \leftarrow & q(X). \\ q(X) & \leftarrow & r(X). \\ r(X \bullet Y) & \leftarrow & p(X),\ t(X,Y). \end{array} \right.$$

Both programs are recursive, so their predicate dependency graphs both have cycles. The graph of P_2 has a constructive cycle, while the graph of P_1 does not. Thus, P_2 is recursive wrt constructions, while P_1 is not. □

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. Intuitively, this means we can find a *stratification* of the program with respect to constructive rules. In this case, the least fixpoint of the program is finite for every database, since there is no way to construct new sequences of unbounded length. We have following result about the resulting language, which we call *Stratified Sequence Datalog.*

Theorem 1 *Stratified Sequence Datalog is data complete for* PTIME.

Although Stratified Sequence Datalog is complete for PTIME, it has a very limited ability to restructure sequences. Many natural, simple and low-complexity restructurings—such as reversing a sequence or computing the complement—require constructive recursion and cannot be expressed in Stratified Sequence Datalog. To increase expressiveness while preserving finiteness, the next section develops syntactically restricted forms of recursion.

4 Domain Bounded Recursion

In some patterns of recursion, the length of newly constructed sequences is bounded above; that is, the recursive construction of new sequences proceeds up to a certain length and then stops, so that the semantics is finite. This section develops one of these patterns, which we call *domain bounded recursion.* In this form of recursion, 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.

[3] 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.

In previous examples of finite programs, the length of a computed sequence depended only on the length of a single sequence in the database (as when computing the echo of a sequence, in Example 1.5). In the next example, the length of a computed sequence depends on *two* database sequences.

Example 4.1 **[Shuffle]** We define a *shuffle* of two sequences, σ_1 and σ_2, to be any sequence obtained by interleaving the symbols of σ_1 with those of σ_2. For example, the shuffles of ab and 12 are $ab12, a1b2, a12b, 12ab, 1a2b, 1ab2$. The program below computes all the shuffles of pairs of sequences from the unary predicate R. It defines a predicate $shuffle(X, Y, Z)$ to be true iff Z is a shuffle of X and Y, where X and Y are sequences in R. The length of Z is thus the sum of the lengths of X and Y. For this reason, recursion stops after a finite amount of time.

$$
\begin{array}{lrcl}
\gamma_1: & shuffle(\epsilon, \epsilon, \epsilon) & \leftarrow & true. \\
\gamma_2: & shuffle(X[1{:}N+1], Y[1{:}M], X[N+1] \bullet Z) & \leftarrow & R(X),\ R(Y), \\
& & & shuffle(X[1{:}N], Y[1{:}M], Z). \\
\gamma_3: & shuffle(X[1{:}N], Y[1{:}M+1], Y[M+1] \bullet Z) & \leftarrow & R(X),\ R(Y), \\
& & & shuffle(X[1{:}N], Y[1{:}M], Z).
\end{array}
$$

In this program, pairs of sequences from relation R act as inputs to the last two rules, which define $shuffle(X, Y, Z)$. These rules recursively scan the input sequences X and Y while constructing an output sequence, Z. Starting with the empty sequence, these rules nondeterministically scan a symbol in X or a symbol in Y, and then append this symbol to the growing shuffle sequence, Z. $\square$

Before defining *Domain Bounded Recursion*, we need to develop some preliminary notions.

4.1 Reasoning about Length

To determine if a program is finite, we need to reason about what it does. In particular, 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. The idea is that terms can be compared on the basis of their possible instantiations. For instance, if X is a sequence variable, then we would say that the term $X \bullet abc$ is "longer" than the term X. This means that any instantiation of the one term is longer than the corresponding instantiation of the other term. This idea will allow us to reason about programs, and to develop conditions under which they are guaranteed to be finite.

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 character 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.*
- *$L(s_1 \bullet s_2)$ is the expression $L(s_1) + L(s_2)$.*
  e.g., $L(X \bullet atcg) = L_X + 4$
- *$L(s[n_1{:}n_2])$ is the expression $V(n_2, s) - V(n_1, s) + 1$.*

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., $V(end, X) = L_X$ $\quad$ $V(end, atcgatcg) = 8$
- *$V(n_1 \pm n_2, s)$ is the expression $V(n_1, s) \pm V(n_2, s)$.*
  e.g., $V(N+3, X) = V_N + 3$ $\quad$ $V(end - N, X) = L_X - V_N$

Here are some sequence terms and their symbolic lengths:

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

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. In fact, these are the only such situations. After all, if a symbol does not cancel itself out, then a symbolic length cannot be evaluated. *e.g.*, The expression $L_X - L_Y + 3$ cannot be evaluated. This idea gives us a mechanism with which to compare two symbolic lengths.

Definition 2 [Comparability] *Two sequence terms, s_1 and s_2, are* comparable *if the the symbolic expression $L(s_1) - L(s_2)$ can be evaluated, 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.*

Example 4.2 **[Comparing Sequence Terms]** 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

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 forced to range over sequences in relation q. However, in the rule $p(X) \leftarrow q(X[1:3])$, variable X is *not* constrained. To see this, 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. As another example, consider the following rule:

$$p(Y) \leftarrow q(X),\ Y = X[N:M]$$

In this case, X is a constrained variable, and so is Y. First, X is constrained to range over the sequences in relation q. Y is then constrained to range over subsequences of X.

These ideas motivate the following definition. In this definition, and throughout this paper, we use the notation (p, i) to refer to the i^{th} attribute (or argument) of predicate p.

Definition 3 [Constrained Variables] *Let γ be a rule, let S be a sequence variable, and let p be a predicate occurring in the body of γ. We say that S is* constrained *by attribute (p, i) in γ if at least one of the following holds:*

- *variable S is the i^{th} argument of some occurrence of predicate p in the body of γ;*
- *the body of γ contains an equality atom of the form $S = S_1[N_1{:}N_2]$ where S_1 is* constrained *by (p, i) in γ.*

The ideas of constrained variables and symbolic length lead to a simple formulation of structural recursion in Sequence Datalog. For example, consider the following rules, where q is a base predicate:

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

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 argument 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 the head and body of a rule using Definition 2. In this case, $X[1:N+1]$ is longer than $X[1:N]$. The following definition generalizes this idea.

Definition 4 [Variable Growth] *Suppose predicate p occurs in the head and body of a rule. 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.*

4.3 Domain Bounded Recursion: Definition

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 subset of Sequence Datalog that we call Weakly Constructive Sequence Datalog. This subset is defined in terms of four restrictions on Sequence Datalog programs. The first two restrictions are not strictly necessary, since they can be generalized without much difficulty. However, they simplify the development. The last two restrictions are the heart of domain bounded recursion. They are based on the ideas of constrained variables and symbolic length, developed above. The restrictions all apply to recursive constructive rules, that is, to recursive rules that have a constructive term in the head.

The first restriction we impose is that in recursive constructive rules, the head must have exactly one constructive argument. A constructive argument is an argument that contains a constructive term. Thus, the left-hand rule below is allowed, but the right-hand rule is not:

$$p(X,\, X \bullet Y) \;\leftarrow\; q(X),\; p(X, Y). \qquad\qquad p(X \bullet Y,\, X \bullet Y) \;\leftarrow\; q(X),\; p(X, Y).$$

The second 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 [5]. Actually, we require more than mere linearity, since we disallow *mutual* recursion through construction.[4] Thus, the predicate symbol in the head of a recursive constructive rule must also occur in the body of the rule. We call this *restricted linear recursion* through construction. For example, if q is a base predicate, then the following rule is restricted linear:

$$p(X,\, X \bullet Y) \;\leftarrow\; q(X),\; p(X, Y).$$

The rulebases in Examples 4.1 and 1.5, defining the predicates *shuffle* and *echo*, are both restricted linear. This property of a program can easily be checked in polynomial time (polynomial in the number of rules).

Note that 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). Moreover, abolishing mutual recursion through construction does not limit our expressive power, since mutual recursion can always be reduced to non-mutual recursion.

The third restriction we impose is based on an idea we call *constructive variables*. Consider a recursive constructive rule. As above, suppose the rule is restricted linear and has exactly one constructive term in the head. Suppose this term is in argument (p, k). Because the rule is restricted linear, predicate p also occurs in the body, exactly once. Argument (p, k) in the body contains at most one sequence variable, Z (since constructive terms are not allowed in rule bodies). If Z also occurs in argument (p, k) in the head, then we call Z the *constructive variable* of the rule, *e.g.* Variable Z is the constructive variable of predicate *shuffle* in Example 4.1. In other words, Z is passed from argument (p, k) in the body to argument (p, k) in the head, and in the process, some other sequence is appended to it. This property allows the rule to recursively construct new sequences, which can lead to the unbounded generation of new sequences. It is this kind of behavior that we want to limit and control using structural recursion.

Definition 5 [Domain Bounded Recursion] *A Sequence Datalog program is* domain bounded *if every recursive constructive rule in the program satisfies all the following conditions:*

[4] Technically, all constructive cycles in the attribute dependency graph of a program can involve only a single node. Thus, mutually recursive rules are allowed (as in Example 4.1), but mutually recursive predicates are not.

1. the head has exactly one constructive argument;

2. the rule is restricted linear;

3. except for the constructive variable (if any), every sequence variable is constrained;

4. some non-constructive argument in the head of the rule grows.

As mentioned earlier, only items *3* and *4* in this definition are strictly necessary. The others serve to simplify the development. (A more general treatment will soon be available in [21].)

Example 4.3 **[Continued Shuffling]** Consider the predicate *shuffle* in Example 4.1. There are two recursive constructive rules this program, γ_2 and γ_3. Both are restricted linear. and both have one constructive argument in the head. For both rules, Z is the constructive variable, and all other variables (X and Y) are constrained by predicate R. Finally, in the head of rule γ_2, the first argument grows; and in the head of of rule γ_3, the second argument grows. The program is therefore domain bounded. □

Finally, it is worth noting that although domain bounded programs are finite (as we show later), they can generate sequences of exponential length, as the following example shows.

Example 4.4 **[Long Sequences]** The following program is domain bounded:

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

Given a sequence, σ, of length n in predicate *input*, the predicate *doubling* computes a sequence of 1's of length 2^n by doubling the length of a unit sequence n times. □

5 Complexity and Expressibility

As described in Section 4.3, Weakly Constructive Sequence Datalog is the set of programs in Sequence Datalog that are domain bounded.

Theorem 2 *Every Weakly Constructive Sequence Datalog program is finite; that is, it has a finite semantics (least fixpoint) for every database.*

Although finite, Weakly Constructive Sequence Datalog is highly expressive. To see this, we first introduce the class of elementary sequence functions [23]. These are defined in terms of the *hyper-exponential functions*, $hyp_i(n)$, which are defined recursively as follows:

- $hyp_0(n) = n$
- $hyp_{i+1}(n) = 2^{hyp_i(n)}$ for $i \geq 1$

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 in $\cup_{i \geq 0}\mathrm{DTIME}[O(hyp_i(n))]$. The following theorem characterizes the expressibility of Weakly Constructive Sequence Datalog in terms of these functions.

Theorem 3 *Weakly Constructive Sequence Datalog expresses exactly the class of elementary sequence functions.*

Proof Sketch: See Appendix B. □

References

[1] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Comp. and Syst. Sci.*, 43(1), 1991.

[2] K.R. Apt and M.H. Van Emden. Contributions to the Theory of Logic Programming. *Journal of the ACM*, 29(3):841–862, 1982.

[3] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier and Z. Zdonik. The object oriented database manifesto. In *DOOD*, 1989.

[4] P. Atzeni, editor. *LOGIDATA+: Deductive Databases with Complex Objects. LNCS 701* - Springer-Verlag, 1993.

[5] F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *ACM SIGMOD*, 1986.

[6] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. In *ACM STOC*, 1992.

[7] A. J. Bonner. Hypothetical Datalog: complexity and expressibility. *Theoretical Computer Science*, 76, 1990.

[8] V. Breazu-Tannen, P. Buneman and S. Naqvi. Structural recursion as a query language. In *DBPL*, 1991.

[9] V. Breazu-Tannen, P. Buneman and L. Wong. Naturally embedded query languages. In *ICDT*, 1992.

[10] A. K. Chandra and D. Harel. Computable queries for relational databases. *Journal of Comp. and Syst. Sci.*, 21, 1980.

[11] L. S. Colby, E. L. Robertson, L. V. Saxton and D. Van Gucht. A query language for list based complex objects. In *ACM PODS*, 1994.

[12] *Communications of the ACM*. Special issue on the Human Genome project. vol.34, n.11. November 1991.

[13] S. Ginsburg and X. Wang. Pattern matching by RS-Operations: Towards a unified approach to querying sequenced data. In *ACM PODS*, 1992.

[14] G. H. Gonnet. Text dominated databases: Theory, practice and experience. Tutorial presented at *ACM PODS*, 1994.

[15] G. Grahne, M. Nykanen and E. Ukkonen. Reasoning about strings in databases. In *ACM PODS*, 1994.

[16] S. Grumbach and T. Milo. An algebra for POMSETS. In *ICDT*, 1995.

[17] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68, 1986.

[18] N. Immerman, S. Patnaik and D. Stemple. The expressiveness of a family of finite set languages. In *ACM PODS*, 1991.

[19] J. W. Lloyd. *Foundations of Logic Programming*, Second Edition. Springer–Verlag, 1987.

[20] G. Mecca and A. J. Bonner. In preparation. 1995.

[21] G. Mecca and A. J. Bonner. Domain Bounded Recursion. In preparation. 1995.

[22] G. Mecca and A. J. Bonner. Sequences, Datalog and Transducers. To appear in *ACM PODS*, 1995.

[23] C. Papadimitriou. *Computational Complexity*. Addison–Wesley, 1994.

[24] J. Richardson. Supporting lists in a data model (A timely approach). In *VLDB*, 1992.

[25] D. B. Searls. String Variable Grammars: a logic grammar formalism for DNA sequences. Tech. Report, University of Pennsylvania School of Medicine, 1993.

[26] D. Stott Parker, E. Simon and P. Valduriez. SVP – a model capturing sets, streams and parallelism. In *VLDB*, 1992.

[27] The Committee for Advanced DBMS functions. Third-generation database system manifesto. In *SIGMOD Record*, 19(3), 1990.

[28] M.H. Van Emden and R.A. Kowalski. The Semantics of Predicate Logic as a Programming Language, *Journal of the ACM*, 23(4):733–742, 1976.

[29] M. Vardi. The complexity of relational query languages. In *ACM STOC*, 1982.

A Appendix: Semantics of Sequence Datalog

This section sketches the formal semantics of Sequence Datalog programs. A full development can be found in [20]. The semantics is defined in terms of fixpoint theory, using a "T operator" in the Logic Programming tradition [2, 28, 19].

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 $1 \leq \theta(n_1) \leq \theta(n_2) + 1 \leq \text{LEN}(\theta(s)) + 1$. In particular, $\theta(s[n_1{:}n_2])$ is the contiguous subsequence of $\theta(s)$ extending from position $\theta(n_1)$ to position $\theta(n_2)$. Here, terms such as $s[n+1{:}n]$ are conveniently interpreted as the empty sequence, ϵ. 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 $\theta(s_2)$. Examples of interpreted sequence terms are listed in Section 1.1.

The semantics of rules is defined in terms of a least fixpoint theory. As in classical logic programming [19], each Sequence Datalog program, P, has an associated operator, T_P, that maps databases to databases. Each application of T_P may create new atoms, which may contain new sequences. The operator T_P is monotonic and continuous, and thus has a least fixpoint that can be computed in a bottom-up, iterative fashion [21]. Based on this fixpoint semantics, a model theory for Sequence Datalog can be developed in a straightforward way [21].

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 $\mathcal{D}^{ext}_{P,\text{DB}}$. 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 *least fixpoint* [19] of the *operator* T_P is computed in a bottom-up fashion, by starting at the database, DB, and applying the operator T_P repeatedly until a fixpoint is reached. At each step, and for each ground instantiation of each rule in P, if the premise of the rule has been inferred, then the head of the rule is added to the set of inferred facts. Because T_P is continuous, this process is complete [19]; that is, any atom in the least fixpoint of T_P will eventually be inferred.

- 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*. Also note that our semantics for sequence creation resembles the semantics of *value invention* in [1] in that sequences are added to the active domain as a side-effect of rule evaluation. In Sequence Datalog, however, the addition is purely declarative and deterministic, since the least fixpoint is unique.

B Appendix: Proof of Theorem 3

Theorem 3 *Weakly Constructive Sequence Datalog expresses exactly the class of elementary sequence functions.*

Proof Sketch: **[Upper Complexity Bound:]** We first prove the upper complexity bound; that is, any sequence function computable in Weakly Constructive Sequence Datalog is an elementary sequence function. To do this, we show that the least fixpoint of any weakly constructive program can be computed in hyper-exponential time with respect to the size of the input database.

Let us consider a weakly constructive program P and its predicate dependency graph PDG_P. From the definition of domain bounded predicate, we know that the only constructive cycles in PDG_P are loops due to domain bounded constructive rules. The strongly connected components in its predicate dependency graph admit a linearization. In fact, each component contains a set $\mathcal{N}_i$ of nodes such that, for each couple of nodes $n_1, n_2 \in \mathcal{N}_i$, there is a cycle connecting n_1 and n_2. Each node for which there is no cycle is to be considered as a singleton component. Thus, the relation *"there is an arc from component i to component j"* is finite and contains no cycles. Since the relation is well founded, we can assign, for some integer n, an integer $i \in \{1, \ldots, n\}$ to each component, in such a way that, if there is an arc from $\mathcal{N}_i$ to $\mathcal{N}_j$, then $i < j$. Let us call $\mathcal{N}_0, \ldots, \mathcal{N}_n$ the linearized components.

We will now prove that the final extended domain of the program has hyper-exponential size with respect to the initial database. This suffices to prove the claim, since, in this case, the least fixpoint of the program has hyper-exponential size and can be computed in hyper-exponential time.

In fact, consider now a predicate p such that the corresponding node is in $\mathcal{N}_0$. Since there is no clause in P such that p is the predicate symbol in the head, p must be a base predicate, and any sequence occurring in a tuple of p must belong to the input database.

Consider now a predicate q in $\mathcal{N}_1$. We can distinguish two cases: *(i)* q is not a domain bounded predicate, that is, $\mathcal{N}_1$ does not contain constructive cycles; *(ii)* q is domain bounded, that is, $\mathcal{N}_1$ is a singleton component.

Let us first consider case *(i)*. Suppose there is a constructive clause γ for q and $s_1 \bullet s_2 \bullet \ldots \bullet s_m$ is the constructive term in the head. In this case, for every substitution θ, the sequences $\theta(s_1), \ldots, \theta(s_m)$ are finite. In fact, in the body of γ only base predicate can occur and, by hypothesis, each variable in γ is guarded by one of them. This fact implies that in the least fixpoint of P over DB, q must have a finite extent. Moreover, any constructive clause for q can construct at most a polynomial number of new sequences with respect to the size of DB.

Consider now case *(ii)*, that is, the case in which $\mathcal{N}_1$ is a singleton component and q is a domain bounded predicate. Consider a fact of the form $q(\sigma_1, \sigma_2, \ldots, \sigma_n)$, and suppose that (q, n) is the constructive argument of q. In this case, since for each rule in the definition of q all the variables but the constructive variable associated with (q, n) must be constrained by some predicate belonging to a lower stratum, it is clear that there is a finite number of $n-1$ tuples of values $(\sigma_1, \ldots, \sigma_{n-1})$ that can be associated to $(q, 1), \ldots, (q, n-1)$. In fact, any sequence in such tuples must come from a base predicate in $\mathcal{N}_0$, and thus it belongs to the extended active domain of the input database. Moreover, we know that, for every rule γ in the definition of q, there is at least one attribute (q, i) of q other than the constructive argument that grows from body to head. This means that there exist a set $i_1, \ldots, i_j$ of attributes of q, namely the set of growing attributes, such that for any two facts of the form $q(\sigma_1, \sigma_2, \ldots, \sigma_n)$, $q(\sigma'_1, \sigma'_2, \ldots, \sigma'_n)$, for at least one i_k, the length of σ_{i_k} is different from the length of σ'_{i_k}. Thus, if l is the maximum length of a sequence in stratum 0, then there can be at most jl different facts for q. This suffices to prove that the extent of q is finite.

Let us now consider the length of the maximum sequence computed in q. Let γ be a simple linear constructive rule in the definition of q, and Z its constructive variable. The worst case is the one in which the constructive term in the head is of the form $Z \bullet Z \bullet \ldots \bullet Z$, that is, the constructive variable is appended m times to itself. In this case, at each step, the length of the computed sequences is multiplied m times by itself. Since there can be at most jl different istantiations of the predicate in the head, the maximum length must be $\mathcal{O}(m^{jl})$, that is, exponential with respect to the size of the sequences in $\mathcal{N}_0$. Since there are at most 2^l different sequences of length l over a fixed alphabet, the final size of the extended active domain after the evaluation of q will be hyper-exponential with respect to the initial database.

By iterating this argument for each stratum, the claim is proved. □

Proof Sketch: **[Expressive Completeness]:** We now prove that Weakly Constructive Sequence Datalog is expressively complete for the elementary sequence functions; that is, that any elementary sequence function can be expressed as a program in Weakly Constructive Sequence Datalog. More specifically, given a sequence function, f, and an input sequence, σ_{in}, we encode the input sequence as a unary tuple in the predicate *input*, and we compute the output sequence, $\sigma_{out} = f(\sigma_{in})$, in the 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 simulation is made of four main steps:

1. we first construct an 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 the Turing machine M_f runs in time $hyp_i(n)$, we need a counter from 1 to $hyp_i(n)$; we use a unary encoding of integers in order to count, so that the empty sequence will encode the number 0 and a sequence of length l will encode the number l;

2. once the counter has been built, we choose a suitable encoding of the input tape for the Turing machine, in order to start the simulation; [5]

3. then, the simulation is performed;

4. finally, after the simulation has stopped, we decode the result in predicate *output*.

The first step of the simulation consists in constructing the hyper-exponential counter. We will actually use a domain bounded predicate to build a sequence of length $hyp_i(n)$, and then use it to define predicates *first(counter)* and *next(counter,counter)*. This can be done with the follwing rules.

$$
\begin{array}{llcl}
\gamma_{1,1}: & hyp_1(\epsilon, 1) & \leftarrow & true. \\
\gamma_{2,2}: & hyp_1(X[1{:}N+1], Y \bullet Y) & \leftarrow & input(X), hyp_1(X[1{:}N], Y). \\
\gamma_{2,1}: & hyp_2(\epsilon, 1) & \leftarrow & true. \\
\gamma_{2,2}: & hyp_2(X[1{:}N+1], Y \bullet Y) & \leftarrow & hyp_1(X), hyp_2(X[1{:}N], Y). \\
\ldots & & & \\
\gamma_{i,1}: & hyp_i(\epsilon, 1) & \leftarrow & true. \\
\gamma_{i,2}: & hyp_i(X[1{:}N+1], Y \bullet Y) & \leftarrow & hyp_{i-1}(X), hyp_i(X[1{:}N], Y). \\
\gamma_1: & counter(X) & \leftarrow & input(X_0), hyp_1(X_0, X_1), \\
 & & & hyp_2(X_1, X_2), \ldots, hyp_i(X_{i-1}, X). \\
\gamma_2: & first(1) & \leftarrow & true. \\
\gamma_3: & next(X[1{:}N], X[1{:}N+1]) & \leftarrow & counter(X).
\end{array}
$$

[5] In the proof we suppose that the input sequence is nonempty. The proof can be easily generalized to empty input sequences.

Rules $\gamma_{1,1}$ to $\gamma_{i,2}$ simply generate a sequence of length $hyp_i(n)$, where n is the length of the input sequence. Then, rules $\gamma_1, \gamma_2, \gamma_3$ compute the *first* and the *next* predicate.

Now that we have a suitable counter, we can easily simulate the computation of M_f over σ_{in}. We use the counter to mark tape cells and keep track of time. In order to carry on the simulation, we will use two predicates:

- *cell(position, time, symbol)*, used to encode the content of the tape; the predicate tells for each position of the tape the value of the cell at different steps of the computation; the empty sequence will be used to mark blank cells;
- *head(position, time, status)*, used to keep track of the position of the head during the computation and the state of the machine.

The simulation is inizialized by encoding the input tape. In particular, cells from 1 to n will contain the input sequence, and every other cell the empty sequence.

$$\begin{array}{lllll}
\gamma_4: & encode(Pos_0, X[1], X[1]) & \leftarrow & input(X), first(Pos_0). \\
\gamma_5: & encode(Pos', X[N+1], X[1{:}N+1]) & \leftarrow & input(X), encode(Pos, _, X[1{:}N]), \\
& & & next(Pos, Pos'). \\
\gamma_6: & encode(Pos', \epsilon, stop) & \leftarrow & input(X), encode(Pos, _, X), \\
& & & next(Pos, Pos'). \\
\gamma_7: & encode(Pos', \epsilon, stop) & \leftarrow & encode(Pos, \epsilon, stop), \\
& & & next(Pos, Pos'). \\
\gamma_8: & cell(Pos, T_0, Symb) & \leftarrow & encode(Pos, Symb, _), first(T_0). \\
\gamma_9: & head(Pos_0, T_0, q_0) & \leftarrow & first(Pos_0), first(T_0).
\end{array}$$

Once the initial encoding of the tape has been computed, is easy to carry on the simulation. Different rules are necessary to encode different transitions of the Turing machines. More rules are needed to encode the frame axiom. We omit the simulation here. We only say that the following rule is used to detect when the simulation stops, that is, when the machine reachs the halting state q_h.

$$\gamma_{10}: \quad halt(T) \quad \leftarrow \quad head(\_, T, q_h).$$

After the simulation stops, we need to decode the output sequence. In particular, we need to concatenate together the symbols in the nonempty cells of the output tape, as follows:

$$\begin{array}{llll}
\gamma_{11}: & decode(\epsilon, \epsilon) & \leftarrow & true. \\
\gamma_{12}: & decode(X[1{:}N+1], C \bullet Y) & \leftarrow & counter(X), halt(T), \\
& & & cell(X[1{:}N+1], T, C), \\
& & & decode(X[1{:}N], Y). \\
\gamma_{13}: & output(X) & \leftarrow & counter(Y), decode(Y, X).
\end{array}$$

It is easy to show that the sequence in predicate *output* is the output of function f on input σ_{in}. $\square$