Sequence Datalog: Declarative String Manipulation in Databases *

Anthony Bonner[1]
bonner@db.toronto.edu

Giansalvatore Mecca[2]
mecca@dis.uniroma1.it

[1] Department of Computer Science – University of Toronto – Toronto, Ont. – Canada
[2] D.I.F.A. – Università della Basilicata – via della Tecnica, 3 – 85100 Potenza, Italy

Abstract. We investigate logic-based query languages for sequence databases, that is, databases in which strings of symbols over a fixed alphabet can occur. We discuss different approaches to querying strings, including Prolog and Datalog with function symbols, and argue that all of them have important limitations. We then present the semantics of *Sequence Datalog*, a logic for querying sequence databases, and show how this language can be used to perform structural recursion over sequences.

1 Introduction

In the last decade, new applications (*e.g.* CAD and CASE) have been a powerful driving force in the development of new database technology. New paradigms have arisen (*e.g.* object oriented databases [4]) that provide greater flexibility than the traditional relational model. However, in some application areas, such as *genome databases* [14] and *text databases* [16], there is still a need for greater flexibility, both in data representation and manipulation. These applications are characterized by a need to store, manipulate and transform strings.

Strings are a particularly interesting domain for query languages. In contrast to sets, computations over strings can easily become infinite, even when the underlying alphabet is finite. This is because symbols are allowed to repeat; so the number of possible strings 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 strings; on the other hand, the expressive power of the language should be carefully limited, to avoid infinite computations.

In [21], we introduced a logic called *Sequence Datalog* for querying sequence databases. *Sequence Datalog* is a Horn-like logic with special, interpreted function symbols that allow for *structural recursion* over strings. Sequence Datalog has both a clear declarative semantics and an operational semantics. The semantics are based on fixpoint theory, as in classical logic programming [19], and on a new notion of active domain, called the *extended active domain*, introduced to

* The first author was partially supported by an operating grant from the Natural Sciences and Engineering Research Council of Canada (NSERC). The second author was partially supported by MURST and Consiglio Nazionale delle Ricerche (CNR).

allow for a bottom-up evaluation of rules. The language has considerable power to manipulate strings. In fact, Sequence Datalog with stratified negation can express any computable query on sequence databases [22].

In this paper, we discuss in detail the semantics of Sequence Datalog, comparing and contrasting it to traditional logic-based languages. Whereas [21] introduced semantics of Sequence Datalog informally, this paper provides the formal development. The paper is organized as follows. Section 2 discusses other approaches to string manipulation based on logic. Section 3 introduces Sequence Datalog, discussing its syntax and semantics. Section 4 provides preliminary definitions. finally, Section 5, develops the formal semantics of the language.

2 Background

In recent years, several query languages for sequence databases have been proposed in the literature. On one hand, there are languages based on functional and algebraic programming [13, 18]. In this context, some sophisticated and expressive languages have been proposed, and great care has been devoted to the development of *tractable* languages, that is, languages whose complexity is in PTIME.

On the other hand, there are languages based on predicate logic [24, 15, 17]. In this context, powerful logics for expressing string transformations have been proposed, but many of them suffer from the difficult trade-off between expressiveness, finiteness and effective computability. In some cases, a great part of the expressive power of the language has to be sacrificed to achieve finiteness. In other cases, both expressiveness and finiteness are achieved, but at expense of an effective procedure for evaluating queries, *i.e.*, at the expense of an operational semantics.

In [15, 27], for example, an *extended relational model* is defined, where each relation is a set of tuples of sequences over a fixed alphabet. A *sequence logic* [15] is then developed based on the notion of *rs-operations*. Each *rs-operation* is either a *merger* or an *extractor*. Intuitively, given a set of patterns, an associated *merger* uses the patterns to "combine" a set of sequences. Likewise, an *extractor* "retrieves" subsequences of a given sequence. The authors introduce the notion of a *generic a-transducer* as the computational counterpart of rs-operations. Based on their sequence logic, two languages for the extended relational model are defined, called the *s-calculus* and the *s-algebra*. The s-calculus allows for unsafe queries, that is, queries whose answers are not computable. A safe subset of the language is then defined and proven equivalent to the s-algebra. Unfortunately, this safe sublanguage cannot express many queries for which the length of the result depends on the database. These include natural queries such as the reverse and complement of a sequence.

This problem is partially solved by the *alignment logic* of [17], an elegant and expressive first-order logic for a relational model with sequences. The computational counterpart of the logic is the class of *multi-tape, nondeterministic, two-way, finite-state automata*, which are used to accept or reject tuples of sequences. In its full version, alignment logic has the power of recursively enumerable sets [23]. A subset of the language called *right restricted formulas* is then developed. For this sublanguage, the safety problem is shown to be decidable, and complexity results related to the polynomial-time hierarchy are presented. Unfortunately, the non-deterministic nature of the computational model makes the evaluation of queries problematic. In particular, because of existential quantification over the infinite universe of sequences, it is not easy to determine the maximum length of the sequences in a query answer, even when the query is known to be safe.

There is also the traditional logic-programming approach [19]. For example, in *Prolog* [12], lists can be used to represent strings, and the language provides a natural "head-tail" constructor for manipulating them. It is also well known [19] that this language is computationally complete on lists. However, the top-down evaluation strategy of Prolog is widely regarded as inadequate for database applications, since it implies a procedural, "tuple-at-a-time" mode of computation; whereas, a non-procedural, set-oriented evaluation strategy is considered an essential feature of database query languages. To solve this problem, one might consider adopting *Datalog* [10] and its bottom-up, set-oriented evaluation strategy, and enriching it with *function symbols* to model lists. Unfortunately, coupling function symbols with bottom-up evaluation can lead to extremely inefficient computations, as the next example shows.

Example 1. **[Complement of Binary Strings]** Let `list` be a unary predicate whose extent specifies a set of lists. Each list represents a string of binary symbols, that is, a string over the alphabet $\{0, 1\}$. For each of these lists, we want to construct the complementary list, that is, the list obtained by substituting each 0 by a 1, and each 1 by a 0. This can easily be done in Datalog with function symbols, [3] as follows:

```
r1: answer(X)                      :- list(Y),
                                      complement(X, Y).

r2: complement([ ], [ ])           :- true.
r3: complement([H1|T1], [H2|T2])   :- compl(H1, H2),
                                      complement(T1, T2).

r4: compl(0, 1)                    :- true.
r5: compl(1, 0)                    :- true.
```

Rule $\mathtt{r_1}$ states that a list, `X`, is in the query answer if it is the complement of a list, `Y`, in the relation `list`. The predicate `complement` is written using standard tail-recursion over lists. Unfortunately, this Prolog-like program does not work as well with a bottom-up evaluator as with a top-down evaluator. To see this, consider that when evaluated bottom-up, rules $\mathtt{r_2}$ and $\mathtt{r_3}$ start with

[3] We use the binary function symbol `cons` to construct lists. We also adopt standard Prolog notation, so that `[H|T]` denotes the function term `cons(H,T)`, and `[ ]` is a constant symbol denoting the empty list.

the empty list and compute the complement of all possible lists of 0's and 1's, without bound.

This means two things. First, the program consisting of rules `r1` to `r5` has an infinite least fixpoint, even though the extent of predicate `answer` is finite. In addition, even if we prevented infinite computations (by, say, imposing an upper-bound on the length of the computed lists), the least fixpoint would still be very large and expensive to compute. In fact, if l_{max} is the maximum length of a list in `list`, then the program computes the complement of all possible lists of 0's and 1's up to length l_{max}. The program therefore computes an exponential number of unnecessary intermediate results, while the natural algorithm for complement only requires linear time. This problem arises because, unlike Prolog, rules in Datalog are evaluated "from body to head," instead of "from head to body."

In the following sections, we introduce *Sequence Datalog* [21], a query language designed to overcome these limitations, by supporting a safe and efficient form of structural recursion over strings.

3 Overview of Sequence Datalog

We use a data model that is a simple extension of the relational model [15, 17]. In this model, we fix a finite set of symbols, Σ, called *the alphabet*. Database tuples may contain finite sequences of symbols over this alphabet, instead of just single symbols.

Within this model, we develop and investigate different forms of structural recursion over sequences, especially forms that guarantee terminating computations. At the same time, we wish to construct new sequences and restructure old ones. To meet these two goals, programs in *Sequence Datalog* have two kinds of *interpreted* function term: *indexed terms*, which extract subsequences; and *constructive terms*, which concatenate sequences. An indexed term has the form $s[n_1{:}n_2]$, and is interpreted as the subsequence of sequence s from position n_1 to position n_2. Thus, $abcdef[2{:}5] = bcde$. A constructive term has the form $s_1 \bullet s_2$, and is interpreted as the concatenation of sequences s_1 and s_2. Thus, $abc \bullet def = abcdef$.

Example 2. **[Extracting Subsequences]** The following rule extracts all suffixes of all sequences in relation r:

$$suffix(X[N{:}end]) \leftarrow r(X).$$

This rule says that for each sequence X in relation r, and for each integer N, the subsequence of X extending from the N^{th} element to the last element is a suffix of X. By convention, we interpret the keyword *end* as the last position in a sequence.

The semantics of Sequence Datalog is based on a fixpoint theory, as in classical Logic Programming [19]. Thus, each Sequence Datalog program P has an associated "T-operator" that maps databases to databases. Each application of the operator may create new atomic formulas and new sequences. The operator is monotonic and continuous, and thus has a least fixpoint. If the least fixpoint is finite, then we say that the program has a *finite* semantics; otherwise, it has an *infinite* semantics. Because of the importance of finiteness and safety to database applications, the fixpoint semantics for Sequence Datalog involves certain subtleties that are absent in classical logic programming. Without this care, query answers can easily become infinite, as shown in Examples 1 and 3.

Example 3. **[Pattern Matching]** Suppose we are interested in sequences of the form $a^n b^n c^n$ in a unary relation r. In Sequence Datalog, the easiest way to retrieve all such sequences is with the following program, where ϵ is the empty sequence:

$$\begin{array}{ll}\gamma_1 : answer(X) & \leftarrow r(X), \\ & \quad abc_n(X[1{:}N_1],\ X[N_1+1{:}N_2],\ X[N_2+1{:}end]). \\ \gamma_2 : abc_n(\epsilon,\epsilon,\epsilon) & \leftarrow true. \\ \gamma_3 : abc_n(X,Y,Z) & \leftarrow X[1]=a,\ \ Y[1]=b,\ \ Z[1]=c, \\ & \quad abc_n(X[2{:}end],\ Y[2{:}end],\ Z[2{:}end]).\end{array}$$

For each sequence X in r, the formula $answer(X)$ is true if it is possible to split X into three parts such that abc_n is true. Predicate abc_n is true for triples of sequences of the form (a^n, b^n, c^n). Unfortunately, if we evaluate the program over the entire universe of sequences, Σ^*, then the least fixpoint is infinite. In fact, there is an infinite number of sequence triples (x, y, z) that satisfy rule γ_3, and the least fixpoint of the program contains an atom of the form $abc_n(x, y, z)$ for each of them.

The natural solution to these finiteness problems is to adopt an *active domain semantics*, so that rule evaluation is limited to sequences in the database. However, this must be done with some care if we are to support *structural recursion* over sequences, since in addition to the sequences in the database, we need access to their *subsequences*. For this purpose, 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*.[4] Substitutions range over this domain when rules are evaluated.[5]

Example 4. **[Pattern Matching (Continued)]** When evaluated over the extended active domain of the database, the rules in Example 3 have a finite fixpoint. In fact, rule γ_3 is evaluated only on triples of sequences belonging to the

[4] In this paper, we use only *contiguous subsequences*, that is, subsequences specified by a start and end position in some other sequence. Thus, bcd is a contiguous subsequence of $abcde$, but bd is not.

[5] Note that the size of the extended active 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 $\binom{k+1}{2} + 1 = \frac{k(k+1)}{2} + 1$, where the extra "1" is for the empty sequence.

extended active domain of the database, that is sequences in the database or their contiguous subsequences. This has two effects: it allows structural recursion, and at the same time, it prevents infiniteness.

It is worth noting that in our semantics, 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, developed in Section 5, provides a declarative semantics for this apparently procedural notion. In general, the extended active domain of the least fixpoint may be larger than the extended active domain of the database. In the database, the domain consists of the sequences in the database and all their subsequences. In 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 5. **[Concatenating Sequences]** The following rule concatenates all pairs of sequences from relation r:

$$answer(X \bullet Y) \leftarrow r(X),\ r(Y).$$

This rule takes any pair of sequences, X and Y, in relation r, concatenates them, and puts the result in relation *answer*, thereby adding new sequences to the extended active domain. The extended active domain of the least fixpoint is therefore larger than the extended active domain of the database.

Observe that our semantics for sequence creation resembles the semantics of *value invention* in [2] 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.

Compared to Datalog with function symbols, or Prolog, several differences are apparent. The first is that Sequence Datalog has no uninterpreted function symbols, so that 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 and 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, *i.e.*, by the extended active domain of the database. 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 $X[n_1{:}n_2]$. The next example illustrates both forms of recursion.

Example 6. **[Multiple Repeats]** Suppose we are interested in sequences of the form Y^n. The sequence $abcdabcdabcd$ has this form, where $Y = abcd$ and $n = 3$.[6] Here are two straightforward ways of expressing this idea in Sequence Datalog:

$$\begin{array}{ll} rep_1(X, X) & \leftarrow true. \\ rep_1(X, X[1{:}N])) & \leftarrow rep_1(X[N+1{:}end], X[1{:}N]). \end{array}$$

$$\begin{array}{ll} rep_2(X, X) & \leftarrow true. \\ rep_2(X \bullet Y, X) & \leftarrow rep_2(Y, X). \end{array}$$

The formulas $rep_1(X, Y)$ and $rep_2(X, Y)$ both mean that X has the form Y^n for some n. However, rep_2 has an infinite semantics, while rep_1 has a finite semantics. The rules for rep_1 do *not* create new sequences, since they do not use the concatenation operator, $\bullet$. Instead, these rules retrieve all sequences in the extended active domain that fit the pattern Y^n. To do this, they try to recursively "chop" an existing sequence, X, into identical pieces. We call this *structural recursion*. In contrast, the rules for rep_2 *do* create new sequences. They produce sequences of the form Y^n by recursively concatenating each sequence, Y, in the domain with itself. We call this *constructive recursion*. Structural recursion is always safe, leading to a finite least fixpoint. In this example, constructive recursion is unsafe, leading to an infinite least fixpoint.

The clear distinction between recursion over existing sequences and recursion through creation of new sequences allows the programmer to avoid infinite computations in many cases, as the next example shows.

Example 7. **[Complement]** Suppose we are given a set of binary sequences and want to find the complement of all sequences in r_1. There are two ways of expressing this transformation in Sequence Datalog.

(i) The first approach uses tail recursion, as in Example 1. However, in Sequence Datalog, we can prevent infiniteness by using safe, structural recursion over one argument of a predicate to control constructive recursion over the other argument(s).

$$\begin{array}{ll} \gamma_1 : answer(Y) & \leftarrow r_1(X),\ \ complement(X, Y). \\[4pt] \gamma_2 : complement(\epsilon, \epsilon) & \leftarrow true. \\ \gamma_3 : complement(X,\ Y \bullet Z) & \leftarrow compl(X[1],\ Y), \\ & \quad\ complement(X[2{:}end],\ Z). \\[4pt] \gamma_4 : compl(0, 1) & \leftarrow true. \\ \gamma_5 : compl(1, 0) & \leftarrow true. \end{array}$$

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. Thus, unlike Example 1, recursion stops after

[6] Repetitive patterns are of great importance in Molecular Biology [25].

a finite amount of time, since the active domain semantics prevents the generation of sequences of unbounded length.

(*ii*) Although they have a finite semantics, rules γ_1 to γ_5 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_1. This inefficiency can be avoided by expressing the query in another, more-efficient way, again using structural recursion.

$$\begin{array}{ll}
\gamma'_1 : answer(Y) & \leftarrow r_1(X), \quad complement(X, Y). \\
\gamma'_2 : complement(\epsilon, \epsilon) & \leftarrow true. \\
\gamma'_3 : complement(X[1{:}N+1],\ Z \bullet Y) & \leftarrow r_1(X), \quad compl(X[N+1],\ Y), \\
& \quad\ complement(X[1{:}N],\ Z).
\end{array}$$

In these rules, each sequence in relation r_1 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_1, rule evaluation requires only a linear number of database accesses (linear in the sum of the lengths of all sequences in r_1).

4 Preliminary Definitions

This section provides formal definitions for concepts used in the rest of the paper, including the concepts of *sequence database* and *sequence query*.

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, $\sigma_1, \sigma_2 \in \Sigma^*$. $\mathrm{LEN}(\sigma)$ 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 sequences of length 1.

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: $\epsilon, a, b, c, ab, bc, abc$.

The data model for Sequence Datalog is an extension of the relational model in the spirit of [15, 17]. The model allows for tuples containing sequences of symbols, instead of just constant symbols. Formally, a *relation* of arity k over Σ is a finite subset of the k-fold cartesian product of Σ^* with itself. A *sequence database* over Σ (or simply *database*, for short) 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 databases over Σ to the relations over Σ. As usual, we assume fixed input and answer schema for queries [11].

5 Sequence Datalog: Syntax and Semantics

This section introduces the syntax and semantics of the language. We first discuss the syntax informally, and then present two different semantics; the first is a fixpoint semantics, and the second is a model-theoretic semantics. We show that the two semantics are equivalent.

5.1 Syntax

As mentioned in Section 3, *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, then the term $X \bullet Y$ denotes the concatenation of sequences X and Y. Likewise, if I and J are integers, then the term $X[I : J]$ denotes the subsequence of sequence X extending from position I to position J.

To be more precise, the language of terms includes four countable, disjoint sets: the set of non-negative integers, $0, 1, 2 \ldots$; a set of constant symbols, 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*. Here, n_1 and n_2 are called the *indexes* of s. For convenience, sequence terms of the form $s[n_i{:}n_i]$ are 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 $ccgt \bullet S_1[1{:}end - 3] \bullet S_2$ are all sequence terms.

A term is *ground* if it contains no variables.

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 the formulas $s_1 = s_2$ and $s_1 \neq s_2$ are also atoms. From atoms, we build *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 clauses in which constructive terms (terms of the form $s_1 \bullet s_2$) may appear in clause heads, but not in clause bodies.

5.2 Substitutions

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.

Intuitively, $\theta(s[n_1{:}n_2])$ 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. For example,

s	$\theta(s)$
$uvwxy[3:6]$	undefined
$uvwxy[3:5]$	wxy
$uvwxy[3:4]$	wx
$uvwxy[3:3]$	w
$uvwxy[3:2]$	ϵ
$uvwxy[3:1]$	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 $\theta(s_2)$. The rest of this subsection makes these ideas precise.

Definition 1. (Substitutions Based on a Domain) Let $\mathcal{D}$ be a set of sequences and integers. A *substitution based on* $\mathcal{D}$ is a mapping that associates a sequence in $\mathcal{D}$ to every sequence variable, and an integer in $\mathcal{D}$ to every index variable.

Let θ be a substitution based on a domain. We extend θ from a total mapping on variables to a partial mapping on terms, as follows:

- if s is a sequence in Σ^* or an integer, then $\theta(s) = s$;
- in the context of the sequence term $S[n{:}end]$, we define $\theta(end) = \text{LEN}(\theta(S))$;
- if n_1 and n_2 are index terms, then $\theta(n_1 + n_2) = \theta(n_1) + \theta(n_2)$ and $\theta(n_1 - n_2) = \theta(n_1) - \theta(n_2)$;
- if s is a sequence term of the form $S[n_1{:}n_2]$, then $\theta(s)$ is defined if and only if $1 \le \theta(n_1) \le \theta(n_2) + 1 \le \text{LEN}(\theta(S)) + 1$, in which case
  - $\theta(s)$ is the contiguous subsequence of $\theta(S)$ extending from position $\theta(n_1)$ to $\theta(n_2)$ if $\theta(n_1) \le \theta(n_2)$
  - $\theta(s) = \epsilon$ (the empty sequence) if $\theta(n_1) = \theta(n_2) + 1$
- if s_1 and s_2 are sequence terms, then $\theta(s_1 \bullet s_2) = \theta(s_1)\theta(s_2)$ if both $\theta(s_1)$ and $\theta(s_2)$ are defined; otherwise, $\theta(s_1 \bullet s_2)$ is *undefined*.

Substitutions can be extended to atoms, literals and clauses in the usual way. We need only add that $\theta(p(X_1, \ldots, X_n))$ is defined if and only if each $\theta(X_i)$ is defined. Likewise, $\theta(q_0 \leftarrow q_1, \ldots, q_k)$ is defined if and only if each $\theta(q_i)$ is defined.

5.3 Fixpoint Semantics

The semantics of clauses is defined in terms of a least fixpoint theory. As in classical logic programming [19], each Sequence Datalog program P and database DB has an associated operator $T_{P,\mathrm{DB}}$ that maps Herbrand interpretations to Herbrand interpretations. Each application of $T_{P,\mathrm{DB}}$ may create new atoms, which may contain new sequences. As shown below, $T_{P,\mathrm{DB}}$ is monotonic and continuous, and thus has a least fixpoint that can be computed in a bottom-up, iterative fashion.

The following items summarize the main ideas in the fixpoint semantics:

- The *extended active domain* of database DB is the union of the following three sets: (i) the *active domain* of the database, that is, the set of sequences occurring in DB; (ii) all contiguous subsequences of sequences in the active domain; and (iii) the set of integers $\{0, 1, 2, \ldots, l_{max} + 1\}$, where l_{max} is the maximum length of a sequence in the active domain.
- The *least fixpoint* [19] of operator $T_{P,\mathrm{DB}}$ is computed in a bottom-up fashion, by starting at the empty interpretation and applying $T_{P,\mathrm{DB}}$ repeatedly until a fixpoint is reached. At each step, and for each ground instantiation of each clause in P, if each premise of the clause has been derived, then the head of the clause is added to the set of derived facts. Because $T_{P,\mathrm{DB}}$ is continuous, this process is complete [19]; that is, any atom in the least fixpoint will eventually be derived.
- At each step, if a derived 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 active domain (and hence 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*.

To develop the fixpoint semantics formally, we define the *Herbrand base* to be the set of ground atomic formulas built from predicate symbols in $\mathcal{P}$ and sequences in Σ^*. An *interpretation* of a program is a subset of the Herbrand base, and a *database* is a finite subset.

Definition 2. (Extensions) Given a set of sequences, dom, the *extension* of dom, written dom^{ext}, is the set of sequences and integers containing the following elements:

1. each element in dom;
2. for each sequence in dom, all its contiguous subsequences;
3. the set of integers $\{0, 1, 2, \ldots, l_{max} + 1\}$, where l_{max} is the maximum length of a sequence in dom.

Definition 3. (Extended Active Domain) The *active domain* of an interpretation I, denoted $\mathcal{D}_I$, is the set of sequences occurring in I. The *extended active domain* of I, denoted $\mathcal{D}_I^{ext}$, is the extension of $\mathcal{D}_I$.

Note that active domains contain only sequences, while extended active domains contain integers as well. The following lemma states some basic results about extended active domains.

Lemma 4. *If $I_1 \subseteq I_2$ are two interpretations, then $\mathcal{D}_{I_1}^{ext} \subseteq \mathcal{D}_{I_2}^{ext}$. If $\mathcal{I}$ is a set of interpretations, then their union, $\bigcup \mathcal{I}$, is also an interpretation, and $\mathcal{D}_{\bigcup \mathcal{I}}^{ext} = \bigcup_{I \in \mathcal{I}} \mathcal{D}_I^{ext}$.*

Like any power set, the set of interpretations forms a complete lattice under subset, union and intersection. We define a T-operator on this lattice, and show that it is monotonic and continuous. In defining the operator, each database atom is treated as a clause with an empty body.

Definition 5. (T-Operator) The operator $T_{P,\mathrm{DB}}$ associated with program P and database DB maps interpretations to interpretations. In particular, if I is an interpretation, then $T_{P,\mathrm{DB}}(I)$ is the following interpretation:

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

Lemma 6. (Monotonicity) *The operator $T_{P,\mathrm{DB}}$ is* monotonic*; i.e., if $I_1 \subseteq I_2$ are two interpretations, then $T_{P,\mathrm{DB}}(I_1) \subseteq T_{P,\mathrm{DB}}(I_2)$.*

Proof. Follows immediately from Definition 5 and Lemma 4.

Lemma 7. (Continuity) *The operator $T_{P,\mathrm{DB}}$ is* continuous; i.e., *if $I_1 \subseteq I_2 \subseteq I_3 \ldots$ is a (possibly infinite) increasing sequence of interpretations, then:*

$$T_{P,\mathrm{DB}}(\bigcup_i I_i) \subseteq \bigcup_i T_{P,\mathrm{DB}}(I_i)$$

Proof. Let $I = \bigcup_i I_i$ and let α be an atom in $T_{P,\mathrm{DB}}(I)$. We must show that α is also in $\bigcup_i T_{P,\mathrm{DB}}(I_i)$. By Definition 5, $\alpha = \theta(\text{HEAD}(\gamma))$ and $\theta(\text{BODY}(\gamma)) \subseteq I$, for some clause γ in $P \cup \mathrm{DB}$, and some substitution θ based on $\mathcal{D}_I^{ext}$. In addition, since the interpretations I_i are increasing, their extended active domains $\mathcal{D}_{I_i}^{ext}$ are also increasing, by Lemma 4. With this in mind, we proceed in three steps.

- $\theta(\text{BODY}(\gamma))$ is a finite set of ground atomic formulas, and $\theta(\text{BODY}(\gamma)) \subseteq \bigcup_i I_i$. Thus $\theta(\text{BODY}(\gamma)) \subseteq I_{j_1}$ for some j_1, since the I_i are increasing, and $\theta(\text{BODY}(\gamma))$ is finite.
- Let $\mathcal{V}$ be the set of variables in clause γ, and let $\theta(\mathcal{V})$ be the result of applying θ to each variable. Thus $\theta(\mathcal{V})$ is a finite subset of $\mathcal{D}_I^{ext}$, since θ is based on $\mathcal{D}_I^{ext}$. Thus $\theta(\mathcal{V}) \subseteq \bigcup_i \mathcal{D}_{I_i}^{ext}$ by Lemma 4. Thus $\theta(\mathcal{V}) \subseteq \mathcal{D}_{I_{j_2}}^{ext}$ for some j_2, since the $\mathcal{D}_{I_i}^{ext}$ are increasing, and $\theta(\mathcal{V})$ is finite.

– Let j be the larger of j_1 and j_2. Then $\theta(\text{BODY}(\gamma)) \subseteq I_j$ and $\theta(\mathcal{V}) \subseteq \mathcal{D}^{ext}_{I_j}$. Let θ' be any substitution based on $\mathcal{D}^{ext}_{I_j}$ that agrees with θ on the variables in $\mathcal{V}$. Then $\theta'(\gamma) = \theta(\gamma)$. Thus $\theta'(\text{BODY}(\gamma)) \subseteq I_j$. Thus $\theta'(\text{HEAD}(\gamma)) \in T_{P,\text{DB}}(I_j)$ by Definition 5. Thus $\alpha \in T_{P,\text{DB}}(I_j)$. Thus $\alpha \in \bigcup_i T_{P,\text{DB}}(I_i)$.

Given a program P and database DB, we define the following sequence of interpretations:

$$\begin{aligned}
T_{P,\text{DB}} \uparrow 0 &= \{\} \\
T_{P,\text{DB}} \uparrow (i+1) &= T_{P,\text{DB}}(T_{P,\text{DB}} \uparrow i), \quad \textit{for } i \geq 0 \\
T_{P,\text{DB}} \uparrow \omega &= \bigcup_{i=0}^{\infty} T_{P,\text{DB}} \uparrow i
\end{aligned}$$

Because $T_{P,\text{DB}}$ is monotonic and continuous, we can invoke the Knaster-Tarsky fixpoint theorem [19]. $T_{P,\text{DB}}$ thus has a least fixpoint, which is equal to $T_{P,\text{DB}} \uparrow \omega$. We say that $T_{P,\text{DB}} \uparrow \omega$ is the *fixpoint semantics* of program P over database DB. Intuitively, $T_{P,\text{DB}} \uparrow \omega$ is the set of facts implied by program P and database DB.

5.4 Model Theory

In this section we develop a model-theoretic semantics for Sequence Datalog programs, and show that it is equivalent to the fixpoint semantics developed above. Our notion of *model* is similar to the classical notion except that we restrict our attention to substitutions based on the extended active domain.

Definition 8. (Models) An interpretation I is a *model* of a clause γ, written $I \models \gamma$, if and only if the following is true for each substitution θ based on $\mathcal{D}^{ext}_I$ and defined at γ:

$$\textit{if} \quad \theta(\text{BODY}(\gamma)) \subseteq I \quad \textit{then} \quad \theta(\text{HEAD}(\gamma)) \in I$$

An interpretation is a model of a Sequence Datalog program P and a database DB if it is a model of each clause in $P \cup \text{DB}$.

Definition 9. (Entailment) Let P be a Sequence Datalog program, DB be a database, and α be a ground atomic formula. Then, P and DB entail α, written $P, \text{DB} \models \alpha$, if and only if every model of P and DB is also a model of α.

Lemma 10. *Let P be a Sequence Datalog program,* DB *be a database, and I be an interpretation. Then I is a model of P and* DB *if and only if it is a fixpoint of $T_{P,\text{DB}}$, i.e., if and only if $T_{P,\text{DB}}(I) \subseteq I$.*

Proof. To prove the *if* direction, suppose that $T_{P,\text{DB}}(I) \subseteq I$. Let γ be any clause in $P \cup \text{DB}$, and let θ be any substitution based on $\mathcal{D}_I^{ext}$ and defined at γ. If $\theta(\text{BODY}(\gamma)) \subseteq I$ then $\theta(\text{HEAD}(\gamma)) \in T_{P,\text{DB}}(I)$, by Definition 5. Thus $\theta(\text{HEAD}(\gamma)) \in I$, since $T_{P,\text{DB}}(I) \subseteq I$ by hypothesis. Thus, since θ is arbitrary, I is a model of γ. Thus, since γ is arbitrary, I is a model of $P \cup \text{DB}$.

To prove the *only if* direction, suppose that I is a model of $P \cup \text{DB}$. If $\alpha \in T_{P,\text{DB}}(I)$ then by Definition 5, $\alpha = \theta(\text{HEAD}(\gamma))$ for some clause γ in $P \cup \text{DB}$, and some substitution θ based on $\mathcal{D}_I^{ext}$. In addition $\theta(\text{BODY}(\gamma)) \subseteq I$. But I is a model of $P \cup \text{DB}$ by hypothesis, and thus of γ. Hence $\theta(\text{HEAD}(\gamma)) \in I$, so $\alpha \in I$. Thus, any atom in $T_{P,\text{DB}}(I)$ is also in I. Thus $T_{P,\text{DB}}(I) \subseteq I$.

The following corollaries are immediate consequences of Lemma 10 and the fixpoint theory of Section 5.3. They show that the model-theoretic semantics and the fixpoint semantics for Sequence Datalog are equivalent.

Corollary 11. *A Sequence Datalog program P and a database* DB *have a unique minimal model, and it is identical to the least fixpoint of the operator $T_{P,\text{DB}}$.*

Corollary 12. *Let P be a Sequence Datalog program,* DB *be a database, and α be a ground atomic formula. Then $P, \text{DB} \models \alpha$ if and only if $\alpha \in T_{P,\text{DB}} \uparrow \omega$.*

Acknowledgments The authors would like to thank Paolo Atzeni and Victor Vianu for many comments on the subject of the paper.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases.* Addison Wesley Publ. Co., Reading, Massachussetts, 1994.
2. S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computing and System Sciences*, 43(1):62–124, August 1991.
3. K. Apt and M. H. Van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.
4. M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and Z. Zdonik. The object-oriented database manifesto. In *First Intern. Conference on Deductive and Object Oriented Databases (DOOD'89), Kyoto, Japan*, pages 40–57, 1989.
5. P. Atzeni, editor. *LOGIDATA+: Deductive Databases with Complex Objects, Lecture Notes in Computer Science 701.* Springer-Verlag, 1993.
6. A. J. Bonner. Hypothetical Datalog: complexity and expressibility. *Theoretical Computer Science*, 76:3–51, 1990.
7. V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Third Intern. Workshop on Database Programming Languages (DBPL'91)*, pages 9–19, 1991.
8. V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In *Fourth International Conference on Data Base Theory, (ICDT'92), Lecture Notes in Computer Science*, pages 140–154, 1992.
9. S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Data Bases.* Springer-Verlag, 1989.

10. S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, March 1989.
11. A. K. Chandra and D. Harel. Computable queries for relational databases. *Journal of Computing and System Sciences*, 21:333–347, 1980.
12. W. Clocksin and C. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.
13. 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.
14. Communications of the ACM. Special issue on the Human Genome project. vol. 34(11), November 1991.
15. 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.
16. G. H. Gonnet. Text dominated databases: Theory, practice and experience. Tutorial presented at PODS, 1994.
17. 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.
18. 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*, pages 191–207, 1995.
19. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.
20. G. Mecca and A. J. Bonner. Finite query languages for sequence databases. In *Fifth Intern. Workshop on Database Programming Languages (DBPL'95), Gubbio, Italy*, 1995.
21. G. Mecca and A. J. Bonner. Sequences, Datalog and Transducers. In *Fourteenth ACM SIGMOD Intern. Symposium on Principles of Database Systems (PODS'95), San Jose, California*, pages 23–35, 1995.
22. G. Mecca and A. J. Bonner. Sequences, Datalog and Negation. In preparation, 1996.
23. C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
24. J. Richardson. Supporting lists in a data model (a timely approach). In *Eighteenth International Conference on Very Large Data Bases (VLDB'92), Vancouver, Canada*, pages 127–138, 1992.
25. D. B. Searls. String Variable Grammars: a logic grammar formalism for dna sequences. Technical 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 *Eighteenth International Conference on Very Large Data Bases (VLDB'92), Vancouver, Canada*, pages 115–126, 1992.
27. X. Wang. *Pattern matching by RS-operations: Towards a unified approach to querying sequence data*. PhD thesis, University of Southern California, 1992.

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