Sequences, Datalog and Transducers*

Giansalvatore Mecca[†]
Dipartimento di Informatica e Sistemistica
Universitá di Roma "La Sapienza" — Italy
`mecca@infokit.dis.uniroma1.it`

Anthony J. Bonner[‡]
Department of Computer Science
University of Toronto — Canada
`bonner@db.toronto.edu`

Abstract

This paper develops a query language for sequence databases, such as genome databases and text databases. The language, called *Sequence Datalog*, extends classical Datalog with interpreted function symbols for manipulating sequences. It has both a clear operational and declarative semantics, based on a new notion called the *extended active domain* of a database. The extended domain contains all the sequences in the database and all their *subsequences*. This idea leads to a clear distinction between *safe* and *unsafe* recursion over sequences: safe recursion stays inside the extended active domain, while unsafe recursion does not. By carefully limiting the amount of unsafe recursion, the paper develops a safe and expressive subset of Sequence Datalog. As part of the development, a new type of transducer is introduced, called a *generalized sequence transducer*. Unsafe recursion is allowed only within these generalized transducers. Generalized transducers extend ordinary transducers by allowing them to invoke other transducers as "subroutines." Generalized transducers can be implemented in Sequence Datalog in a straightforward way. Moreover, their introduction into the language leads to simple conditions that guarantee safety and finiteness. This paper develops two such conditions. The first condition expresses *exactly* the class of PTIME sequence functions; and the second expresses *exactly* the class of *elementary* sequence functions.

*Appears in *Proceedings of the Fourteenth ACM Symposium on Principles of Database Systems (PODS)*, pages 23–35. Symposium held May 22–25 1995, San Jose, Ca. Invited to a special issue of *Journal of Computer and System Sciences (JCSS)*.

†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

Sequences represent an important feature of *Next Generation Database Systems* [3, 27]. In recent years, new applications have arisen in which the storage and manipulation of sequences of unbounded length is a crucial feature. A prominent example is *genome databases*, in which long sequences representing genetic information are stored, and sophisticated pattern matching and restructuring facilities are needed [11]. These new applications have led to the introduction of sequence types in recent data models and query languages (e.g. [2, 4, 5, 28]). In many cases, however, queries over sequences are described only by means of a set of predefined, *ad hoc* operators, and are not investigated in a theoretical framework. In other cases, (e.g. [16, 25]) query languages concentrate on *pattern extraction* capabilities and do not consider *sequence restructurings*. Although pattern recognition is a fundamental feature of any language for querying sequences, sequence restructurings are equally important. For example, in genome databases, one needs to compute the reverse of a sequence, concatenate sequences together, splice out selected subsequences, etc.

Sequence data presents interesting challenges in the development of query languages. For instance, the query language should be *expressive* both in terms of pattern matching and sequence restructurings. At the same time, it should have a natural syntax and a clear semantics. Finally, it should be *safe*. Safety and finiteness of computations is a major concern when dealing with sequences, since by growing in length, sequences can easily become infinite, even when the underlying alphabet—or domain—is finite. This means that, unlike traditional query languages, sequence queries can end up in nonterminating computations. To achieve expressiveness, database researchers have developed sequence query languages based on abstract machines, such as *transducers* (finite automata that can generate strings as well as read them). Unfortunately, to achieve safety, they have had to impose restrictions that severely limit expressiveness (*e.g.*, [12, 30, 14]).

To address this problem, we propose a new logic called *Sequence Datalog* for reasoning about sequences. Sequence Datalog has both a clear declarative semantics and an operational semantics. The semantics are based on fixpoint theory, as in classical Logic Programming [20]. Thus, 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. T_P is monotonic and continuous and has a least fixpoint [22]. If the least fixpoint is finite, then we say that the program has a *finite* semantics; otherwise, it has an *infinite* semantics.

We show that Sequence Datalog can express all computable sequence functions. To achieve safety and finiteness, we introduce two devices. (*i*) We distinguish between structural recursion (which is safe) and constructive recursion (which is unsafe). We also develop a semantic counterpart to structural recursion, which we call the *extended active domain.* (*ii*) We allow the use of constructive recursion in a controlled fashion. In effect, we allow constructive recursion to simulate a new kind of machine that we call a *generalized transducer.* Intuitively, a generalized transducer is a transducer that can invoke other transducers as subroutines. Like transducers, generalized transducers always terminate. We can therefore use them as the basis for a safe subset of Sequence Datalog, which we call *strongly safe Transducer Datalog.* However, because generalized transducers are more powerful than ordinary transducers, this safe language retains considerable expressive power. For instance, with one level of transducer subroutine calls, it can express any mapping from sequences to sequences computable in PTIME. With two levels of subroutine calls, it can express any mapping from sequences to sequences in the elementary functions [23].

The semantics, the finiteness property and the expressive power represent the main contributions of this paper.

1.1 Background

The trade-off between *expressiveness*, *finiteness* and effective *computability* is typically a hard one. In many cases, powerful logics for expressing sequence transformations have been proposed, but a great part of the expressive power was sacrificed to achieve finiteness. In other cases, both expressiveness and finiteness were achieved, but at expense of an effective procedure for evaluating queries, *i.e.*, at the expense of an operational semantics.

In [12, 30], 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* [12] 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, the associated *merger* can be used to "merge" a set of sequences according to the patterns, whereas an *extractor* is used to "retrieve" subsequences of a given sequence. The authors introduce the notion of *generic a-transducer* as the computational counterpart of rs-operations. Based on the 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 semantics is not computable. A *safe* subset of the language is then defined and proven equivalent to the s-algebra. Intuitively, the *safe* version of the s-calculus is restricted to queries for which it is possible to bound, independently of the database, the length of sequences in the final result and in intermediate results [30]. This means that queries for which the length of the result depends on the database (such as the *reverse* of a sequence) cannot be expressed in the safe version of the language.

This problem is partially solved in the *alignment logic* of [14], 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 presented. For this subset, the safety problem is shown to be decidable, and some complexity results related to the polynomial-time hierarchy are presented. Unfortunately, the nondeterministic nature of the computational model makes the evaluation of queries problematic. In fact, existential quantification over the infinite universe of sequences, Σ^*, is allowed; thus, even when the query is known to be safe, it is not easy to determine the maximum length of the result.

Another interesting proposal for the use of logic in querying sequences is [24]. In this case, *temporal logic* is used as a base for a list query language. Conceptually, each successive position in a list is interpreted as a successive instance in time. This yields a query language in which temporal predicates can be used to investigate the properties of lists. However, temporal logic cannot be used to express some simple properties of sequences, such as whether a certain predicate is true at every even position of a list, or whether a sequence contains one or more copies of another sequence [32].

1.2 Overview of the Language

This paper builds on the works of [12, 14, 24] to propose a query language that is safe, expressive and has a clear declarative and operational semantics. This language, called *Sequence Datalog*, is a Horn-like logic with special, interpreted function symbols that allow for *structural recursion* over sequences.

Structural recursion has been extensively investigated in the context of *set* based query languages, and many interesting results have been presented (see for example [8, 19]). Unfortunately, as clearly argued in [10], such results cannot be extended to sequences. In fact, structural recursion over sequences has many peculiar aspects, the main one being that, while the number of possible *sets* over a finite alphabet is finite, the number of possible *sequences* is not. Because symbols in sequences can repeat, sequences can be of arbitrary length.

The goal of this paper is to 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 function term: *indexed terms* to extract subsequences, and *constructive terms* to concatenate sequences. An indexed term has the form $s[n_1{:}n_2]$, and is interpreted as a subsequence of s. A constructive term has the form $s_1 \bullet s_2$, and is interpreted as the concatenation of s_1 and s_2, which is a supersequence of both.

Example 1.1 **[Extracting Subsequences]**
The following rule extracts all prefixes of all 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 $N \geq 1$ and no longer than the length of X. □

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 may be larger than the extended active domain of the database. In the database, it consists of the sequences in the database and all their subsequences. In the least fixpoint, it consists of the sequences in the database and any new sequences created during rule evaluation, and all their subsequences.

Example 1.2 **[Concatenating Sequences]**
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 puts the result in relation *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 makes 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. 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 kinds of recursion.

Example 1.3 **[Multiple Repeats]**
Suppose we are interested in sequences of the form Y^n. The sequence *abcdabcdabcd* has this form, where $Y = abcd$ and $n = 3$.[3] Here are two straightforward ways of expressing this idea in *Sequence Datalog*:

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

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

[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 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 $\sum_{i=0}^{2} \binom{k}{i}$, that is, $\frac{k(k+1)}{2} + 1$.

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

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. They try to recursively "chop" an existing sequence 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 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. □

1.3 Safety and Transducers

We say that a program is *finite* if it has a finite semantics (*i.e.* a finite least fixpoint) for every database. Given a Sequence Datalog program, determining whether it has a finite semantics is an undecidable problem. The challenge, then, is to develop subsets of the logic that are both finite and expressive. As shown in Example 1.3, constructive recursion is the basic source of non-finiteness. Thus, a simple way to guarantee finiteness is to forbid constructive recursion. However, this approach greatly reduces the expressiveness of the language, since it almost eliminates the ability to restructure sequences. *e.g*, It would not be possible to compute the reverse or the complement of a sequence. Instead, the approach taken in this paper is to allow constructive recursion only in the context of a precise (and novel) computational model.

The model we develop is called *generalized sequence transducers* (or *generalized transducers*, for short), which are a simple extension of ordinary transducers. Typically [12, 26, 13], a *transducer* is defined as a machine with n input lines, one output line and an internal state. The machine sequentially "reads" the input strings, and progressively "computes" the output. At each step of the computation, the transducer reads the left-most symbol on each input tape, and "consumes" one of them. The transducer then changes state and appends a symbol to the output. The computation stops when all the input sequences have been consumed. Termination is therefore guaranteed for finite-length inputs. Unfortunately, ordinary transducers have very low complexity, essentially linear time. This means that they cannot perform complex operations, such as detecting context-free or context-sensitive languages, as it is often needed in genome databases [25].

We generalize this machine model by allowing one transducer to call other transducers, in the style of subroutines. At each step, a generalized transducer can *append* a symbol to its output or it can *transform* its output by invoking a sub-transducer. Like transducers, generalized transducers consume one input symbol at each step, and are thus guaranteed to terminate on finite inputs. In this way, we increase expressibility while preserving termination. We shall see that the depth of subroutine calls within a generalized transducer is a key determinate of their computational complexity.

Unlike other list-based query languages, *Sequence Datalog* provides a natural framework for implementing generalized transducers. The consumption of input symbols is easily implemented as structural recursion; appending symbols to output tapes is easily implemented as constructive recursion; and sub-transducers are easily implemented as subroutines, in the logic-programming sense. Moreover, by introducing transducers into the logic, we can develop simple syntactic restrictions that guarantee finiteness. These restrictions allow constructive recursion only "inside" transducers. Using this idea, we develop safe and finite subsets of *Sequence Datalog*, and establish their complexity and expressibility.

1.4 Expressibility

Two different kinds of transformation can be used to measure the expressive power of a sequence query language. The first kind is a *sequence query*, which is a straightforward generalization of relational query, *i.e.*, a mapping from sequences databases to sequence relations. The second kind of transformation is a *sequence function*, which has no counterpart in traditional database languages. Following [10], we define a *sequence function* to be a mapping that takes a sequence as input and returns a sequence as output. Sequence functions can be thought of as queries from a database, $\{input(\sigma_{in})\}$, containing a single sequence tuple, to a relation, $\{output(\sigma_{out})\}$, 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.

In this paper, we characterize the expressive power of subsets of Sequence Datalog in terms of sequence functions. We prove expressibility results for both the class of PTIME sequence functions and the class of *elementary* sequence functions [23]. PTIME expressibility results were first reported in [10, 15] with respect to list-based databases of complex-objects. Here, we extend those results to any hyper-exponential time function. In [17], expressibility results for intermediate types were proved in terms of hyper-exponential time. We extend these results to sequences. In [22], we extend our results about *function expressibility* to results about *query expressibility* by introducing *negation* into the language.

2 Preliminary Definitions

This section provides 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 [12, 14]. 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 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, $Q(\text{DB})$ is the result of evaluating Q over DB. Similarly, a *sequence function* [10] is a partial mapping from Σ^* to itself. A sequence function f is *computable* if it is partial recursive. Usually, a notion of *genericity* [9] 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* [9] if it is generic and partial recursive.

In this paper, we address 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.

A language L is said to *express* a class of sequence functions C 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. Likewise, we say that a language L *expresses* a class of queries QC if (i) each sequence query expressible in L has complexity in C, and conversely, (ii) each sequence query with complexity in C can be expressed in L. Note that a language, L, expresses a class of sequence queries QC only if it expresses the corresponding class of sequence functions C; that is, function expressibility is a necessary condition for query expressibility.

3 Sequence Datalog

This section introduces *Sequence Datalog*, a query language for the extended relational model defined in the previous section. Sequence Datalog extends Datalog to sequence databases. Its syntax is that of classical Datalog enriched with two types of *complex term*, called *indexed* and *constructive*. *Indexed sequence terms* have the form $s[n_1{:}n_2]$, and they "extract" contiguous subsequences from a given sequence. *e.g.*, $abcdef[2{:}4] = bcd$. *Constructive sequence terms* have the form $s_1 \bullet s_2$, and they concatenate sequences. *e.g.*, $abc \bullet def = abcdef$. Before defining the syntax and semantics of the language, we give examples to show how sequences are manipulated in the language.

Example 3.1 **[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, where ϵ is the empty sequence:

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

The formula $answer(X)$ is true for a sequence X in r if 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. □

Example 3.2 **[Sequence Restructuring]**
Suppose r is a unary relation containing a set of binary sequences. We want to generate the so-called reverse complement of every sequence in r. *e.g.*, The reverse complement of 110000 is 111100. The query $answer(Y)$ generates these sequences, where the predicate $answer$ is defined by the following rules:

$$answer(Y) \leftarrow r(X),\ rev\_comp(X,Y).$$

$$rev\_comp(\epsilon,\epsilon) \leftarrow true.$$
$$rev\_comp(X[1{:}N+1], Z \bullet Y) \leftarrow r(X),\ rev\_comp(X[1{:}N],Y),\ comp(X[N+1],Z).$$

$$comp(0,1) \leftarrow true$$
$$comp(1,0) \leftarrow true$$

In this program, the sequences in r act as input for the third rule, which defines the predicate $rev\_comp(X,Y)$. This rule recursively scans each input sequence, X, while constructing an output sequence, Y. For each bit in the input sequence, a complementary bit is appended to the other end of the output sequence. The rule generates the complement of each prefix of each sequence in r. The first rule then retrieves the complements of only the sequences in r. The predicate *comp* specifies that 0 and 1 are complementary. □

The rest of this section presents the syntax and semantics of Sequence Datalog. Due to space limitations, only informal definitions are possible. A complete, formal development can be found in [22].

3.1 Syntax

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 X and Y. Likewise, if I and J are integers, then 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 $ccgt \bullet S_1[1{:}end-3] \bullet S_2$ are all sequence terms.

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 [20]. 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 (terms of the form $s_1 \bullet s_2$) 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 to be guarded.

3.2 Semantics

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 $1 \leq \theta(n_1) \leq \theta(n_2) + 1 \leq \textsc{len}(\theta(s)) + 1$. 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. 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 semantics of rules is defined in terms of a least fixpoint theory. As in classical logic programming [20], 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 [22]. Based on this fixpoint semantics, a model theory for Sequence Datalog can be developed in a straightforward way [22].

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

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

- The *least fixpoint* [20] 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 [20]; 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.

3.3 Expressive Power and Finiteness

This section establishes basic results about the expressive power and finiteness of Sequence Datalog programs.

Because Sequence Datalog can construct sequences of arbitrary length, it is not hard to simulate counters, which in turn can be used to simulate arbitrary Turing machines. This is the basis for our first result.

Theorem 1 *Sequence Datalog expresses exactly the class of computable sequence functions.*

Note that although Sequence Datalog is function complete, it is not query complete, since it only expresses *monotonic* queries. Functional completeness is, however, a necessary condition for query completeness. In [22], we show that *Sequence Datalog with negation* can express any computable query.

Theorem 1 implies that the property of *finiteness* is undecidable.

Theorem 2 *The finiteness property is undecidable for Sequence Datalog programs.*

Subsequent sections define subsets of Sequence Datalog that are finite. For now, we observe that the simplest way to enforce finiteness in Sequence Datalog is to forbid the construction of new sequences. The resulting language, in which constructive terms of the form $s_1 \bullet s_2$ cannot be used, is called *Non-constructive* Sequence Datalog. In this language, we cannot express queries beyond PTIME, since for each non-constructive program P, and each database DB, the extended active domain is fixed and does not grow during the computation. We have the following theorem.

Theorem 3 *The data complexity of Non-constructive Sequence Datalog is complete for* PTIME.

Although Non-constructive Sequence Datalog has low complexity, it expresses a wide range of *pattern matching* queries. This is evident in Example 3.1, in which a non context-free language is recognized.

4 Generalized Sequence Transducers

Because they cannot construct new sequences, nonconstructive programs have weak data-restructuring capabilities. To increase these capabilities—while preserving finiteness—we use an abstract computational device called a *generalized sequence transducer*. Transducers are low-complexity devices that take sequences as input and produce new sequences as output. They are therefore natural devices for restructuring sequences (see also [12, 14, 13, 26]). Moreover, we can exploit the low complexity of transducers to guarantee finiteness.

A *transducer* is usually defined as a machine with n input lines, one output line, and an internal state. The machine sequentially "reads" the input strings, and progressively "computes" the output. At each step of the computation, the current input symbols are read and, based on the current state, a sequence is appended to the output and a new state is chosen. The computation is guaranteed to terminate since at each step at least one input symbol is "consumed". Transducers therefore have very low complexity—essentially linear time. There are therefore many sequence restructurings that they cannot perform. To allow for more complex restructurings, we introduce a new computational device, which we call a *generalized sequence transducer*.

Intuitively, a generalized transducer is a transducer that can invoke another transducer. At each step of a computation, a generalized transducer must consume an input symbol, and may append a new symbol to its output, just like an ordinary transducer. In addition, at each step, a generalized transducer may transform its entire output sequence by sending it to another transducer, which we call a *subtransducer*. This process of invoking subtransducers may continue to many levels. Thus a subtransducer may transform its own output by invoking a subsubtransducer, etc. Subtransducers are analogous to subroutine calls in programming languages, or, in some way, to oracle Turing machines of low complexity.

We shall actually define a hierarchy of transducers. $\mathcal{T}^k$ represents the set of generalized transducers that invoke subtransducers to a maximum depth of $k-1$. $\mathcal{T}^1$ thus represents the set of ordinary transducers, which do not invoke any subtransducers. We define $\mathcal{T}^{k+1}$ in terms of $\mathcal{T}^k$, where $\mathcal{T}^0$ is the empty set. For convenience, we shall often refer to members of $\mathcal{T}^1$ as *base transducers*, and to any generalized sequence transducer simply as a transducer.

To formally define the notion of generalized sequence transducers, we use three special symbols, $\lhd$, $\rightarrow$ and $-$. $\lhd$ is an end-of-tape marker, and is the last symbol (rightmost) of every input tape. $\rightarrow$ and $-$ are commands for the input tape heads: $\rightarrow$ tells a tape head to move one symbol to the right (*i.e.*, to consume an input symbol); and $-$ tells a tape head to stay where it is. Although the following definition is for deterministic transducers, it can easily be generalized to allow nondeterministic computations. As such, it generalizes many of the transducer models proposed in the literature (see for example [12, 26]).

Definition 1 [Generalized Transducers] *A generalized n-ary sequence transducer of order $k > 0$ is a 4-tuple $\langle K, q_0, \Sigma, \delta\rangle$ where:*

1. *K is a finite set of elements, called* states;
2. *$q_0 \in K$ is a distinguished state, called the* initial state.
3. *Σ is a finite set of symbols, not including the special symbols, called the* alphabet;
4. *δ is a partial mapping from $K \times \{\Sigma \cup \{\lhd\}\}^n$ to $K \times \{-, \rightarrow\}^n \times \{\Sigma \cup \{\epsilon\} \cup \mathcal{T}^{k-1}\}$, called the* transition function.
5. *For each transition, $\delta(q, a_1, \ldots, a_n)$ of the form $\langle q', c_1, \ldots, c_n, \text{OUT}\rangle$, we impose three restrictions: (i) at least one of the c_i must be $\rightarrow$, (ii) if $a_i = \lhd$ then $c_i = -$, and (iii) if $\text{OUT} \in \mathcal{T}^{k-1}$ then it must be an $n+1$-ary transducer.*

$\mathcal{T}^k$ consists of all generalized transducers of order at most k, for $k > 0$; $\mathcal{T}^0 = \{\}$.

In this definition, the restrictions in item *5* have a simple interpretation. Restriction (*i*) says that at least one input symbol must be consumed at each step of a computation (c_i is a command to input head i). Restriction (*ii*) says that an input head cannot move past the end of its tape (a_i is the symbol below input head i). Restriction (*iii*) says that a subtransducer must have one more input than its calling transducer.

The computation of a generalized sequence transducer over input strings $\langle\sigma_1, \ldots, \sigma_n\rangle$ proceeds as follows. To start, the machine is in its initial state, q_0, each input head scans the first (*i.e.*, leftmost) symbol of its tape, and the output tape is empty. At each point of the computation, the internal state and the tape symbols below the input heads determine what transition to perform. If the internal state is q and the tape symbols are $a_1 \ldots a_n$, then the transition is $\delta(q, a_1, \ldots, a_n) = \langle q', c_1, \ldots, c_n, \text{OUT}\rangle$. This transition is carried out as follows:

- If OUT is a symbol in Σ, then it is appended to the output sequence; if $\text{OUT} = \epsilon$, then the output is unchanged.
- If OUT represents a call to a transducer $T \in \mathcal{T}^{k-1}$ then T is invoked as a subtransducer. In this case, the transducer suspends its computation, and the subtransducer begins. The subtransducer has $n+1$ inputs: a copy of each input of the calling transducer, plus a copy of its current output. The output of the subtransducer is initialized to the empty sequence. When the subtransducer is finished computing, its output is copied to (overwrites) the output tape of the calling transducer.
- The transducer "consumes" some input by moving at least one tape head one symbol to the right.
- The transducer enters the next state, q', and resumes its computation.

The transducer stops when every input tape has been completely consumed, that is, when every input head reads the symbol $\lhd$. Since transducers (and subtransducers) must consume all their input, the computation of every generalized transducer is guaranteed to terminate.

Finally, note that an n-ary transducer defines a *sequence mapping*, $T : (\Sigma^*)^n \rightarrow \Sigma^*$, where $T(\sigma_1, \ldots, \sigma_n)$ is the output of the transducer on inputs $\sigma_1, \ldots, \sigma_n$. Generalized transducers express a much wider class of mappings than ordinary transducers. For instance, they can compute outputs whose length is polynomial and even exponential in the input lengths, as illustrated by the following example.

Example 4.1 **[Quadratic Output]**
Let T_{square} be a generalized transducer with one input. At each step of its computation, T_{square} calls a subtransducer, T_{append}, with two inputs. One input to T_{append} is the input to T_{square}, and the other is the output of T_{square}. T_{append} simply appends its two inputs. The output of T_{append} then becomes the output for T_{square}, overwriting the old output.

Let σ_{in} be the input to T_{square}. If σ_{in} has length n, then at the end of its computation, the output of T_{square} will have length n^2, obtained by concatenating σ_{in} with itself n times. To see this, note that T_{square} calls T_{append} exactly n times, once for each symbol in σ_{in}:

- At time 1, the two inputs to T_{append} are σ_{in} and the empty sequence, ϵ (since the output of T_{square} is initially empty). Thus, the output of T_{append} at this step is σ_{in}, which becomes the new output for T_{square}.

- At time 2, the two inputs to T_{append} both contain a copy of σ_{in}. Thus, the output of T_{append} at this step is the concatenation of σ_{in} with itself, which is a sequence of length $2n$. This sequence then becomes the new output for T_{square}.

- In general, at time i, for $1 \leq i \leq n$, the two inputs to T_{append} are σ_{in} and the sequence obtained by concatenating σ_{in} with itself $i-1$ times. Thus, the output of T_{append} at this step is the sequence obtained by concatenating σ_{in} with itself i times. This sequence then becomes the new output for T_{square}.

Thus, after n steps, the output of T_{square} is a sequence of length n^2, namely the n-fold concatenation of σ_{in} with itself. □

4.1 Transducer Networks

Transducers can be combined to form *networks*, in which the output of one transducer is an input to another transducer. Since we are interested in finite computations, we only consider *acyclic networks*, in which the output of a transducer is never fed back to its own input. For each transducer network, some transducer inputs are designated as network inputs, and some transducer outputs are designated as network outputs. Each network then computes a *mapping* from sequence tuples to sequence tuples. When the network has only one output, the network computes a sequence function.

This section presents basic results about the complexity of generalized transducer networks. A more detailed analysis is beyond the scope of this paper and will be reported elsewhere [21].

The computational complexity of the sequence function computed by a transducer network depends on two parameters. The first is the *diameter* of the network, *i.e.*, the maximum length of a path in the network. The diameter determines the maximum number of transformations that a sequence will undergo in traveling from the input to the output of a network. The second parameter is the *order* of the network. This is maximum order of any transducer in the network. If the set of transducers in the network is a subset of $\mathcal{T}^k$, then the order of the network is at most k. Intuitively, the order of a network is the maximum depth of subtransducer nesting.

We now establish a basic result about the complexity of acyclic networks. This result involves the elementary sequence functions [23], which are defined in terms of the *hyper-exponential functions*, $hyp_i(n)$. These latter functions are defined recursively as follows:

- $hyp_0(n) = n$

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

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 $\cup_{i \geq 0}\text{DTIME}[O(hyp_i(n))]$.

The theorems below characterize the complexity and expressibility of two classes of transducer networks, those of order 2 and 3, respectively. Higher order networks will be investigated in [21]. Our first results concern the output size of transducer networks.

Theorem 4 *Consider an acyclic network of transducers with m inputs and 1 output.*

- *If the network has order 2, then the length of the output is (at most) polynomial in the sum of input lengths.*

- *If the network has order 3, then the length of the output is (at most) hyper-exponential in the sum of input lengths.*

This theorem allows us to formally prove expressibility results for acyclic networks of transducers. We again distinguish two cases, based on the order of a transducer network. The first is for transducer networks of order 2. Recall that each transducer in such a network is either a base transducer (*i.e.*, in $\mathcal{T}^1$) or it invokes a base transducer as a subroutine. Building on Theorem 4, we get the following result.

Theorem 5 *Acyclic transducer networks of order 2 express exactly the class of sequence functions computable in* PTIME.

This theorem provides a characterization of PTIME in terms of transducer networks. Other PTIME characterizations have been presented in the literature (*e.g.*, [10, 6, 15]); but transducer networks admit a fine-grained characterization in terms of network diameter [21].

When the order of transducer networks increases from 2 to 3, the complexity of the resulting sequence functions increases dramatically, as the following theorem shows.

Theorem 6 *Acyclic transducer networks of order 3 express exactly the class of elementary sequence functions.*

There is a close relationship between the diameter of transducer networks and levels in the hyper-exponential hierarchy. For instance, any sequence function in EXPTIME can be expressed by a single transducer of order 3. These ideas are developed in [21].

5 Sequence Datalog with Transducers

This section develops a new language by introducing generalized transducers into Sequence Datalog. This new language forms the basis of a safe and finite query language for sequence databases in the next section.

To invoke transducer computations from within a logical rule, we augment the syntax of Sequence Datalog with special, interpreted function symbols, one for each generalized sequence transducer. From these function symbols, we build function terms of the form $T(s_1, \ldots, s_n)$, called *transducer terms*. Intuitively, the term $T(s_1, \ldots, s_n)$ is interpreted as the output of transducer T on inputs $s_1, \ldots, s_n$. Like constructive terms, such as $X \bullet Y$, transducer terms are allowed only in the heads of rules. The resulting language is called *Sequence Datalog with Transducers*, or simply *Transducer Datalog*. Although this language might appear more powerful than Sequence Datalog, we show that any program in *Transducer Datalog* can be translated into an equivalent program in Sequence Datalog. In other words, we can implement generalized sequence transducers in Sequence Datalog.

To clearly distinguish programs in Transducer Datalog from those in Sequence Datalog, we use two different implication symbols. Whereas rules in Sequence Datalog use the symbol $\leftarrow$, rules in Transducer Datalog use the symbol $\Leftarrow$, as in $p \Leftarrow q, r$.

Transducer Datalog generalizes an idea already present in Sequence Datalog, namely, the use of interpreted function terms. To illustrate, consider the following constructive rule in Sequence Datalog:

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

This rule concatenates every pair of sequences, X and Y, in predicate q. The constructive term $X \bullet Y$ in the head is *interpreted* as the result of concatenating the two sequences together. Transducer Datalog generalizes this mechanism to arbitrary transducers. For example, the following Transducer Datalog program is equivalent to the Sequence Datalog program above:

$$p(T_{append}(X, Y)) \ \Leftarrow \ q(X, Y).$$

where T_{append} is a transducer that concatenates its two inputs. As this example shows, constructive terms are not needed in Transducer Datalog, since they can be replaced by transducer terms. Thus, in the sequel, they will not be used in Transducer Datalog programs.

In Sequence Datalog, rules with constructive terms in the head are called constructive rules (or clauses). The above example suggests a natural extension of this idea: in Transducer Datalog, rules with transducer terms in the head will also be called *constructive rules*.

The semantics of Transducer Datalog is an extension of the semantics of Sequence Datalog. The only change is to extend the interpretation of sequence terms to include transducer terms. This can be done in a natural way. Let θ be a mapping that interprets sequence terms. Thus, $\theta(s)$ is a sequence in Σ^* for any sequence term, s. To extend θ to transducer terms, define $\theta(T(s_1, \ldots, s_m))$ to be the output of transducer T on inputs $\theta(s_1)$,…,$\theta(s_m)$. Except for this change, the semantics of Transducer Datalog is identical to that of Sequence Datalog.

A Transducer Datalog program can be thought of as a *network* of transducers, and vice-versa. This is because the *result* of a transducer term in one rule can be used as an *argument* for a transducer term in another rule. This corresponds to feeding the output of one transducer to an input of another transducer. For example, the following rules feed every sequence in relation *input* through a series of three transducers—first transducer T_1, then transducer T_2, and then transducer T_3:

$$\begin{aligned} p(T_3(X)) &\ \Leftarrow\ q(X). \\ q(T_2(X)) &\ \Leftarrow\ r(X). \\ r(T_1(X)) &\ \Leftarrow\ input(X). \end{aligned}$$

Below we give two examples of sequence restructurings in Molecular Biology. They are naturally represented as transducers or transducer networks. By embedding these transducers in Transducer Datalog, an entire database of sequences can be restructured and queried.

Example 5.1 **[RNA Transcription]**
A fundamental operation in Molecular Biology is the transcription of DNA into RNA. DNA sequences can be modeled as strings over the alphabet $\{a, c, g, t\}$, where each character represents a *nucleotide*. Likewise, RNA sequences can be modeled as strings over the alphabet $\{a, c, g, u\}$, where each character represents a *ribonucleotide*. Each nucleotide in a DNA sequence is transcribed into a ribonucleotide in a RNA sequence according to the following rules:

Each a becomes a u.	Each c becomes a g.
Each g becomes a c.	Each t becomes an a.

Thus, the DNA sequence *acgtacgt* is transcribed into the RNA sequence *ugcaugca*.[4]

This transformation is easily and naturally expressed as a sequence transducer, $T_{transcribe}$, in which the input is a DNA sequence, and the output is a RNA sequence. Given a relation, *dna_seq*, containing DNA sequences, the following Transducer Datalog rule transcribes each of the sequences into RNA:

$$rna\_seq(T_{transcribe}(S)) \Leftarrow dna\_seq(S).$$

□

Although the Transducer Datalog program in Example 5.1 consists of only one rule, two features of this rule are worth noting: (i) all sequence restructurings performed by the program take place "inside" the transducer $T_{transcribe}$; and (ii) the program terminates for every database, since there is no recursion through construction of new sequences. This rule is translated into a more complex Sequence Datalog program in Example 5.3.

Example 5.2 **[Protein Translation]**
Another fundamental operation in Molecular Biology is the translation of RNA into protein. As mentioned above, RNA sequences can be modelled as strings over the alphabet $\{a, c, g, u\}$. Likewise, proteins can be modelled as sequences over a twenty-character alphabet, {A, R, N, D, C, Q, E, G, H, I, L, K, M, F, P, S, T, W, Y, V}, where each character represents an *amino acid*. To translate RNA into protein, ribonucleotides are grouped into triplets, called *codons*, such as *aug*, *acg*, *ggu*, ...[5] Each codon is then translated into a single amino-acid. Different codons may have the same translation. For example, the codons *gau* and *gac* both translate to *aspartic acid*, denoted D in the twenty-letter alphabet. Thus, the RNA sequence *gaugacuuacac* is first grouped into a sequence of four codons, *gau*/*gac*/*uua*/*cac*, and then translated into a sequence of four amino acids, DDLH.[6]

This process is easily and naturally expressed as a sequence transducer, $T_{translate}$, in which the input is a RNA sequence, and the output is a protein sequence. Given a relation, *rna_seq*, containing RNA sequences, the following Transducer Datalog rule translates each of the sequences into protein:

$$protein\_seq(T_{translate}(S)) \Leftarrow rna\_seq(S).$$

The transducer $T_{translate}$ can be combined with the transducer $T_{transcribe}$ from Example 5.1 to form a simple serial network in which the output of $T_{transcribe}$ is the input to $T_{translate}$. This network simulates the transformation of DNA into RNA into protein. This network is represented by the following program:

$$\begin{aligned} rna\_seq(T_{transcribe}(S)) &\Leftarrow dna\_seq(S). \\ protein\_seq(T_{translate}(S)) &\Leftarrow rna\_seq(S). \end{aligned}$$

This program transforms every DNA sequence in the database into an RNA sequence, and then into a protein sequence. □

5.1 Equivalence to Sequence Datalog

At first glance, it might seem that Transducer Datalog is a more powerful language than Sequence Datalog. This is not the case, however, as the following theorem shows.

Theorem 7 *Transducer Datalog and Sequence Datalog are expressively equivalent. i.e., Every database query expressible in Transducer Datalog can be expressed in Sequence Datalog, and vice-versa.*

The crux of the proof is the simulation of transducer computations by rules in Sequence Datalog. In the simulation, transducer output is constructed by recursive concatenation, while safe, structural recursion "scans" the input sequences and ensures termination. Moreover, and even more important, the translation of Transducer Datalog into Sequence Datalog preserves the complexity of programs. That is, if the least fixpoint of a Transducer Datalog program over a database can be computed in time t, then the fixpoint of the translation can be computed in time $\mathcal{O}(t)$ [22].

Example 5.3 **[Simulating Transducers]**
The Sequence Datalog program below simulates the Transducer Datalog rule in Example 5.1:

[4] For simplicity, this example ignores complications such as *intron splicing* [31], even though it can be encoded in Transducer Datalog without difficulty.

[5] This grouping is analogous to the grouping of *bits* into *bytes* in computers.

[6] For simplicity, this example ignores complications such as reading frames, ribosomal binding sites, and stop codons [31].

$$rna\_seq(R) \leftarrow dna\_seq(D),\quad transcribe(D, R).$$

$$transcribe(D[1{:}N+1],\, R \bullet T) \leftarrow dna\_seq(D),\quad transcribe(D[1{:}N], R),\ trans(D[N+1], T).$$

$$transcribe(\epsilon, \epsilon) \leftarrow true.$$

$$trans(a, u) \leftarrow true. \qquad trans(t, a) \leftarrow true.$$
$$trans(g, c) \leftarrow true. \qquad trans(c, g) \leftarrow true.$$

The first rule transcribes every DNA sequence in the relation *dna_seq* into an RNA sequence, by invoking the predicate *transcribe*. This predicate simulates the transducer $T_{transcribe}$ in Example 5.1. The formula $transcribe(D, R)$ is true iff D is a prefix of a DNA sequence in the database and R is the RNA transcription of D. The second rule recursively scans each DNA sequence, while constructing an RNA sequence. For each character in the DNA sequence, its transcription, T, is concatenated to the growing RNA sequence. The last four rules specify the transcription of individual characters. *i.e.*, The formula $trans(d, r)$ means that d is a character in the DNA alphabet, and r is its transcription in the RNA alphabet.

□

Theorem 7 shows that the introduction of transducers into Sequence Datalog does not increase the expressive power of the logic. However, transducers do provide a framework for defining natural syntactic restrictions that guarantee safety and finiteness while preserving much of the expressive power of the full logic, as the next section shows.

6 A Safe Query Language for Sequences

This section develops syntactic restrictions that define a sublanguage of Transducer Datalog, called *strongly safe Transducer Datalog*, that is both finite and highly expressive. The restrictions forbid recursion through transducer terms. Intuitively, this ensures that the transducer network corresponding to a program is acyclic.

The syntactic restrictions are defined in terms of predicate dependency graphs. These graphs represent dependencies between predicates in rule heads and rule bodies.

Definition 2 [Dependent Predicates]
Let P be a Transducer Datalog program. A predicate symbol p depends *on predicate symbol q in program P if for some rule in P, 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.*

Definition 3 [Dependency Graph]
Let P be a Transducer Datalog program. The predicate dependency graph of P is a directed graph whose nodes are the predicate symbols in P. There is an arc from p to q in the graph if p depends on q in program P. The edge is constructive *if p depends constructively on q. A* constructive cycle *is a cycle in the graph containing a constructive edge.*

We say that a Transducer Datalog program is *strongly safe* if its predicate dependency graph does not contain any constructive cycles. The programs in Examples 5.1 and 5.2 are strongly safe since they are non-recursive, and thus their dependency graphs contain no cycles. In the following example, all the programs are recursive, and one of them is strongly safe.

Example 6.1 Consider the following three Transducer Datalog programs, P_1, P_2 and P_3:

$$P_1 : \begin{cases} p(X) \Leftarrow r(X,Y),\ q(Y). \\ q(X) \Leftarrow r(X,Y),\ p(Y). \\ r(T_1(X), T_2(Y)) \Leftarrow a(X,Y). \end{cases}$$

$$P_2 : \{\ p(T(X)) \Leftarrow p(X).$$

$$P_3 : \begin{cases} q(X) \Leftarrow r(X). \\ r(T(X)) \Leftarrow p(X). \\ p(X) \Leftarrow q(X). \end{cases}$$

All three programs are recursive, so their predicate dependency graphs all have cycles. The graphs of P_2 and P_3 have constructive cycles, while the graph of P_1 does not. Thus, P_1 is strongly safe, while P_2 and P_3 are not. □

Theorem 8 *If a Transducer Datalog program is strongly safe, then it is finite.*

The complexity of a strongly safe program depends on the order of its transducers. We say that a program in Transducer Datalog is of *order k* if k is the maximum order of all transducers in the program. (A program with no transducers has order 0.) The following theorems build on Theorems 5 and 6, and establish expressibility results for strongly safe programs.

Theorem 9 *Strongly Safe Transducer Datalog programs of order 2 express exactly the class of sequence functions computable in* PTIME.

Theorem 10 *Strongly Safe Transducer Datalog programs of order 3 expresses exactly the class of elementary sequence functions.*

Acknowledgments
The authors would like to thank Steve Cook, Faith Fich and Charles Rackoff for fruitful discussions on the subject of the paper, and especially on the use of machines for computing sequence functions.

References

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

[2] A. Albano, L. Cardelli and R. Orsini. Galileo: A strongly typed language for complex objects. In *ACM TODS*, 10(2), 1985.

[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, S. Cluet and C. Delobel. A query language for the O_2 object-oriented database system. In *DBPL*, 1989.

[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 L. Wong. Naturally embedded query languages. In *ICDT*, 1992.

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

[10] 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.

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

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

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

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

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

[16] C. Hegelsen and P. R. Sibbald. PALM – A pattern language for molecular biology. In *Proc. First Int. Conf. on Intelligent Systems for Molecular Biology (ISMB)*, 1993.

[17] R. Hull and J. Su. On the expressive power of database queries with intermediate types. *Journal of Comp. and System Sc.*, 43(1), 1991.

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

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

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

[21] G. Mecca and A. J. Bonner. Generalized Sequence Transducers. In preparation. 1995.

[22] G. Mecca and A. J. Bonner. Sequences, Datalog and Transducers. Technical Report. In preparation. 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] S. Vandenberg and D. De Witt. Algebraic support for complex objects with arrays, identity and inheritance. In *ACM SIGMOD*, 1991.

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

[30] X. Wang. Pattern matching by RS-Operations: towards a unified approach to querying sequenced data. PhD Dissertation, University of Southern California, 1992.

[31] J.D. Watson, et al. *Molecular Biology of the Gene*, Fourth Edition. Benjamin/Cummings Publishing Company, 1987.

[32] P. Wolper. Temporal logic can be more expressive.
Information and Control, 56, 1983.