Querying Sequence Databases with Transducers

Anthony Bonner
Department of Computer Science
University of Toronto — Canada M5S 1A1
`bonner@db.toronto.edu`

Giansalvatore Mecca
D.I.F.A. — Università della Basilicata
Via della Tecnica, 3 — 85100 Potenza, Italy
`mecca@dis.uniroma1.it`

Abstract

This paper develops a database query language called *Transducer Datalog* motivated by the needs of a new and emerging class of database applications. In these applications, such as text databases and genome databases, the storage and manipulation of long character sequences is a crucial feature. The issues involved in managing this kind of data are not addressed by traditional database systems, either in theory or in practice. To address these issues, in recent work, we introduced a new machine model called a *generalized sequence transducer*. These generalized transducers extend ordinary transducers by allowing them to invoke other transducers as "subroutines." This paper establishes the computational properties of Transducer Datalog, a query language based on this new machine model. In the process, we develop a hierarchy of time-complexity classes based on the Ackermann function. The lower levels of this hierarchy correspond to well-known complexity classes, such as polynomial time and hyper-exponential time. We establish a tight relationship between levels in this hierarchy and the depth of subroutine calls within Transducer Datalog programs. Finally, we show that Transducer Datalog programs of arbitrary depth express exactly the sequence functions computable in primitive-recursive time.

Appears in *Proceedings of the Sixth International Workshop on Database Programming Languages (DBPL)*, August 18–20, 1997, Estes Park, Colorado.

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

1 Introduction

This paper develops a database query language, called *Transducer Datalog*, motivated by the needs of a new and emerging class of database applications. In these applications, such as text databases and genome databases, the storage and manipulation of long sequences is a crucial feature [12, 10]. The issues involved in managing this kind of data are not addressed by traditional database systems, either in theory or in practice. In particular, traditional database query languages are poor at manipulating sequence data. This is because they treat data items as atomic entities without any internal structure. Thus, they cannot interrogate or manipulate the sequential structure of items such as documents, DNA sequences and protein sequences.

To address this limitation, sequence operations have been introduced into recent data models and query languages (e.g. [4, 7]). In many cases, however, the sequence operations are *ad hoc* and are not investigated in a theoretical framework. In other cases, (e.g. [17, 26]) the operations are designed for pattern extraction but not for sequence restructuring. Although pattern recognition is a fundamental feature of any language for querying sequences, sequence restructurings are equally important. For example, in genome databases, one frequently needs to concatenate sequences together, to splice out selected subsequences, and to compute the reverse of a sequence.

Introducing sequence operations into database query languages presents interesting theoretical challenges. For instance, the resulting query language should be *expressive* both in terms of pattern matching and sequence restructurings. At the same time, it should be *safe*; that is, a query should always return a finite answer and terminate in a finite amount of time. Unlike traditional databases, safety and finiteness of computations are major problems in sequence databases, since sequences can grow to arbitrary length, and the set of possible sequences is infinite, even when the underlying alphabet—or domain—is finite. This means that, unlike traditional database queries, sequence queries can end up in non-terminating computations.

To achieve expressiveness, database researchers have developed sequence query languages based on abstract machines, such as automata or *transducers*. Unfortunately, the resulting languages are unsafe, and to achieve safety, syntactic restrictions have been imposed, restrictions that severely limit expressiveness (*e.g.*, [11, 14]). To resolve these problems, we recently developed a query language for sequence data called *Transducer Datalog* [22, 6, 20], which extends classical Datalog [8] so that it can handle sequence data. A key element in Transducer Datalog is a new machine model called a *generalized sequence transducer*. Intuitively, a generalized transducer is a transducer that can invoke other transducers as "subroutines." These *subtransducers* can likewise invoke other transducers as sub-subroutines. Etc. Like ordinary transducers, generalized transducers always terminate. The result is a query language for sequence databases that is safe; moreover, because generalized transducers are more powerful than ordinary transducers, the query language has considerable expressive power, and its computational complexity can be tuned by restricting the transducers used in the language. These properties of the query language, which are a result of the computational properties of the generalized transducers, are studied in this paper.

By coupling a declarative query language, like Datalog, with a procedural model of computation, like generalized transducers, we have been able to develop a flexible paradigm for querying sequence databases. In Transducer Datalog, set-oriented database operations are easily expressed using a semantics based on Datalog, while powerful sequence manipulation procedures can be programmed using transducers. The language itself combines database query primitives with a powerful computational model for sequences, which is an important requirement for sequence databases. For example, in genome databases, because genome-technology is rapidly evolving, new sequence operations are constantly needed, operations that cannot be anticipated in advance. Genome databases thus need to combine a query language with arbitrary procedures for executing sequence operations [13]. Clearly, different models of computation could be embedded in the query language to achieve the same goal. However, unlike other models, generalized transducers are highly modular, and although very powerful, they preserve finiteness and allow one to tune the complexity of the language.

In previous work ([22, 6]), we established the complexity and expressibility of a simple class of generalized transducers, namely those with at most two levels of subtransducer calls. In this paper, we generalize and extend those results by characterizing the complexity of arbitrarily nested transducers. Section 3 first reviews the definition of generalized transducers. It then introduces a restriction called *controlled feedback*, and shows that this leads to an elegant characterization of the complexity and expressibility of generalized transducers in terms of the Ackermann function [2, 25].

Section 4 reviews the query language Transducer Datalog, which is obtained by enriching classical Datalog with interpreted function terms called *transducer terms*. This language allows a user to compose transducers and combine them into "networks," in which the output of one transducer is fed to the inputs of other transducers. The language is an extension of Horn logic and is capable of expressing *sequence queries*, *i.e.*, queries to databases containing sequences over a fixed alphabet. The expressive power of classical Datalog grows dramatically when transducers are added to it. We show, for instance, that Transducer Datalog with stratified negation is computationally complete for sequence databases. This completeness result extends our previous results on function expressibility [22, 6] to query expressibility.

Like classical Datalog, queries in Transducer Datalog can be recursive. However, in Transducer Datalog, the combination of recursion and transducers allows queries to be infinite and non-terminating. To guarantee finiteness and termination, we previously introduced a syntactic restriction called *safety*, in which recursion through transducer terms is not allowed [22, 6, 20]. Section 5 reviews the definition of safety, and then establishes several new expressibility results for safe programs. We show in particular that the complexity of a safe program depends on its *order*, *i.e.*, on the maximum depth to which transducers are invoked as subroutines. Formally, we first develop a hierarchy of time-complexity classes based on the Ackermann function [2, 25]. We then show that the lower levels of this hierarchy correspond to well-known complexity classes, such as PTIME and the *elementary functions* [24], i.e. hyper-exponential time. This extends the complexity results of our previous work [22, 6]. Finally, we show that safe programs of order k express exactly the sequence functions in the k^{th} level of this hierarchy. As a corollary, we show that safe Transducer Datalog expresses exactly the sequence functions computable in primitive-recursive time.

The query completeness result and the characterization in terms of the Ackermann complexity hierarchy represent the main contributions of the paper. For space reasons, proofs are omitted and can be found in the full paper [5].

Note that the Ackermann hierarchy is somewhat similar to the *Grzegorczyk hierarchy* [16] of primitive recursive functions. However, the latter is based on a different set of initial functions and each level is obtained by closure with respect to substitution and bounded primitive recursion, instead of function composition. In [18] a hierarchy of primitive recursive functions based on the Grzegorczyk hierarchy is presented to characterize the behavior of a family of finite automata. However, unlike the generalized transducers used here, the machines used in [18] are not finite state devices in the usual sense, since a "state" is essentially a counter, so a machine of this kind can have infinitely many states. In contrast, generalized transducers have a finite number of states.

Other characterizations of the primitive recursive functions have been presented in the literature. For instance, a characterization in terms of *loop programs* is developed in [23]. The complexity of loop programs is also based on a hierarchy of complexity classes; however, the latter is different from the Ackermann's hierarchy developed in this paper, since, for example, there is no level equivalent to PTIME.

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. We now describe an extension of the relational model, in the spirit of [11, 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 databases over Σ to the relations over Σ. Given a sequence query, Q, and a database, DB, $Q(\text{DB})$ is the result of evaluating Q over DB. Usually, a notion of *genericity* [9] is introduced for queries. The notion can be extended to sequence queries in a natural way. To do this, we consider permutations of alphabet symbols in Σ, that is, one-to-one mappings $\rho : \Sigma \to \Sigma$ of Σ onto itself; permutations can be extended in a natural way to strings in Σ^*, tuples of strings, relations and databases. We say that a sequence query Q is *generic* if, for each permutation ρ of alphabet symbols, Q commutes with ρ, that is, for every database DB such that Q is defined on DB, it is the case that: $Q(\rho(\text{DB})) = \rho(Q(\text{DB}))$. We say that a sequence query Q is *computable* [9] if it is generic and partial recursive.

A *sequence function* is a partial mapping from Σ^* to itself. 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. Thus, we say that a language, L, expresses a sequence function if it expresses the corresponding query. A sequence function f is *computable* if it is partial recursive. This paper focuses on sequence functions for which the alphabet, Σ, is finite. Of course, the domain of such functions may still be infinite; *e.g.*, it may be all of Σ^*.

In this paper, we address the complexity of sequence functions, and the data complexity [28] of sequence queries. Given a sequence function, f, the *complexity* of f is defined in the usual way, as the complexity of computing $f(\sigma_{in})$, measured with respect to the length of the input sequence, σ_{in}. 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 language, L, is said to *express* a class C of sequence functions, if (i) each sequence function expressible in L has complexity in C, and conversely, (ii) each sequence function with complexity in C can be expressed in L. Likewise, we say that a language L *expresses* a class QC of queries 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.

3 Generalized Sequence Transducers

The notion of generalized sequence transducer was introduced in [22, 6, 20]. After reviewing the definition, this section introduces a restriction called *controlled feedback*, and shows that it leads to a characterization of the complexity of generalized transducers in terms of the Ackermann function [2, 25].

Ordinary *transducers* – i.e., finite state machines with n input lines and one output line – 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 as a subroutine. 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 sub-subtransducer, etc. Subtransducers are analogous to subroutine calls in programming languages, and to some extent, 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 $k-1$; k is called the *order* of the transducer. $\mathcal{T}^1$ 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, $\triangleleft$, $\rightarrow$ and $-$. $\triangleleft$ 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 other transducer models proposed in the literature (for example, see [11, 27]).

A *generalized n-input sequence transducer* of *order k* (for $k > 0$) 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 $K \times \{\Sigma \cup \{\triangleleft\}\}^n$ to $K \times \{-, \rightarrow\}^n \times \{\Sigma \cup \{\epsilon\} \cup \mathcal{T}^{k-1}\}$, called the *transition function*. $\mathcal{T}^k$ consists of all generalized transducers of order at most k, for $k > 0$; $\mathcal{T}^0 = \{\}$.

For each transition, $\delta(q, a_1, \ldots, a_n)$ of the form $\langle q', c_1, \ldots, c_n, \textsc{out}\rangle$, we impose three restrictions: (i) at least one of the c_i must be $\rightarrow$, (ii) if $a_i = \triangleleft$ then $c_i = -$, and (iii) if $\textsc{out} \in \mathcal{T}^{k-1}$ then it must be an $n+1$-ary transducer. These restrictions 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 $\delta(q, a_1, \ldots, a_n) = \langle q', c_1, \ldots, c_n, \textsc{out}\rangle$. This transition is carried out as follows:

- If $\textsc{out}$ is a symbol in Σ, then it is appended to the output sequence; if $\textsc{out} = \epsilon$, then the output is unchanged. This takes constant time.

- If $\textsc{out}$ 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. When the subtransducer starts, its head scan the first symbol of the corresponding input tape, and the output tape is initialized to the empty sequence. The transfer of control and initialization of the subtransducer takes constant time. The subtransducer then consumes its inputs and produces an output sequence. When it is finished, the output of the subtransducer is copied to (overwrites) the output tape of the calling transducer. The overwriting takes constant time.

step	input	output	operation	new output
1	$\boxed{a}bc\triangleleft$	ϵ	invoke T_{append}	abc
2	$a\boxed{b}c\triangleleft$	abc	invoke T_{append}	$abcabc$
3	$ab\boxed{c}\triangleleft$	$abcabc$	invoke T_{append}	$abcabcabc$

Figure 1: Squaring the input.

This table illustrates the computation of transducer T_{square} from Example 3.1 on the input sequence abc. *Each row of the table represents one step of the computation. The column labelled* "input" *shows the position of the input head just before the step, the column labelled* "output" *shows the contents of the output tape just before the step, and the column labelled* "new output" *shows the contents of the output tape just after the step. The column labelled* "operation" *describes what the step does. The table shows that at each step, T_{square} appends a copy of its input to its output by invoking the subtransducer T_{append}.*

- The transducer "consumes" some input by moving at least one tape head one symbol to the right. This takes constant time.
- The transducer enters the next state, q', and resumes its computation. This takes constant time.

The transducer stops when every input tape has been completely consumed, that is, when every input head reads the symbol $\triangleleft$. 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 in Example 3.1.

Example 3.1 **[Quadratic Output]** Let T_{square} be a generalized transducer with one input and one subtransducer. At each computation step, T_{square} appends a copy of its input to its output. After k steps, the output consists of k copies of the input concatenated together. At the end of the entire computation, the output has length n^2, where n is the length of the input. To append its input to its output, T_{square} invokes a subtransducer called T_{append} with two inputs. One input to T_{append} is the input to T_{square} and the other input 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.

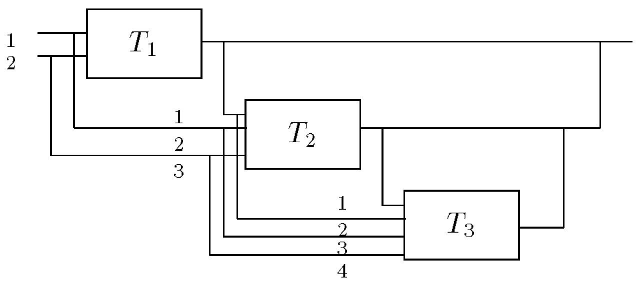
To illustrate the combined effect of transducer and subtransducer, let σ_{in} be the input to T_{square}. Just before the i^{th} invocation of the subtransducer, one input to T_{append} is σ_{in} and the other input is $i-1$ copies of σ_{in} concatenated together. Thus, when T_{append} finishes this computation, its output is i copies of σ_{in} concatenated together. This behaviour is illustrated in Figure 1, in which the transducer input is abc, and the output is $abcabcabc$. □

3.1 Controlled Feedback

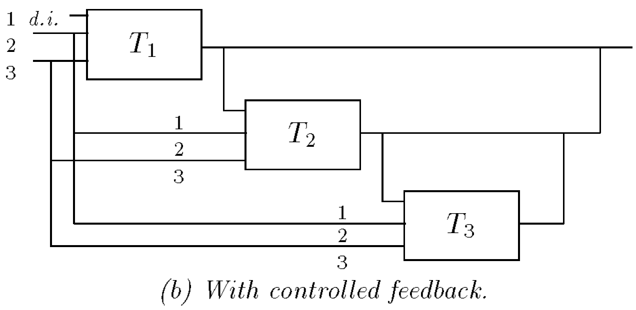
In [22, 6], we established the complexity of generalized transducers of order 2 and 3. This paper extends those results to transducers of arbitrary order. To do this, we focus on a transducer model that is slightly more restrictive than the one described above, but which is identical to it for transducers of order 1 and 2.

In the generalized transducer model developed above, a subtransducer has access to all the input lines of the transducer that invoked it, plus a copy of its output line. A subtransducer thus has one more input than its parent. Consequently, suppose that transducer T_1 invokes T_2, which invokes T_3, which invokes... T_i, which invokes T_{i+1}. Then the input to transducer T_{i+1} consists of copies of all the inputs to T_1 plus all the outputs of $T_1, T_2, ..., T_i$. Thus, if T_1 has n inputs, then T_{i+1} has $n + i$ inputs, as illustrated in Figure 2(a).

To reduce the amount of feedback and simplify the computation, we now restrict the model. In the restricted transducer model, a subtransducer has access to the output line of the transducer that invoked it (its "parent"), but *not* the output lines of any higher transducers. Thus, the input to transducer T_{i+1} above includes the output of T_i, but *not* the outputs of $T_1, ..., T_{i-1}$. To formalize this idea, we provide each transducer with a single *distinguished input* line. When a transducer invokes a subtransducer, all non-distinguished inputs of the transducer become non-distinguished inputs of the subtransducer. In addition, the output of the transducer becomes the distinguished input of the subtransducer. In this way, all non-distinguished inputs are passed from a transducer to all its descendants, but the output is passed only to the transducer's immediate descendants (its "children"). We define the distinguished input to the top-level transducer to be the empty sequence.



(a) Without controlled feedback.



(b) With controlled feedback.

Figure 2: An order-3 transducer with and without controlled feedback.
T_1 is the top-level transducer, T_2 is a subtransducer of T_1, and T_3 is a sub-subtransducer. In (b), the distinguished input (d.i.) of T_1 is empty, by convention.

Consequently, suppose that transducer T_1 invokes T_2, which invokes T_3, which invokes... T_i, where T_1 is the top-level transducer. Then, the non-distinguished inputs of each T_j are a copy of the non-distinguished inputs of T_1. In addition, the distinguished input of T_1 is empty, and the distinguished input of T_{j+1} is a copy of the output of T_j, for $1 \leq j < i$. Each transducer T_j thus has the *same* number of inputs. Moreover, since all output lines are initially empty, all distinguished input lines are initially empty too. These ideas are illustrated in Figure 2(b). Because this model restricts the amount of feedback from outputs to inputs within a generalized transducer, we refer to it as the model with *controlled feedback*.

In the following sections, we will always refer to transducers with controlled feedback. Note, however, that the distinction is significant only for transducers of order greater than 2, since for orders 1 and 2, the distinction is irrelevant.

3.2 Complexity of Generalized Transducers

We are now ready to establish two important results about the complexity of generalized sequence transducers. We shall see that the deterministic time complexity of a transducer is closely related to the *Ackermann function* [2, 25], also called the *generalized exponential function*. This is a ternary function $\textsc{ack}(k, x, y)$ such that:

$$\begin{array}{lcl} \textsc{ack}(1,x,y) & = & y+x, \\ \textsc{ack}(2,x,y) & = & yx, \\ \textsc{ack}(3,x,y) & = & y^x, \\ \ldots & & \end{array}$$

We call the first argument, k, the *level* of the function. For each level, $\textsc{ack}(k, x, y)$ can be viewed as a function of two variables, x and y. Thus, the first levels of the function capture well known operations: level 1 is *addition*, level 2 is *multiplication*, and level 3 is *exponentiation*. Formally, the Ackermann function is defined using double induction, as follows:

Definition 1 (Ackermann's Function)[1] *For $k, x, y \geq 0$,*

$$\begin{array}{lcl} \textsc{ack}(0,x,y) & = & 0, \\ \textsc{ack}(1,0,y) & = & y, \\ \textsc{ack}(1,x+1,y) & = & \textsc{ack}(1,x,y)+1, \\ \textsc{ack}(2,0,y) & = & 0, \\ \textsc{ack}(k,0,y) & = & 1, \quad \textit{for all } k>2, \\ \textsc{ack}(k+1,x+1,y) & = & \textsc{ack}(k,\textsc{ack}(k+1,x,y),y), \quad \textit{for all } k \geq 1. \end{array}$$

This paper uses mainly a one-argument version of the Ackermann function in which x is equal to y and the level, k, is fixed. We therefore define the following: $\textsc{ack}_k(y) = \textsc{ack}(k, y, y)$.

The following theorem provides an upper bound on the time complexity of generalized transducers with controlled feedback in terms of the Ackermann function.

Theorem 1 (Upper Complexity Bound) *Given a generalized transducer of order k, its running time (and output length) is $\textsc{ack}_k(\mathcal{O}(n))$, where n is the sum of the input lengths.*

The next theorem shows that the relationship between generalized transducers and the Ackermann function is even closer. In particular, for each k, the function $\textsc{ack}(k, x, y)$ can be computed by a generalized transducer of order k. This result is based on a unary encoding of integers. An integer n is thus encoded as a sequence of length n consisting of n occurrences of the same character, #. We say that a transducer *computes* an integer function $v = f(w)$ if on input $\#^w$, of length w, it produces the output $\#^{f(w)}$, of length $f(w)$.

[1]This definition is a minor variant of the definition given in [2]. In particular, $\textsc{ack}(k, x, y)$ in our definition coincides with $\textsc{ack}(k-1, x, y)$ in the definition of [2].

Theorem 2 **(Computing Ackermann's Function)** *For every integer $k \geq 1$, there is a generalized sequence transducer of order k with 2 inputs that computes $\textsc{ack}(k, x, y)$. That is, for all inputs x, y with $x > 0$, the output of the transducer is $\textsc{ack}(k, x, y)$.*

Theorem 2 shows that the upper bound in Theorem 1 is tight. These results will prove useful in the following sections.

4 Transducer Datalog

In [22, 6, 20], we developed a query language for sequence databases by incorporating generalized transducers into classical Datalog [8]. This language allows an application programmer to use the framework of classical Datalog to combine generalized transducers into *networks*, in which the output of one transducer can be fed to the inputs of other transducers. Such networks greatly increase the ability of Transducer Datalog to express database queries, as we shall see. This section first reviews the syntax and semantics of Transducer Datalog, and then shows that it is expressively complete for sequence database queries. Due to space limitations, we cannot present the formal semantics of Transducer Datalog here. Instead, we illustrate the semantics informally through examples. The interested reader is referred to [22, 6, 20] for details.

4.1 Syntax and Semantics

To invoke transducer computations from within a logical rule, we augment the syntax of classical Datalog (function-free Horn logic) with special interpreted function symbols, one symbol 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*. The term $T(s_1, \ldots, s_n)$ is interpreted as the output of transducer T on inputs $s_1, \ldots, s_n$. Transducer terms are allowed only in the heads of rules. The resulting language is called *Transducer Datalog*. Transducer Datalog uses generalized transducers to manipulate sequences in the language. For example, suppose we want to concatenate every pair of sequences X, Y in predicate q. The following Transducer Datalog program performs this task:

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

where T_{append} is a transducer that concatenates its two inputs. In Transducer Datalog, rules with transducer terms in the head are called *constructive* rules (or clauses), since they can construct new sequences during inference. We also say that a program in Transducer Datalog has *order k* if k is the maximum order of all transducers in the program. A program with no transducers, *i.e.* a classical Datalog program, has order 0.

The semantics of Transducer Datalog generalizes the semantics of classical Datalog. The main change is to extend the interpretation of terms to include transducer terms. 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 a set of *function symbols*, $T_1, T_2, \ldots$, each with an associated arity. A constant sequence (or *sequence*, for short) is an element of Σ^*. From these sets, we construct the set of *sequence terms* (or *terms*, for short), as follows:

- each constant sequence in Σ^* is a term;
- each sequence variable in V_Σ is a term;
- given n terms $s_1, \ldots, s_n$ and a function symbol T of arity n, $T(s_1, \ldots, s_n)$ is a term, called a *transducer term*; we require that transducer terms not contain other transducer terms; thus, each argument s_i is either a constant sequence or a variable.

As in most logics, the language of formulas for *TransducerDatalog* also includes a countable set of predicate symbols, $p, q, r, ...$, each with an associated arity. If p is a predicate symbol of arity n, and $s_1, ..., s_n$ are sequence terms, then $p(s_1, ..., s_n)$ 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 *facts* and *clauses* in the usual way [19]. A clause that contains a transducer term in its head is called a *constructive clause*. A Transducer Datalog *program* is a set of Transducer Datalog clauses. As usual, we require that each variable occurring in the head of a rule also occur in the body of the rule.

A substitution, θ, is a mapping that associates a sequence with each variable in V_Σ. Substitutions can be extended to mappings on sequence *terms* in a straightforward way. Because these terms are interpreted, the result of a substitution is a constant sequence. Formally, let θ be a substitution. θ is the identity function for each sequence in Σ^*. To extend θ to transducer terms, we define $\theta(T(s_1, \ldots, s_m))$ to be the output of transducer T on inputs $\theta(s_1), \ldots, \theta(s_m)$, where the symbols of each $\theta(s_i)$ are consumed from left to right.

The semantics of clauses is defined in terms of a least fixpoint theory. As in classical logic programming [19, 8], each Transducer Datalog program, P, and database, DB, has an associated operator, $T_{P,\mathrm{DB}}$, that maps relational databases to relational databases. Each application of $T_{P,\mathrm{DB}}$ may create new atoms, which may contain new sequences. We have shown that $T_{P,\mathrm{DB}}$ is monotonic and continuous, and thus has a least fixpoint that can be computed in a bottom-up, iterative fashion [22, 6, 20].

If a Transducer Datalog program has recursion through transducer terms, then the least fixpoint of the program can be infinite. That is, as a result of evaluating transducer terms, new sequences can be computed; and because of recursion, the length of these sequences can be unbounded. In this case, we say that the semantics of P over DB is *infinite*; otherwise, we say it is *finite*.

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.

Below we give an example of sequence restructuring in Molecular Biology. It is naturally represented as a transducer network. By embedding this network in Transducer Datalog, an entire database of sequences can be restructured and queried.

Example 4.1 **[From DNA to RNA to Protein]** Two fundamental operations in Molecular Biology are the transcription of DNA into RNA, and the translation of RNA into protein. All three molecules in this process can be modelled as sequences. DNA molecules are modeled as sequences over the alphabet $\{a, c, g, t\}$, where each character represents a *nucleotide*. RNA molecules are modeled as sequences over the alphabet $\{a, c, g, u\}$, where each character represents a *ribonucleotide*. During transcription, each nucleotide in a DNA sequence is converted into a ribonucleotide in an RNA sequence according to the following rules:

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

For example, the DNA sequence *acgtacgt* is transcribed into the RNA sequence *ugcaugca*.[2]

Protein molecules are modeled 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*, ...[3] Each codon is then translated into a single amino-acid. Different codons may

[2]For simplicity, this example ignores biological complications such as intron splicing [29], even though it can be encoded in Transducer Datalog without difficulty.

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

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.[4]

The transformation of DNA into the corresponding protein is easily and naturally expressed using a Transducer Datalog program. Given a relation, *dna_seq*, containing DNA sequences, the following Transducer Datalog program associates a protein with each sequence.

$$\begin{aligned} rna\_seq(D, T_{transcribe}(D)) &\leftarrow dna\_seq(D). \\ protein\_seq(D, T_{translate}(R)) &\leftarrow rna\_seq(D, R). \end{aligned}$$

The program implements a simple serial network which transforms DNA into RNA, and then RNA into protein. It uses two sequence transducers: (i) $T_{transcribe}$, in which the input is a DNA sequence, and the output is a RNA sequence; (ii) $T_{translate}$, in which the input is a RNA sequence, and the output is a protein sequence. Intuitively, the predicate $rna\_seq(D, R)$ means that D is a DNA sequence, and R is the corresponding RNA sequence. Likewise, the predicate $protein\_seq(D, P)$ means that D is a DNA sequence, and P is the corresponding protein sequence.

Although the Transducer Datalog program consists of only two rules, two features are worth noting: (*i*) all sequence restructurings performed by the program take place "inside" the transducers; and (*ii*) the program terminates for every database, since there is no recursion through construction of new sequences.

□

4.2 Expressive Power

Transducer Datalog has considerable power for manipulating sequences, and can express any computable sequence function [22]. However, although Transducer Datalog is function complete, it is not query complete, since it expresses only *monotonic* queries. Thus, like classical Datalog, it cannot express non-monotonic queries, even very simple ones, such as the difference of two database relations. To achieve query completeness, we must add some form of non-monotonicity to the language. We do this by allowing negated atoms in rule bodies, and interpreting negation as failure. To avoid the plethora of problems associated with recursion through negation, we restrict our attention to *stratified negation* [3, 8], which can be extended from classical Datalog to Transducer Datalog in a natural way. The following theorem is the main result about this language, which we call *stratified Transducer Datalog.*

Theorem 3 (Completeness) *Stratified Transducer Datalog expresses all computable sequence queries.*

5 Safe Transducer Datalog

In [22, 6, 20], we introduced syntactic restrictions that define a sublanguage of Transducer Datalog called *safe Transducer Datalog.* The restrictions forbid recursion through transducer terms in a rather standard way [1], in order to guarantee that the transducer network corresponding to a Transducer Datalog program is acyclic. From a database perspective, safe Transducer Datalog programs are interesting because they are *finite*, *i.e.*, their least fixpoint is finite for every database [22, 6, 20], and can thus be materialized. This section reviews the definition of safe Transducer Datalog, and establishes results on its expressibility. Formally, we introduce a hierarchy of complexity classes

[4] For simplicity, this example ignores biological complications such as reading frames, ribosomal binding sites, and stop codons [29].

based on the Ackermann function, and then we characterize the expressibility of natural fragments of safe Transducer Datalog in terms of this hierarchy. As a corollary, we show that safe Transducer Datalog expresses exactly the sequence functions computable in primitive-recursive time.

5.1 Safety

Let P be a Transducer Datalog program. We say that 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. Now, 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 edge 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.

Definition 2 (Safe Programs) *A Transducer Datalog program is* safe *if its predicate dependency graph does not contain any constructive cycles.*

The program in Example 4.1 is safe since it is non-recursive, and thus its dependency graph contains no cycles.

5.2 The Ackermann Time Hierarchy

This section develops a hierarchy of time-complexity classes based on the Ackermann function. This hierarchy will be used in the next subsection to characterize the expressibility of Transducer Datalog programs. In particular, we shall see that programs of order k express exactly the sequence functions in the k^{th} level of the hierarchy.

Our time-complexity classes are defined in terms of repeated function composition. Given a unary function $f(x)$, the *composition* of f with itself i times, denoted $f^{(i)}(x)$, is defined in the usual way: (i) $f^{(1)}(x) = f(x)$; (ii) $f^{(i+1)}(x) = f(f^{(i)}(x))$ *for* $i \geq 1$. By composing the level-k Ackermann function $\text{ACK}_k(y)$ with itself i times, we obtain a new function, $\text{ACK}_k^{(i)}(y)$. It is not hard to see that for each i, $\text{ACK}_2^{(i)}(y)$ is a polynomial in y, since $\text{ACK}_2(y) = y^2$, and $\text{ACK}_3^{(i)}(y)$ is a hyper exponential in y, since $\text{ACK}_3(y) = y^y$.

Definition 3 (Ackermann Time Hierarchy) *For each $k \geq 0$, k-ACKTIME is the class of sequence functions computable in deterministic time $\text{ACK}_k^{(i)}(\mathcal{O}(n))$ for some i. That is:*

$$k\text{-ACKTIME} = \bigcup_{i \geq 1} \text{DTIME}[\text{ACK}_k^{(i)}(\mathcal{O}(n))]$$

The Ackermann time hierarchy *is the union of k-ACKTIME for all k:*

$$\text{ACK-HIER} = \bigcup_{k \geq 0} k\text{-ACKTIME}$$

The Ackermann time hierarchy has close connections with well-known complexity and computability classes. For instance, it is not hard to see that (i) 2-ACKTIME is PTIME, and (ii) 3-ACKTIME is the class of *elementary sequence functions* [24], that is, the class of sequence functions computable in hyper-exponential time. In addition, the hierarchy is related to the classical notion of *primitive recursive function* [25]. Formally, let PR-TIME denote the class of sequence functions computable in primitive-recursive time. That is, $\text{PR-TIME} = \bigcup_{f \in \mathcal{PR}} \text{DTIME}[f(\mathcal{O}(n))]$, where $\mathcal{PR}$

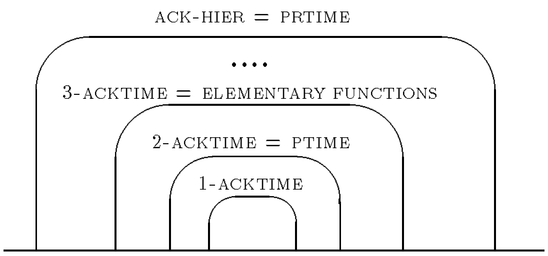


Figure 3: The Ackermann's hierarchy of complexity classes

is the set of primitive recursive functions. The Ackermann time hierarchy coincides with this time class; that is ACK-HIER = PR-TIME. These relationships are summarized in Figure 3.

Note that the Ackermann hierarchy is somewhat similar to the *Grzegorczyk hierarchy* [16] of primitive recursive functions. However, the latter is based on a different set of initial functions and each level is obtained by closure with respect to substitution and bounded primitive recursion, instead of function composition.

5.3 Expressibility of Safe Programs

The following theorems establish expressibility results for safe Transducer Datalog programs in terms of the Ackermann time hierarchy. All theorems refer to the transducer model with controlled feedback.

Theorem 4 (Safe Programs of Order-k) *Safe Transducer Datalog programs of order k express exactly k-ACKTIME, for $k \geq 2$.*

Note that since 2-ACKTIME is PTIME, and 3-ACKTIME is the class of elementary sequence functions, Theorem 4 leads immediately to Corollaries 1 and 2, below. In [6], we show that Corollary 2 also holds for the unrestricted transducer model developed in Section 3, *i.e.*, the model *without* controlled feedback. The two transducer models are therefore equivalent for order 3. That is, in both models, acyclic transducer networks of order 3 express the same class of sequence functions. Finally, Theorem 4 also leads to Corollary 3, which gives a simple characterization of the expressive power of safe Transducer Datalog programs.

Corollary 1 (Safe Programs of Order-2) *Safe Transducer Datalog programs of order 2 express exactly* PTIME *[22].*

Corollary 2 (Safe Programs of Order-3) *Safe Transducer Datalog programs of order 3 express exactly the elementary sequence functions.*

Corollary 3 (Safe Programs of Arbitrary Order) *Safe Transducer Datalog programs express exactly* PR-TIME*, the class of sequence functions computable in primitive-recursive time.*

Corollary 3 is especially interesting, since it shows that the subtransducer mechanism greatly enhances the power of finite state transducers, taking them well beyond the linear time bound of basic transducers.

Acknowledgements: The authors would like to thank several people for their contributions: Steve Cook, Faith Fich and Charles Rackoff for fruitful discussions on the use of machines for computing sequence functions; and Paolo Atzeni and Victor Vianu for numerous comments and suggestions about the language. The first author was partially supported by an operating grant from the Natural Sciences and Engineering Research Council of Canada (NSERC), and the second author was partially supported by MURST and Consiglio Nazionale delle Ricerche (CNR).

References

[1] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *ACM SIGMOD International Conf. on Management of Data*, pages 159–173, 1989.

[2] W. Ackermann. Zum Hilbertschen Aufbau der reellen Zahlen. *Math. Annalen*, 99:118–133, 1928.

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

[4] F. Bancilhon, S. Cluet, and C. Delobel. A query language for the O_2 object-oriented database system. In *Second Intern. Workshop on Database Programming Languages (DBPL'89)*, pages 122–138, 1989.

[5] A. J. Bonner and G. Mecca. Querying sequence databases with Transducers. `http://-poincare.inf.uniroma3.it:8080`, 1997.

[6] A. J. Bonner and G. Mecca. Sequences, Datalog and Transducers. *Journal of Computing and System Sciences*, Special Issue on PODS'95, 1997. To Appear. `http://poincare.inf.-uniroma3.it:8080`.

[7] R. G. G. Cattel. *The Object Database Standard ODMG-93*. Morgan Kaufmann Publishers, San Francisco, CA, 1994.

[8] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Data Bases*. Springer-Verlag, 1989.

[9] A. K. Chandra and D. Harel. Computable queries for relational databases. *Journal of Computing and System Sciences*, 21:333–347, 1980.

[10] Communications of the ACM. Special issue on the Human Genome project. vol. 34(11), November 1991.

[11] 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.

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

[13] N. Goodman. Research issues in Genome databases. Tutorial presented at PODS, 1995.

[14] 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.

[15] 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.

[16] A. Grzegorczyk. Some classes of recursive functions. *Rozprawy Matematyczne*, 4, 1953. Instytut Matematyczne Polskiej Akademie Nauk, Warsaw.

[17] C. Hegelsen and P. R. Sibbald. PALM – a pattern language for molecular biology. In *First Intern. Conference on Intelligent Systems for Molecular Biology*, pages 172–180, 1993.

[18] V. A. Kozmidiadi. On a generalization of finite automata generating a hierarchy analogous to the A. Grzegorczyk's classification of the primitive recursive functions. *Problemi Kibernetiki*, 23:127–170, 1970.

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

[20] G. Mecca. *From Datalog to Sequence Datalog: Languages and Techniques for Querying Sequence Databases.* PhD thesis, Università di Roma "La Sapienza", Dipartimento di Informatica e Sistemistica, 1996. `http://poincare.inf.uniroma3.it:8080`.

[21] G. Mecca and A. J. Bonner. Finite query languages for sequence databases. In *Fifth Intern. Workshop on Database Programming Languages (DBPL'95), Gubbio, Italy.* electronic Workshops in Computing - Springer-Verlag, 1995. `http://www.springer.co.uk/eWiC/Workshops/-DBPL5.html`.

[22] 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. `http://poincare.inf.uniroma3.it:8080`.

[23] A. R. Meyer and D. M. Ritchie. Computational complexity and program structure. *I.B.M. Res. Rep.*, 1817, 1967.

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

[25] H. Rogers, Jr. *Theory of recursive functions and effective computability.* MIT Press, Cambridge, Mass., 1987.

[26] D. B. Searls. String Variable Grammars: a logic grammar formalism for dna sequences. Technical report, University of Pennsylvania, School of Medicine, 1993.

[27] 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.

[28] M. Vardi. The complexity of relational query languages. In *Fourteenth ACM SIGACT Symp. on Theory of Computing*, pages 137–146, 1988.

[29] J. D. Watson et al. *Molecular biology of the gene.* Benjamin and Cummings Publ. Co., Menlo Park, California, fourth edition, 1987.