Querying Sequence Databases with Transducers*

Anthony J. Bonner
University of Toronto
Department of Computer Science
Toronto, Ontario, Canada M5S 3G4
tel: +416 978-7441, fax: +416 978-4765
`bonner@cs.toronto.edu`
`http://www.cs.toronto.edu/~bonner`

Giansalvatore Mecca
D.I.F.A.
Università della Basilicata
via della Tecnica, 3 – 85100 Potenza, Italy
tel: +39 971 474-638, fax: +39 971 56537
`mecca@dia.uniroma3.it`
`http://www.difa.unibas.it/users/gmecca`

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, we recently 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.

*Preliminary portions of this work appeared under the same title in the proceedings of the *Sixth International Workshop on Database Programming Languages (DBPL'97).*

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 [11, 9]. 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. [3, 5]). In many cases, however, the sequence operations are *ad hoc* and are not investigated in a theoretical framework. In other cases, (e.g. [16, 27]) 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.*, [10, 13]). To resolve these problems, we recently developed a query language for sequence data called *Transducer Datalog* [22, 4, 20], which extends classical Datalog [6] 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 [12]. 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, 4]), 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 have shown [22, 4], for instance, that Transducer Datalog is computationally complete, i.e., it can express any computable sequence function.

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, 4, 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, 4]. 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.

This characterization in terms of the Ackermann complexity hierarchy represent the main contribution of the paper.

Note that the Ackermann hierarchy is somewhat similar to the *Grzegorczyk hierarchy* [15] 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 finite (or 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 [10, 13]. 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* [7] 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 $\rho : \Sigma \rightarrow \Sigma$ 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* [7] 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 [31] 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, 4, 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].

Typically [10, 30, 11], 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. Although rather simple, this form of computation over sequences plays an important role in several fields, molecular biology being a prominent example.

Example 3.1 **[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 modeled 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*.[1] This transformation is easily encoded as a transducer, $T_{transcribe}$, in which the input is a DNA sequence, and the output is a RNA sequence. In essence, at each step of its computation, $T_{transcribe}$ reads a nucleotide from the input DNA sequence, and outputs the corresponding ribonucleotide.

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*, ...[2] 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

[1] For simplicity, this example ignores biological complications such as intron splicing [32], even though it can be encoded in Transducer Datalog without difficulty.

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

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.[3] As before, this transformation is easily encoded as a transducer, $T_{translate}$, in which the input is a RNA sequence, and the output is a protein sequence. $T_{translate}$ reads its input sequence, and outputs an amino-acid for each triplet of input symbols that have been consumed. □

3.1 The Computational Model

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 [27]. 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 [10, 30]).

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 $\textsc{out} \in \mathcal{T}^{k-1}$

[3] For simplicity, this example ignores biological complications such as reading frames, ribosomal binding sites, and stop codons [32].

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 $\delta(q, a_1, \ldots, a_n) = \langle q', c_1, \ldots, c_n, \text{OUT} \rangle$. This transition is carried out as follows:

- If OUT is a symbol in Σ, then it is appended to the output sequence; if OUT $= \epsilon$, then the output is unchanged. This takes constant time.
- 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. When the subtransducer starts, its heads 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.
- 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*, $T : (\Sigma^*)^n \to \Sigma^*$, 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 or even exponential in the input lengths, as illustrated below.

3.2 Examples

Example 3.2 **[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

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

Figure 1: Squaring the input.

This table illustrates the computation of transducer T_{square} from Example 3.2 on the input sequence abc. *Each row of the table represents one step of the computation. The column labeled* "input" *shows the position of the input head just before the step, the column labeled* "output" *shows the contents of the output tape just before the step, and the column labeled* "new output" *shows the contents of the output tape just after the step. The column labeled* "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}.*

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 behavior is illustrated in Figure 1, in which the transducer input is abc, and the output is $abcabcabc$. □

Example 3.3 **[Exponential Output]** We describe a generalized transducer, T_{exp}, that given an input of length n, produces an output of length n^{n+1}, i.e., of exponential length. T_{exp} invokes two subtransducers. One is T_{square}, the generalized transducer defined in Example 3.2. The other is T_{mult}, a simple variant of T_{square} that takes two inputs, of length n and m, and produces an output of length nm by appending m times the first input to itself. Both subtransducers have order 2, so T_{exp} has order 3. T_{exp} works as follows:

- As T_{exp} consumes the first symbol on its input, it calls T_{square}, to produce an output of length n^2.
- As T_{exp} consumes the remaining input (of length $n-1$), it calls T_{mult} once for each input symbol. At each call, T_{mult} has two inputs: the original input to T_{exp} (of length n), and the current output of T_{exp}. T_{mult} "multiplies" these two inputs and overwrites the result to the output of T_{exp}. The computation therefore proceeds as follows:
  - the first time T_{exp} calls T_{mult}, the latter has inputs of length n and n^2, and produces an output of length n^3;

– the second time T_{exp} calls T_{mult}, the latter has inputs of length n and n^3, and produces an output of length n^4;
– the i-th time T_{exp} calls T_{mult}, the latter has inputs of length n and n^{i+1}, and produces an output of length n^{i+2};
– the last ($n - 1^{st}$) time that T_{exp} calls T_{mult}, the latter has inputs of length n and n^n, and produces an output of length n^{n+1}.

At this point, T_{exp} has consumed its entire input, so the computation stops.

By iterating this process to deeper levels, dramatically longer outputs – e.g., outputs of hyper-exponential length – can be easily generated. □

The examples above illustrate how subtransducer calls increase the complexity and expressibility of generalized transducers relative to ordinary ones. There are, in fact, basic string transformations – like, for example, reversing the input sequence – that cannot be computed by ordinary transducers, but can be computed by generalized transducers, as shown in the next example.

step	input	output	operation	new output
1	$\underline{a}bc\lhd$	ϵ	call to $T_\#$	$\#\#\#$
2	$a\underline{b}c\lhd$	$\#\#\#$	call to T_{tape}	$\#\#c$
3	$ab\underline{c}\lhd$	$\#\#c$	call to T_{tape}	$\#cb$
4	$abc\underline{\lhd}$	$\#cb$	call to T_{tape}	cba

Figure 2: Computation of $T_{reverse}$ on input abc.

Example 3.4 **[Reverse]** We describe a generalized transducer, $T_{reverse}$, whose output is the reverse of its input. This transducer is of order 2, and has two subtransducers of order 1. The first subtransducer, $T_\#$, is invoked exactly once at the beginning of the entire computation. If the input sequence has length n, then $T_\#$ generates an output sequence $\#\#...\#$ of length n. The other subtransducer, T_{tape}, uses this sequence as a kind of blank tape. As $T_{reverse}$ scans its input from *left to right*, T_{tape} copies the input symbols to the blank tape from *right to left*. Specifically, each time T_{tape} is invoked, its effect is to copy a symbol from the input to the right-most # symbol of the tape. T_{tape} is invoked a total of n times, once for each input symbol. At the end, the entire input has been copied to the tape, but in reverse order. For instance, if the input sequence is $abcd$, then T_{tape} will be invoked four times, and with each invocation, the output of $T_{reverse}$ will change from $\#\#\#\#$ to $\#\#\#a$ to $\#\#ba$ to $\#cba$ to $dcba$.[4]

The subtransducer T_{tape} has two inputs: the input to its parent transducer, $T_{reverse}$, and the current output of $T_{reverse}$. We shall refer to these as the *main input* and the *work tape*, respectively. We shall also refer to each # character as a *blank*. At each invocation, T_{tape} consumes both its inputs, and the effect is to copy a symbol from the main input to the right end of the work tape. To do this, T_{tape} works in five stages:

[4]Since the input is of length n and the computation of $T_{reverse}$ requires $n+1$ steps, we assume that $T_{reverse}$ has a fixed secondary input of length 1. This input is consumed at the very beginning, when $T_\#$ is invoked.

Computation of T_{tape} on inputs abc and $\#\#\#$.

step	input tapes	output tape	commands	output symbol	stage
1	$\underline{a}bc\lhd$ $\underline{\#}\#\#\lhd$	ϵ	$-$ $\rightarrow$	ϵ	1
2	$\underline{a}bc\lhd$ $\#\underline{\#}\#\lhd$	ϵ	$\rightarrow$ $\rightarrow$	$\#$	2
3	$a\underline{b}c\lhd$ $\#\#\underline{\#}\lhd$	$\#$	$\rightarrow$ $\rightarrow$	$\#$	2
4	$ab\underline{c}\lhd$ $\#\#\#\underline{\lhd}$	$\#\#$	$\rightarrow$ $-$	c	4
5	$abc\underline{\lhd}$ $\#\#\#\underline{\lhd}$	$\#\#c$	stop		

Computation of T_{tape} on inputs abc and $\#\#c$.

step	input tapes	output tape	commands	output symbol	stage
1	$\underline{a}bc\lhd$ $\underline{\#}\#c\lhd$	ϵ	$-$ $\rightarrow$	ϵ	1
2	$\underline{a}bc\lhd$ $\#\underline{\#}c\lhd$	ϵ	$\rightarrow$ $\rightarrow$	$\#$	2
3	$a\underline{b}c\lhd$ $\#\#\underline{c}\lhd$	$\#$	$-$ $\rightarrow$	c	3
4	$a\underline{b}c\lhd$ $\#\#c\underline{\lhd}$	$\#c$	$\rightarrow$ $-$	b	4
5	$ab\underline{c}\lhd$ $\#\#c\underline{\lhd}$	$\#cb$	$\rightarrow$ $-$	ϵ	5
6	$abc\underline{\lhd}$ $\#\#c\underline{\lhd}$	$\#cb$	stop		

Computation of T_{tape} on inputs abc and $\#cb$.

step	input tapes	output tape	commands	output symbol	stage
1	$\underline{a}bc\lhd$ $\underline{\#}cb\lhd$	ϵ	$-$ $\rightarrow$	ϵ	1
2	$\underline{a}bc\lhd$ $\#\underline{c}b\lhd$	ϵ	$-$ $\rightarrow$	c	3
3	$\underline{a}bc\lhd$ $\#c\underline{b}\lhd$	c	$-$ $\rightarrow$	b	3
4	$\underline{a}bc\lhd$ $\#cb\underline{\lhd}$	cb	$\rightarrow$ $-$	a	4
5	$a\underline{b}c\lhd$ $\#cb\underline{\lhd}$	cba	$\rightarrow$ $-$	ϵ	5
6	$ab\underline{c}\lhd$ $\#cb\underline{\lhd}$	cba	$\rightarrow$ $-$	ϵ	5
7	$abc\underline{\lhd}$ $\#cb\underline{\lhd}$	cba	stop		

Figure 3: Computations of T_{tape}

1. It consumes one symbol from the work tape (just to put the work-tape head one position ahead of the main-input head).

2. It simultaneously consumes both the work tape and the main input, while writing blanks to the output.

3. When it encounters the first non-blank character on the work tape, it stops consuming the main input, but continues to consume the work tape, while copying it to the output.

4. When the work tape has been completely consumed, it consumes one character from the main input, and copies it to the output.

5. It consumes the rest of the main input, while leaving the output unchanged.

Figure 2 shows the computation of $T_{reverse}$ on input abc. This computation results in three invocations of T_{tape}, which are each illustrated in Figure 3. □

By combining the kind of techniques illustrated in Examples 3.2, 3.3 and 3.4, it is possible to build generalized transducers that compute very complex sequence mappings. In fact, as discussed in the proof of Theorem 3, it is possible to define a generalized transducer that simulates the computation of any Turing machine whose running time is bounded above by the Ackermann function.

Example 3.5 **[Simulating a Turing Machine]** Since generalized sequence transducers always terminate, they are obviously less powerful than Turing machines. However, given any Turing machine, M, and an integer, k, it is possible to construct a generalized transducer, T_M, of order 2 that simulates the first k of moves of M. This transducer has two inputs – one corresponding to the input of the Turing machine, and another corresponding to a unary counter of length $k+2$. At each step of its computation, T_M consumes one symbol of the counter and calls a subtransducer that simulates one move of the Turing machine.

We encode a configuration of the Turing machine as a sequence in a standard way. Suppose that the contents of the machine tape is $b_1 b_2 \ldots b_m$, that the tape head is currently scanning symbol b_i, and that the machine is in state q. We represent this configuration by the sequence $b_1 b_2 \ldots b_{i-1} q b_i b_{i+1} \ldots b_m$. With this representation, we can construct a base transducer that transforms one Turing machine configuration into the next. That is, if the input to the base transducer is a configuration of M, then the output is the next configuration of M.

The simulation works as follows.

- Given an input sequence $\sigma = a_1 a_2 \ldots a_n$ for machine M, transducer T_M first calls a subtransducer that constructs the initial configuration of M, and overwrites it to the output of T_M. This configuration is simply the sequence $q_0 a_1 a_2 \ldots a_n$, where q_0 is the initial state of the Turing machine. This construction is easily carried out by a transducer of order 1.

- T_M then calls a subtransducer k times while consuming the counter. The subtransducer encodes the transition function of machine M. With each call, the subtransducer transforms the output of T_M from one configuration of M to the next.

- Finally, consuming the last counter symbol, T_M calls a base transducer T_{decode} that decodes the current configuration of machine M by removing the machine state. The final output of transducer T_M is therefore the content of M's tape after k steps.

In general, the output of transducer T_M will not correspond to the final output of Turing machine M, which may need more than k steps to terminate, or might not terminate at all. However, suppose we are given a Turing machine which is known to run in time $f(n)$ on inputs of length n, where $f(n)$ is any primitive recursive function (such as n^2 or n^{n+1}). In this case, we can completely simulate the computation of the machine on any input by using a transducer network [4]. In such a network, we first generate a sequence of length $f(n)$ (for example, using transducer T_{square} in Example 3.2, or T_{exp} in Example 3.3 above), and then use this as the counter input for T_M. After $f(n)$ steps, we are guaranteed that the Turing machine terminates, and therefore that the transducer output corresponds to the final output of M. $\square$

It is worth noting that complex sequence mappings are quite important in string databases, and there are several interesting computational problems over sequences that have inherently high complexity. Such problems often arise in molecular biology. For instance, it has been proven [8, 26] that some patterns of biological interest in a DNA sequence are not context-free. The main example is *multiple repeats*, which have important biological functions and are not a context-free language, and which require context-sensitive parsing. This fact greatly increases the complexity of parsing genome sequences,[5] since testing for membership in a context-sensitive language is a PSPACE-complete problem [29, 24]. This problem is closely related to another important problem: testing the equivalence of regular expressions. Variants of this problem have complexity that is even higher than PSPACE. As shown in [29], some of these variants belong to the class of *elementary functions* [24], i.e., functions of hyperexponential time complexity. Other variants, involving negation, have even higher complexity. As shown in the following sections (see Corollaries 1, 2, and 3), generalized transducers capture all of these problems.

3.3 Controlled Feedback

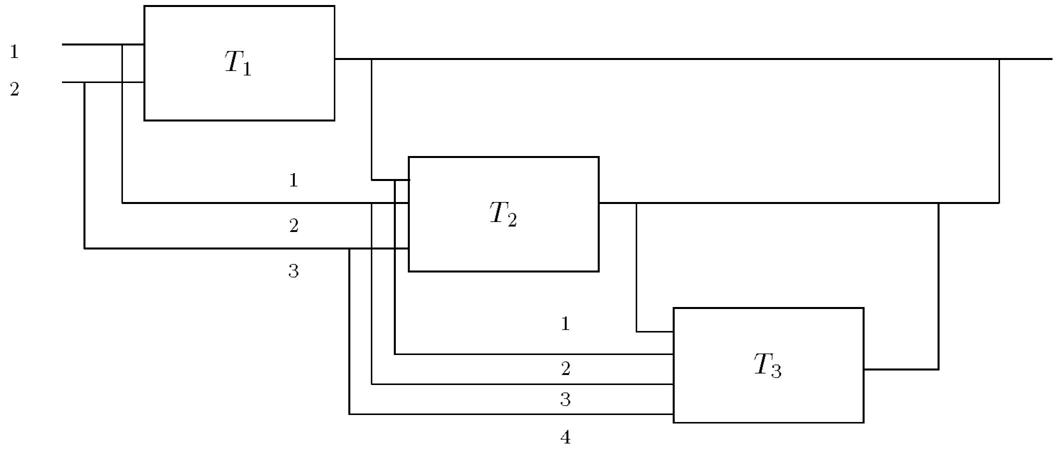
In [22, 4], we established the complexity of generalized transducers of order 2 and 3. This paper extends those results to transducers of arbitrary order. To do this, we focus on a transducer model that is slightly more restrictive than the one described above, but which is identical to it for transducers 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 4(a).

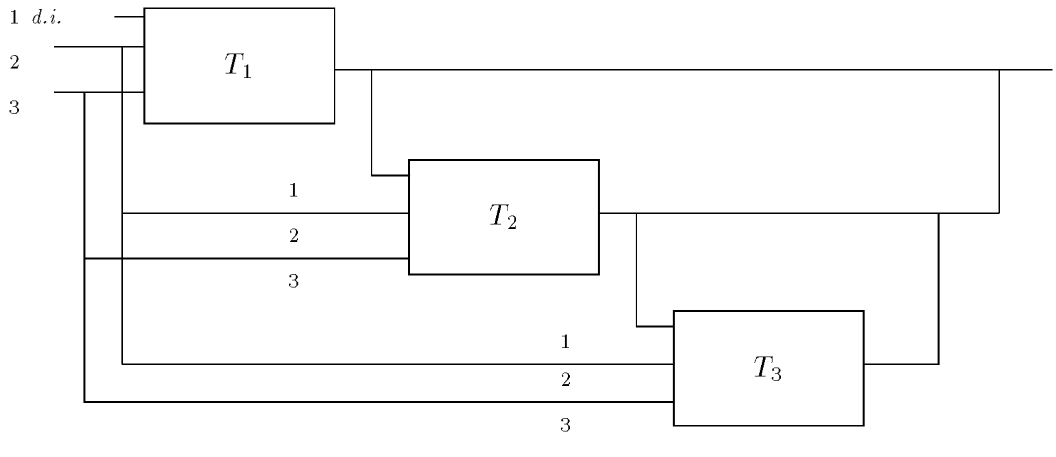
To reduce the amount of feedback and simplify the computation, we now restrict the model. In the restricted transducer model, a subtransducer has access to the output line of the transducer that invoked it (its "parent"), but *not* the output lines of any higher transducers. Thus,

[5]Not surprisingly, the computational linguistics of DNA sequences has become a field of great interest, and several kinds of grammar have been developed that lie somewhere between context-free and context-sensitive [27, 28].

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 4: 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 4(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.

3.4 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:

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

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)[6] *For $k, x, y \geq 0$,*

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

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.*

[6]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].

Proof: In the proof we use the following notation: T_k always represents a generalized transducer of order k; $|T_k(m,n)|$ represents the length of the final output of transducer T_k on inputs m, n; m and n represent the length of the distinguished input and the length of the ordinary input, respectively. Clearly, for a top-level transducer the length of the final output on input of size n is $|T_k(0,n)|$, since the distinguished input of this transducer is empty by definition.

We need to prove that, given a transducer T_k of order k, with inputs of total length n, the length of the final output of T_k is bounded above by the Ackermann's function, $\textsc{ack}_k(\mathcal{O}(n))$, that is:

$$\forall k \geq 1, \quad \exists c \geq 0, n_0 \;\; s.\ t. \;\; \forall n > n_0 \;:\; |T_k(n)| \leq \textsc{ack}_k(cn) \tag{1}$$

Recall from Definition 1 that $\textsc{ack}_k(y)$ is simply a shorthand for $\textsc{ack}(k, y, y)$.

We now define recursively the length of the output of a transducer T_k of order k on inputs m, n – where m is the length of the distinguished input and n the length of the ordinary input. We consider the worst case, that is, the case in which at each step T_k calls a subtransducer; in this case, we have:

$$\begin{cases} |T_1(m,n)| & = \quad m+n, \\ |T_{k+1}(m,n)| & = \quad \begin{cases} |T_k(|T_{k+1}(m-1,n)|,n)| & \text{or} \\ |T_k(|T_{k+1}(m,n-1)|,n)| \end{cases} \end{cases}$$

The definition is easily interpreted: *(i)* the output length of a base transducer is (at most) equal to the sum of the input lengths; *(ii)* on the other side, the output length of a transducer T_{k+1} of order $k+1$ is the output length of a subtransducer T_k, called over the input and the current output of T_{k+1}; two different definitions are needed to distinguish the case in which T_{k+1} consumes the distinguished input to call T_k, from the case in which T_{k+1} consumes the ordinary input.

To prove claim (1), we first prove that the following holds:

$$\forall k \geq 1, m, n > 0, \; \exists c \geq 0 \;\; s.\ t. \;\; |T_k(m,n)| \leq \textsc{ack}(k, m+cn, cn) \tag{2}$$

We use a double induction, with respect to k and with respect to the total input length $m+n$.

– *(Base cases)* If $k = 1$, Equation 2 holds trivially for every value of the input length, $m+n$; in fact, the output length of a base transducer is at most equal to the total input length, that is:

$$|T_1(m,n)| \leq m+n \leq \textsc{ack}(1, m+n, n) = m+2n$$

Consider now the case $m = 0, n = 1$, that is, an input length of 1; in this case, it is easy to see that there exist some c such that, for any k:

$$|T_k(0,1)| \leq 1 \leq \textsc{ack}(k, c, c)$$

– *(Induction)* In the inductive case, there are several inductive hypotheses:

1. We suppose that Equation 2 holds for level $k-1$ and any value of m, n;

2. moreover, we suppose that Equation 2 holds for a total input length of $m+n-1$ and level k; note that an input length of $m+n-1$ may correspond to two different configurations of the input, that is, we have two different inductive hypotheses:

2.a) if we divide the total input length as $(m-1)+n$, then the inductive hypothesis tells us that there exist $c \geq 0$ such that

$$|T_k(m-1,n)| \leq \text{ACK}(k,(m-1+cn),cn).$$

2.b) if we divide the total input length as $m+(n-1)$, then the inductive hypothesis tells us that there exist $c \geq 0$ such that

$$|T_k(m,n-1)| \leq \text{ACK}(k,(m+c(n-1)),c(n-1));$$

We want to prove that Equation 2 holds also for level k and input length $m+n$. In the proof, we make use of the fact that the Ackermann's function $\text{ACK}_k(y)$ grows faster than y, that is:

$$\text{ACK}_k(y)+y = \mathcal{O}(\text{ACK}_k(y)) = \text{ACK}_k(\mathcal{O}(y)) \tag{3}$$

Consider now $|T_k(m,n)|$. According to the recursive definition given above, we can distinguish two cases, based on the symbol consumed by T_k to call the subtransducer.

– *(Case a)*: It T_k consumes the distinguished input we have that:

$$|T_k(m,n)| = |T_{k-1}(|T_k(m-1,n)|,n)|$$

In this case:

$$\begin{array}{rcll}
|T_k(m,n)| & = & |T_{k-1}(|T_k(m-1,n)|,n)| & \text{(by definition)}\\
& \leq & \text{ACK}(k-1,|T_k(m-1,n)|+cn,cn) & \text{(for some } c\text{, by hyp. 1)}\\
& \leq & \text{ACK}(k-1,\text{ACK}(k,m-1+c'n,c'n)+cn,cn) & \text{(for some } c'\text{, by hyp. 2.a)}\\
& \leq & \text{ACK}(k-1,\text{ACK}(k,m+c''n-1,c''n),c''n) & \text{(for some } c''\text{, by 3)}\\
& = & \text{ACK}(k,m+c''n,c''n),\text{ for some } c'' & \text{(by definition);}
\end{array}$$

– *(Case b)*: It T_k consumes the ordinary input we have that:

$$|T_k(m,n)| = |T_{k-1}(|T_k(m,n-1)|,n)|$$

In this case:

$$\begin{array}{rcll}
|T_k(m,n)| & = & |T_{k-1}(|T_k(m,n-1)|,n)| & \text{(by definition)}\\
& \leq & \text{ACK}(k-1,|T_k(m,n-1)|+cn,cn) & \text{(for some } c\text{, by hyp. 1)}\\
& \leq & \text{ACK}(k-1,\text{ACK}(k,m+c'(n-1),c'(n-1))+cn,cn) & \text{(for some } c'\text{, by hyp. 2.b)}\\
& \leq & \text{ACK}(k-1,\text{ACK}(k,m+c''n-1,c''n),c''n) & \text{(for some } c''\text{, by 3)}\\
& = & \text{ACK}(k,m+c''n,c''n),\text{ for some } c'' & \text{(by definition);}
\end{array}$$

Thus the double induction is complete. Now it is easy to see that Equation 1 (with $n_0 > 0$) follows immediately from Equation 2, since for a top-level transducer:

$$|T_k(n)| = |T_k(0,n)|$$

This completes the proof. □

The next theorem shows that the relationship between generalized transducers and the Ackermann function is even closer. In particular, for each k, the function $\text{ACK}(k,x,y)$ 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 $f(w)$.

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* $\textsc{ack}(k, x, y)$*.*

Proof: Let us consider a generalized transducer T_k^{ack} of order k with 2 inputs and 1 output. In order to distinguish the inputs, we call the first input, x, the *counter input*, and the second input, y, the *argument input*. Note that, as for every top-level transducer, T_k^{ack} also has a distinguished input, which is conventionally empty. This transducer behaves as follows:

- if $k = 1$, T_1^{ack} simply appends x and y, producing $x + y$;
- if $k > 1$, then:
  1. for the first y steps of the computation, T_k^{ack} consumes the argument input, y, and copies it to the output;
  2. then, T_k^{ack} uses the counter input, x, to call $x - 1$ times a subtransducer T_{k-1}^{ack}; the fact that T_k^{ack} does only $x - 1$ calls explains the hypothesis $x > 0$;
  3. T_{k-1}^{ack} has access to the inputs of the father, x, y, plus to the current output of the father, y; T_{k-1}^{ack} consumes input x without changing the output; then, it behaves exactly like the father, using the output of the father as counter input and the argument input of the father as argument input.

We now want to derive a recursive definition for the output of transducer T_k^{ack}. To do this, consider the following facts:

- first note that T_k^{ack} has an empty distinguished input, and any other subtransducer invoked in the course of the computation ignores the counter input of the father, that is, it consumes it without changing the output; in this sense, the behavior of these transducers is equivalent to the behavior of a transducer with only two inputs; we omit the third input in the recursive definition of T_k^{ack}, and denote by $|T_k^{ack}(x, y)|$ the size of the output of T_k^{ack} on counter input x and argument input y;
- T_k^{ack} first copies the argument input to the output; then it uses the counter input, x, to call a subtransducer $x - 1$ times; this means that, if $x = 1$, T_k^{ack} does not perform any call, and thus:
$$|T_k^{ack}(1, y)| = y;$$
- finally, at any level of the chain, transducer T_{i+1}^{ack} calls T_i^{ack} using the counter input; whenever this happens, we have that:
$$|T_{i+1}(x, y)| = |T_i(|T_{i+1}(x - 1, y)|, y)|.$$

With this in mind, a double induction on k and x shows that $|T_k^{ack}(x, y)|$ equals $\textsc{ack}(k, x, y)$, for $x > 0$. In fact, we have the following.

Base cases: We have two base cases:

- first, for $k = 1$, we have that, for all $x > 0, y$:

$$|T_1^{ack}(x, y)| = \text{ACK}(1, x, y) = x + y;$$

- second, for $x = 1$, we have that, for all $k > 0, y$:

$$|T_k^{ack}(1, y)| = \text{ACK}(k, 1, y) = y.$$

Induction: it is apparent that the two functions have the same inductive definition; in fact:

$$|T_{i+1}^{ack}(x + 1, y)| = |T_i^{ack}(|T_{i+1}^{ack}(x, y)|, y)|, \quad \forall i \geq 2$$

$$\text{ACK}(i + 1, x + 1, y) = \text{ACK}(i, \text{ACK}(i + 1, x, y), y), \quad \forall i \geq 2$$

This completes the proof. □

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, 4, 20], we developed a query language for sequence databases by incorporating generalized transducers into classical Datalog [6]. 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, 4, 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, $p, q, r, \ldots$, each with 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 *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, 6], 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, 4, 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.

4.2 Examples of Transducer Datalog Programs

Below we give some examples of sequence restructuring in string databases, and show how, by encoding a transducer network in Transducer Datalog, an entire database of sequences can be restructured and queried. Two features of the examples are worth noting: (i) all sequence restructurings take place "inside" transducers; and (ii) the Transducer Datalog programs terminate for every database, since there is no recursion through the construction of new sequences.

Example 4.1 **[Querying String Databases in Transducer Datalog]** Suppose we are given a string database containing a binary relation $R(X,Y)$. We would like to retrieve all tuples in R in which the first argument is the reverse of the second. We can do this in Transducer Datalog with the rules below. Here, $T_{reverse}$ is the generalized transducer of order 2 defined in Example 3.4.

$$\begin{aligned} reverse(X, T_{reverse}(X)) &\leftarrow R(X,Y). \\ answer(X,Y) &\leftarrow R(X,Y),\;\; reverse(X,Y). \end{aligned}$$

Using $T_{reverse}$, the first rule computes pairs of sequences and their reverses. The second rule uses this to select all those pairs in R in which the second sequence is the reverse of the first. □

Example 4.2 **[From DNA to RNA to Protein in Transducer Datalog]** Using Transducer Datalog, the transformation of DNA into protein described in Example 3.1 can easily be converted to a database query. 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 that transforms DNA into RNA, and then RNA into protein. It uses the two sequence transducers of order 1 introduced in Example 3.1: (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. □

Example 4.3 **[Hyperexponential Complexity in Transducer Datalog]** Suppose we are given a relation $R(X)$ of strings. For each X in R, the rules below compute $hyp_i(X,Y)$ where Y is a string whose length is hyperexponential to level i in the length of X. That is, if X has length n, then Y has length $n^{n^{\cdots^{n}}}$ (i times). The rules use a binary transducer T_{raise} of order 3, which is a variant of the unary transducer T_{exp} described in Example 3.3. Given two strings X and Y of length n and m, respectively, the value of $T_{raise}(X,Y)$ is a string of length n^m.

$$
\begin{array}{rcl}
hyp_1(X, T_{exp}(X, X)) & \leftarrow & R(X). \\
hyp_2(X, T_{exp}(X, Y)) & \leftarrow & hyp_1(X, Y). \\
& \cdots & \\
hyp_i(X, T_{exp}(X, Y)) & \leftarrow & hyp_{i-1}(X, Y).
\end{array}
$$

The program implements a transducer network that progressively generates a string of hyper-exponential length by applying the transducer T_{raise} to a string i times. This is done for every string X in relation R. □

Example 4.4 **[Parsing Context-Sensitive Grammars in Transducer Datalog]** Given a relation $R(X)$, we want to select all those strings X in R that belong to some context-sensitive language L, such as $L = \{a^m b^m c^m \mid m \geq 0\}$. The *TransducerDatalog* program below does exactly this by using several of the generalized transducers introduced in Section 3.2. In this program, M is a Turing machine that decides whether a given string is in L. This machine can be easily constructed from the productions of L [17]. Given an input string, σ, we assume that M outputs *yes* if $\sigma \in L$, and *no* otherwise. We use the transducer T_M of order 2 from Example 3.5 to simulate the computation of M. Recall that T_M has two inputs: a sequence σ, and another sequence of length $k+2$. T_M simulates the first k steps of machine M on input σ. Since context-sensitive parsing takes polynomial space (and thus at most exponential time), we can use a transducer of order 3, like T_{exp} in Example 3.3, to generate a string of exponential length, to be used as the second input to T_M. This ensures that T_M will simulate machine M to completion, as described in Example 3.5.

$$
\begin{array}{rcl}
start(X, T_{exp}(X)) & \leftarrow & R(X). \\
decide(X, T_M(X, C)) & \leftarrow & start(X, C). \\
answer(X) & \leftarrow & R(X), \quad decide(X, \text{``}yes\text{''}).
\end{array}
$$

Here, $start(X, C)$ implies that X is in R and that C is exponentially longer than X. $decide(X, D)$ means that X is in R and that machine M makes decision D on input X. Finally, $answer(X)$ means that X is in relation R and in language L. □

5 Safe Transducer Datalog

In [22, 4, 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, 4, 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 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 4.2 is safe since it is non-recursive, and thus its dependency graph contains no cycles. In the following example, all the programs are recursive, and one of them is safe.

Example 5.1 Consider the following three Transducer Datalog programs, P_1, P_2 and P_3:

$$P_1 : \left\{ \begin{array}{rcl} p(X) & \leftarrow & r(X,Y),\ q(Y). \\ q(X) & \leftarrow & r(X,Y),\ p(Y). \\ r(T_1(X),T_2(Y)) & \leftarrow & a(X,Y). \end{array} \right.$$

$$P_2 : \left\{ \begin{array}{rcl} p(T(X)) & \leftarrow & p(X). \end{array} \right.$$

$$P_3 : \left\{ \begin{array}{rcl} q(X) & \leftarrow & r(X). \\ r(T(X)) & \leftarrow & p(X). \\ p(X) & \leftarrow & q(X). \end{array} \right.$$

All three programs are recursive, so their predicate dependency graphs all have cycles. (See Figure 5.) The graphs of P_2 and P_3 have constructive cycles, while the graph of P_1 does not. Thus, P_1 is safe, while P_2 and P_3 are not. □

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 $\text{ACK}_3(y) = y^y$.

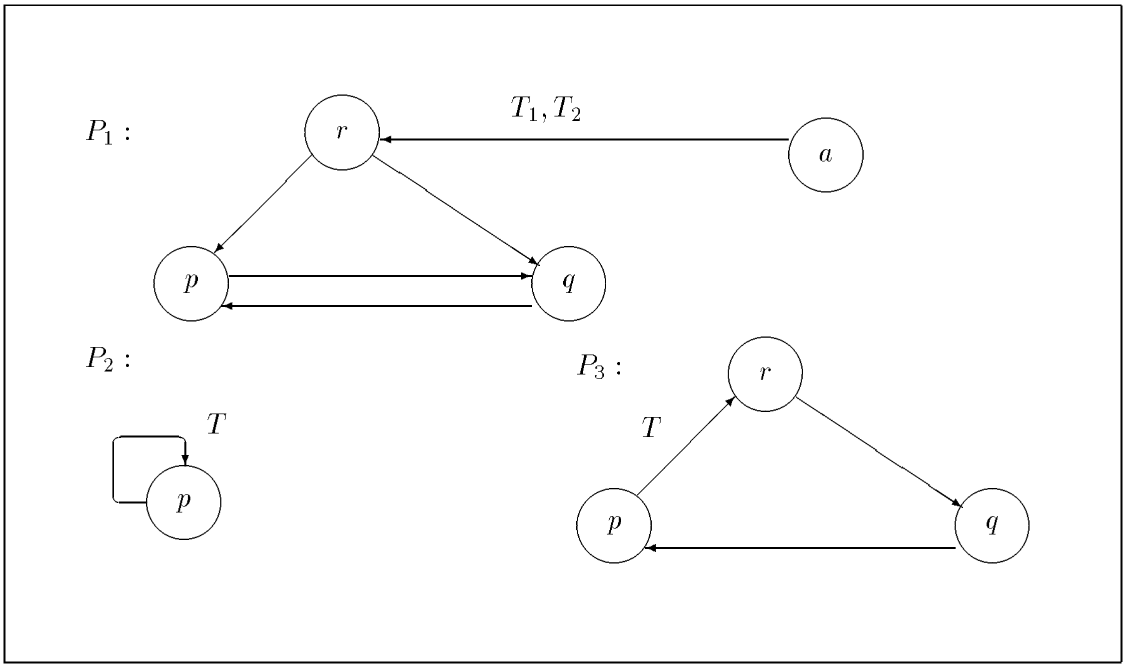


Figure 5: Predicate dependency graphs

These three graphs correspond to the three programs in Example 5.1. Constructive edges are labeled by transducer names.

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 $\mathcal{PR}$ is the set of primitive recursive functions. The Ackermann time hierarchy coincides with this time class; that is $\text{ACK-HIER} = \text{PR-TIME}$. These relationships are summarized in Figure 6.

Note that the Ackermann hierarchy is somewhat similar to the *Grzegorczyk hierarchy* [15] of primitive recursive functions. However, the latter is based on a different set of initial func-

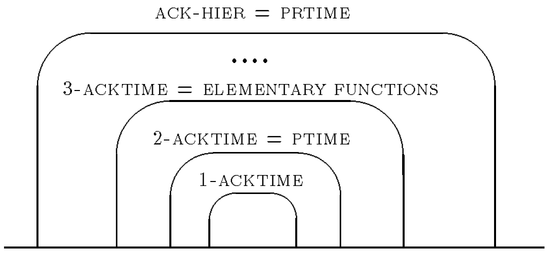


Figure 6: The Ackermann's hierarchy of complexity classes

tions 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 3 (Safe Programs of Order-k) *Safe Transducer Datalog programs of order k express exactly k-ACKTIME, for $k \geq 2$.*

Proof: We shall first prove the expressibility lower bound and then the complexity upper bound.

– *(Lower Bound)* To prove the lower bound, we need to prove that any function in k-ACKTIME, $k \geq 2$, can be expressed by a safe Transducer Datalog program of order k. Any such sequence function is computed by some single-tape Turing machine, M, that runs in time $\text{ACK}_k^{(i)}(n)$, for some i, on inputs of length n. We shall simulate the computation of this latter machine using a safe Transducer Datalog program, as follows. The first rules in the program are used to compose a network consisting of a sequence of $i+1$ transducers. The first i transducers have a single input each, and together they generate a string of length $\text{ACK}_k^{(i)}(n)$. The final transducer uses this string as a "counter" to simulate the computation of machine M. This transducer essentially correspond to transducer T_M introduced in Example 3.5.

Remember that we encode a configuration of the Turing machine as a sequence in a standard way. Suppose that the contents of the machine tape is $b_1 b_2 \dots b_m$, that the tape head is currently scanning symbol b_i, and that the machine is in state q. We represent this configuration by the sequence $b_1 b_2 \dots b_{i-1} q b_i b_{i+1} \dots b_m$. With this representation, we can construct a base transducer that transforms one Turing machine configuration into the next. That is, if the input to the transducer is a configuration of M, then the output is the next configuration of M.

To compute a sequence function, our program performs three tasks: it constructs the initial configuration of the Turing machine, it simulates $\text{ACK}_k^{(i)}(n)$ Turing-machine steps, and it extracts the output sequence from the final configuration. These tasks are carried out as follows:

- Given an input sequence $a_1 a_2 \dots a_n$, an order 1 transducer constructs the initial configuration of the Turing machine, $q_0 a_1 a_2 \dots a_n$, where q_0 is the initial state of the Turing machine.

- Given the input sequence (of length n), a series of transducers generates a sequence, $\sigma_{counter}$, of length $\text{ACK}_k^{(i)}(n)$. We use this sequence to count time. It is easily generated using i transducers of order k.

- A transducer T_M of order 2 simulates the computation of Turing machine M. This transducer has three inputs: the counter sequence $\sigma_{counter}$, the initial configuration of M, and a string of length 1. T_M first moves the initial configuration of M from its input to its output. It then repeatedly calls a subtransducer while consuming $\sigma_{counter}$. It thus calls the subtransducer $\text{ACK}_k^{(i)}(n)$ times. The subtransducer encodes the transition function of machine M. With each call, the subtransducer transforms the output of T_M from one configuration of M to the next. When T_M has completely consumed $\sigma_{counter}$, its output represents the final configuration of machine M.

- Finally, consuming the single character on its third input, T_M calls a base transducer T_{decode} that decodes the final configuration of machine M. To do this, we can assume the final configuration has the form $c_1 c_2 \dots c_m q_z xxxxxx \dots xx$, where $c_1 c_2 \dots c_m$ is the output sequence computed by M, q_z is the final state of M, and each x denotes a blank tape character. T_{decode} simply removes the blanks and the machine state, leaving the output of machine M.

This proves the claim.

– *(Upper Bound):* The upper bound follows from Theorem 1. To see this, consider a safe program of order k. It is possible to see that the least fixpoint of the program is finite. The proof is based on the strongly connected components of the predicate dependency graph of P. Recall that any two nodes in a strongly connected component have a cycle passing through them. If a node in the graph belongs to no cycle, then the node is treated as a singleton component. By hypothesis, the predicate dependency graph of P has no constructive cycles. Constructive edges thus occur only between different components.

For any graph, the relation *"there is an arc from component i to component j"* is finite and acyclic. We can therefore linearize the components by a topological sort. That is, if there are k strongly connected components in a graph, then we can assign a distinct integer $i \in \{1, \dots, k\}$ to each component in such a way that there is an arc from component i to component j iff $i < j$. Let us denote the linearized components by $\mathcal{N}_1, \dots, \mathcal{N}_k$. The linearization induces a *stratification* on the program P, where the i^{th} stratum consists of those rules in P that define predicates in $\mathcal{N}_i$. We let P_i denote the i^{th} stratum of P. The P_i are disjoint, and $P = P_1 \cup P_2 \cup \dots \cup P_k$.

Because Transducer Datalog has a fixpoint semantics based on a monotonic and continuous T-operator, the exact order in which rules are applied does not matter. Any order will converge to the least model of P and DB, so long as the rules are applied until the extent of each predicate stops growing, just as in classical Datalog and Horn logic programming. In addition, because P is guarded, the extent of a predicate defined in P_i depends only on the rules in $P_1 \cup \dots \cup P_i$. We can therefore apply the rules in a bottom-up fashion, one stratum at a time. That is, we can first apply the rules in P_1 to saturation, then the rules in P_2, then those in P_3, etc. Formally, given a database DB, we define a sequence of minimal models $\mathcal{M}_1, \ldots, \mathcal{M}_k$, $\mathcal{M}_{k+1}$ as follows:

$$\begin{aligned}\mathcal{M}_1 &= \text{DB} \\ \mathcal{M}_{i+1} &= T_{P_i,\mathcal{M}_i} \uparrow \omega \quad \textit{for} \;\; 1 \leq i \leq k\end{aligned}$$

$\mathcal{M}_{i+1}$ is the minimal model of P_i and $\mathcal{M}_i$. Once $\mathcal{M}_{i+1}$ has been computed, the extent of each predicate defined in $P_1 \cup \cdots \cup P_i$ is completely materialized. Thus, $\mathcal{M}_{i+1}$ is also the minimal model of $P_1 \cup \cdots \cup P_i$ and DB. In particular, $\mathcal{M}_{k+1}$ is the minimal model of P and DB.

The rules in P_i are of two types: constructive and unconstructive. In a constructive rule, each predicate symbol in the body is defined at a *lower* stratum, that is, in $P_1 \cup \cdots \cup P_{i-1}$. The extent of these predicates is completely materialized in $\mathcal{M}_i$. In addition, each constructive rule, like the entire program, is guarded. Thus, each sequence variable can bind only to sequences in $\mathcal{M}_i$. Consequently, the constructive rules need only be applied *once*, since additional applications will not infer any new atoms. After this, the unconstructive rules can be applied repeatedly until the minimal model of P_i and $\mathcal{M}_i$ is reached. Formally, if P_i^c and P_i^u denote respectively the constructive and unconstructive rules in P_i, then

$$\mathcal{M}_{i+1} \; = \; T_{P_i^u,\mathcal{M}_i'} \uparrow \omega \qquad \textit{where} \quad \mathcal{M}_i' \; = \; T_{P_i^c,\mathcal{M}_i}(\mathcal{M}_i)$$

Only constructive rules can expand the active domain by generating new sequences. Thus, $\mathcal{M}_{i+1}$ and $\mathcal{M}_i'$ have the same active domain, and all new sequences are created in computing $\mathcal{M}_i'$ from $\mathcal{M}_i$.

A simple inductive argument shows that the size of the minimal model, $\mathcal{M}_{k+1}$ is finite. In fact, we know that the size of $\mathcal{M}_1 = \text{DB}$ is finite. It is also easy to see that, for any $i \geq 1$, the size of $\mathcal{M}_{i+1}$ is finite if the size of $\mathcal{M}_i$ is finite. To see this, we first show that the size of $\mathcal{M}_i'$ is finite if the size of $\mathcal{M}_i$ is finite. In fact, let n_i' and n_i denote the sequences in $\mathcal{M}_i'$ and $\mathcal{M}_i$, respectively. Also, let l_i' denote the maximum length of any sequence in the extended active domains of $\mathcal{M}_i'$. Note now that l_i' is finite, since the active domain $\mathcal{M}_i'$ consists of the sequences in $\mathcal{M}_i$, plus the sequences created by transducer terms in P_i^c, plus all their contiguous subsequences. Since transducers always produce finite-length outputs on finite inputs, the longest sequence created by any transducer in P_i^c is finite.

To bound n_i', let $T(s_1, \ldots, s_k)$ be a transducer term in P_i^c. As described above, each sequence variable will bind only to sequences occurring in $\mathcal{M}_i$. Each sequence term s_i will thus bind only to sequences in $\mathcal{M}_i$. Thus, the number of input tuples to this transducer term is at most n_i^k; so the number of output sequences is also at most n_i^k. The transducer term thus contributes at most a finite and polynomial number of sequences to the active domain of $\mathcal{M}_i'$. This is true for each transducer term in P_i^c. This means that $\mathcal{M}_i'$ has finite size. Since the size of $\mathcal{M}_{i+1}$ equals the size of $\mathcal{M}_i$, also $\mathcal{M}_{i+1}$ is finite. This completes the induction.

Let us now study the size of the least fixpoint. Recall that, for any $i \geq 1$, $\mathcal{M}_{i+1}$ and $\mathcal{M}_i'$ have the same active domain, and all new sequences are created in computing $\mathcal{M}_i'$ from $\mathcal{M}_i$. We shall show that, if s is the size of $\mathcal{M}_i$, than the size of $\mathcal{M}_i'$ is at most $\text{ACK}_k(s)$. Recall that the number of sequences grows polynomially going from one stratum to the next. Let l_i' and l_i denote the maximum lengths of any sequence in the active domains of $\mathcal{M}_i'$ and $\mathcal{M}_i$, respectively. To bound l_i', observe that the active domain of $\mathcal{M}_i'$ consists of the sequences in $\mathcal{M}_i$, plus the sequences created by transducer terms in P_i^c. By hypothesis, each transducer in P_i^c has order at most k. Such transducers create sequences of length at most $\text{ACK}_k(n)$, where n is the length of the input. Thus, $l'_i = \text{ACK}_k(l_i)$. Each stratum of program P thus increases the size of the minimal model by at most a factor of $\text{ACK}_k(n)$. Thus, since the number of strata is fixed, the size of the minimal model of P and DB is at most $\text{ACK}_k^{(p)}(n)$, for some j, if n is the size of DB.

Suppose that P expresses a sequence function, that is, a mapping from a database, $\text{DB} = \{\{in(\sigma)\}\}$, containing a single sequence of length n, to a relation, $\{out(\sigma')\}$, containing a single sequence. Thus, the minimal model is computed after a number of applications of $T_{P,\text{DB}}$ which is at most $\text{ACK}_k^{(p)}(n)$. Moreover, each application of $T_{P,\text{DB}}$ takes at most time $\text{ACK}_k^{(j)}(n)$. This proves that the complexity of the computed function is in k-ACKTIME with respect to the input sequence size. □

Based on Theorem 3 above, we can establish upper bounds on the complexity of the sequence functions computed by the Transducer Datalog programs in Section 4.2. Since all the programs are safe, we can say that the functions computed by the program in Example 4.2 have complexity in 1-ACKTIME (the program implements a transducer network of order 1); the program in Example 4.1 has polynomial complexity (it uses one transducer of order 2); the programs in Examples 4.3 and 4.4 have elementary complexity (they implement networks of order 3).

Note that since 2-ACKTIME is PTIME, and 3-ACKTIME is the class of elementary sequence functions, Theorem 3 leads immediately to Corollaries 1 and 2, below. In [4], 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 3 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] 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.

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

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

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

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

[8] J. Collado Vides. The search for a grammatical theory of gene regulation is formally justified by showing the inadequacy of context-free grammars. *Computer Application in the Biosciences (CABIOS)*, 7(3):321–326, 1991.

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

[10] 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.

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

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

[13] 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.

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

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

[16] 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.

[17] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages and computation.* Addison Wesley Publ. Co., Reading, Massachussetts, 1979.

[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://www.difa.unibas.it/users/gmecca`.

[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://www.difa.unibas.it/users/gmecca`.

[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. Representing genetic information with formal grammars. *Proc of the Nat. Conf. of the American Association for Artificial Intelligence*, 7:386–391, 1988.

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

[28] D. B. Searls. The computational linguistics of biological sequences. In L. Hunter, editor, *Artificial Intelligence and Molecular Biology*, pages 47–120. AAAI Press, 1994.

[29] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. In *Proc. of 5th ACM Symposium on the Theory of Computing (STOC'73)*, pages 1–9, 1973.

[30] 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.

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

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