UNIVERSITÀ DEGLI STUDI DI ROMA TRE
Dipartimento di Discipline Scientifiche
Via della Vasca Navale, 84 – 00146 Roma, Italy.

Sequences, Datalog and Transducers

ANTHONY J. BONNER†, GIANSALVATORE MECCA‡

RT-INF-14-96 **Giugno 1996**

† Department of Computer Science
University of Toronto
Toronto, Canada

‡ Università della Basilicata,
D.I.F.A. – via della Tecnica, 3
85100 Potenza, Italy

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

ABSTRACT

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

Contents

1 Introduction

Sequences represent an important feature of *Next Generation Database Systems* [3, 27]. In recent years, new applications have arisen in which the storage and manipulation of sequences of unbounded length is a crucial feature. A prominent example is *genome databases*, in which long sequences representing genetic information are stored, and sophisticated pattern matching and restructuring facilities are needed [11]. These new applications have led to the introduction of sequence types in recent data models and query languages (e.g. [2, 4, 5, 28]). In many cases, however, queries over sequences are described only by means of a set of pre-defined, *ad hoc* operators, and are not investigated in a theoretical framework. In other cases, (e.g. [19, 25]) query languages concentrate on *pattern extraction* capabilities and do not consider *sequence restructurings*. Although pattern recognition is a fundamental feature of any language for querying sequences, sequence restructurings are equally important. For example, in genome databases, one frequently needs to concatenate sequences together, to splice out selected subsequences, and to compute the reverse of a sequence. In addition, 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 declarative query language with arbitrary procedures for efficiently executing sequence operations [14].

Sequence data presents interesting challenges in the development of query languages. For instance, the query language should be *expressive* both in terms of pattern matching and sequence restructurings. At the same time, it should have a natural syntax and a clear semantics. Finally, it should be *safe*. Safety and finiteness of computations is a major concern when dealing with sequences, since by growing in length, sequences can easily become infinite, even when the underlying alphabet—or domain—is finite. This means that, unlike traditional query languages, sequence queries can end up in nonterminating computations.

To address this problem, we propose a new logic called *Sequence Datalog* for reasoning about sequences. Sequence Datalog has both a clear declarative semantics and an operational semantics. The semantics are based on fixpoint theory, as in classical Logic Programming [21].

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

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

1.1 Background

Sequence query languages have been recently investigated in the context of functional and algebraic programming [10, 18], and some sophisticated and expressive languages have been proposed. Great care has been devoted to the development of *tractable* languages, that is, languages whose complexity is in PTIME.

In [10], for example, a functional query language for nested lists is obtained by introducing a new form of structural recursion called *list traversal*, in which two different lists are used. One list is used to control the number of iterative steps, while the other list can be modified at each iteration. This mechanism ensures that query answers are finite. The language is then restricted to express exactly the class of PTIME mappings over nested lists.

A similar result is reported in [18], in which an algebra for *partially ordered multi-sets* (POMSETS) is defined. The algebra is obtained by extending the *bag algebra* of [17] with new operators to handle arbitrary POMSETS, plus an iterator for performing structural recursion [8]. The authors define a tractable fragment of the language by restricting the class of structures allowed and introducing a form of *bounded iterator* to prevent exponential growth of query answers. When restricted to relational structures, the tractable language is shown to be equivalent to first-order logic augmented with operators for counting and computing least fixpoints [15]. When restricted to lists, that is, to totally ordered structures, the language expresses exactly the class of PTIME mappings over lists.

The characterizations of PTIME reported in [10, 18] represent fundamental contributions to the manipulation of lists in databases. However, in our opinion, these languages lack the elegance and simplicity of relational algebra, especially when applied to lists. For this reason, we explore a different approach to the problem, an approach based on *logic* programming, instead of *functional* programming. In particular, we develop an extension of Datalog that is a simple yet powerful tool for manipulating sequences in databases. We first propose a simple intuitive syntax for the language, and develop a clear logical semantics. We then establish its data complexity, expressive power, and finiteness properties.

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

In [12, 30], for example, an *extended relational model* is defined, where each relation is a set of tuples of sequences over a fixed alphabet. A *sequence logic* [12] is then developed based on the notion of *rs-operations*. Each *rs-operation* is either a *merger* or an *extractor*. Intuitively, given a set of patterns, an associated *merger* uses the patterns to "merge" a set of sequences. An *extractor* "retrieves" subsequences of a given sequence. The authors introduce the notion of a *generic a-transducer* as the computational counterpart of rs-operations. Based on their sequence logic, two languages for the extended relational model are defined, called the *s-calculus* and the *s-algebra*. The s-calculus allows for *unsafe* queries, that is, queries whose semantics is not computable. A *safe* subset of the language is then defined and proven equivalent to the s-algebra. Unfortunately, this safe sublanguage cannot express many queries for which the length of the result depends on the database. These include natural queries such as the *reverse* and the *complement* of a sequence.

This problem is partially solved by the *alignment logic* of [16], an elegant and expressive first-

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

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

1.2 Overview of the Language

This paper builds on the works of [12, 16, 24] to propose a query language that is safe, expressive and has a clear declarative and operational semantics. This language, called *Sequence Datalog*, is a Horn-like logic with special, interpreted function symbols that allow for *structural recursion* over sequences.

The goal of this paper is to develop and investigate different forms of structural recursion over sequences, especially forms that guarantee terminating computations. At the same time, we wish to construct new sequences and restructure old ones. To meet these two goals, programs in *Sequence Datalog* have two kinds of function term: *indexed terms* to extract subsequences, and *constructive terms* to concatenate sequences. An indexed term has the form $s[n_1{:}n_2]$, and is interpreted as a subsequence of s. A constructive term has the form $s_1 \bullet s_2$, and is interpreted as the concatenation of s_1 and s_2, which is a supersequence of both.

Example 1.1 **[Extracting Subsequences]**
The following rule extracts all prefixes of all sequences in relation r:

$$prefix(X[1{:}N]) \leftarrow r(X)$$

For each sequence, X, in r, this rules says that a prefix of X is any subsequence starting with the first element and ending with the N-th element, so long as $N \geq 0$ and no longer than the length of X. □

The universe of sequences over an alphabet, Σ, is infinite. Thus, to keep the semantics of programs finite, we do not evaluate rules over the entire universe, Σ^*. Instead, we introduce a new active domain for sequence databases, called the *extended active domain*. This domain contains all the sequences occurring in the database, plus all their *subsequences*.[1] Substitutions

[1] In this paper, we always refer to *contiguous subsequences*, that is, subsequences specified by a start and end position in some other sequence. Thus, *bcd* is a contiguous subsequence of *abcde*, whereas *bd* is not.

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

The extended active domain is not fixed during query evaluation. Instead, whenever a new sequence is created (by the concatenation operator, $\bullet$), the new sequence—and its subsequences—are added to the extended active domain. The fixpoint theory of Sequence Datalog provides a declarative semantics for this apparently procedural notion. In the fixpoint theory, the extended active domain of the least fixpoint may be larger than the extended active domain of the database. In the database, it consists of the sequences in the database and all their subsequences. In the least fixpoint, it consists of the sequences in the database and any new sequences created during rule evaluation, and all their subsequences.

Example 1.2 **[Concatenating Sequences]**
The following rule constructs all possible concatenations of sequences in relation r:

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

This rule takes any pair of sequences, X and Y, in relation r, concatenates them, and puts the result in relation *answer*, thereby adding new sequences to the extended active domain. □

Compared to Datalog with function symbols, or Prolog, two differences are apparent. The first is that Sequence Datalog has no uninterpreted function symbols, so it is not possible to build arbitrarily nested structures. On the other hand, Sequence Datalog has a richer syntax than the $[Head|Tail]$ list constructor of Prolog. This richer syntax is motivated by a natural distinction between two types of recursion, one safe and the other unsafe. Recursion through construction of new sequences is *inherently unsafe* since it can create longer sequences, which can make the active domain grow indefinitely. On the other hand, structural recursion over existing sequences is *inherently safe*, since it only creates shorter sequences, so that growth in the active domain is bounded. Typically, languages for list manipulation do not discriminate between these two types of recursion. Sequence Datalog does: constructive recursion is performed using constructive terms, of the form $X \bullet Y$, while structural recursion is performed using indexed terms, of the form $X[n_1{:}n_2]$. The next example illustrates both kinds of recursion.

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

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

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

The formulas $rep_1(X, Y)$ and $rep_2(X, Y)$ both mean that X has the form Y^n for some n. However, rep_2 has an infinite semantics, while rep_1 has a finite semantics. The rules for rep_1 do *not* create new sequences, since they do not use the concatenation operator, $\bullet$. Instead, these

[2] Note that the size of the extended active domain is at most quadratic in the size of the database domain. In fact, including the empty sequence, the number of different contiguous subsequences of a sequence of length k is at most $1 + \binom{k}{1} + \binom{k}{2}$, that is, $\frac{k(k+1)}{2} + 1$.

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

rules retrieve all sequences in the extended active domain that fit the pattern Y^n. They try to recursively "chop" an existing sequence into identical pieces. We call this *structural recursion.* In contrast, the rules for rep_2 *do* create new sequences. They produce sequences of the form Y^n by recursively concatenating each sequence in the domain with itself. We call this *constructive recursion.* Structural recursion is always safe, leading to a finite least fixpoint. In this example, constructive recursion is unsafe, leading to an infinite least fixpoint. □

1.3 Safety and Transducers

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

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

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

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

1.4 Expressibility

Two different kinds of transformation can be used to measure the expressive power of a sequence query language. The first kind is a *sequence query*, which is a straightforward generalization of relational query, *i.e.*, a mapping from sequences databases to sequence relations. The second kind of transformation is a *sequence function*, which has no counterpart in traditional database languages. Following [10], we define a *sequence function* to be a mapping that takes a sequence as input and returns a sequence as output. Sequence functions can be thought of as queries from a database, $\{input(\sigma_{in})\}$, containing a single sequence tuple, to a relation, $\{output(\sigma_{out})\}$, containing a single sequence tuple. Expressibility results formulated in terms of sequence functions are especially meaningful for sequence query languages, since they provide a clear characterization of the power of the language to manipulate sequences: a sequence query language cannot express complex queries over sequence databases if it cannot express complex sequence functions. In short, function expressibility is necessary for query expressibility.

In this paper, we characterize the expressive power of subsets of Sequence Datalog in terms of sequence functions. We prove expressibility results for both the class of PTIME sequence functions and the class of *elementary* sequence functions [23]. PTIME expressibility results were first reported in [10, 18] with respect to list-based databases of complex-objects. Here, we extend those results to any hyper-exponential time function. In [20], expressibility results for intermediate types were proved in terms of hyper-exponential time. We extend these results to sequences.

2 Preliminary Definitions

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

Let Σ be a countable set of symbols, called the *alphabet*. Σ^* denotes the set of all possible sequences over Σ, including the *empty sequence*, ϵ. $\sigma_1\sigma_2$ denotes the concatenation of two sequences, $\sigma_1, \sigma_2 \in \Sigma^*$. $\text{LEN}(\sigma)$ denotes the length of sequence σ, and $\sigma(i)$ denotes its i-th element. With an abuse of notation, we blur the distinction between elements of the alphabet and 1-ary sequences.

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

We now describe an extension of the relational model, in the spirit of [12, 16]. 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. Similarly, a *sequence function* [10] is a partial mapping from Σ^* to itself. A sequence function f is *computable* if it is partial recursive. Usually, a notion of *genericity* [9] is introduced for queries. The notion can be extended to sequence queries in a natural way. We say that a sequence query Q is *computable* [9] if it is generic and partial recursive.

In this paper, we address the complexity of sequence functions, and the data complexity [29] of sequence queries. Given a sequence function, f, the *complexity* of f is defined in the usual way, as the complexity of computing $f(\sigma_{in})$, 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 query language, L, is *complete* in the complexity class C if: (i) each query expressible in L has complexity in C; (ii) there is a query, Q, expressible in L such that computing $Q(\text{DB})$ is a complete problem for the complexity class C.

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

3 Sequence Datalog

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

Example 3.1 **[Pattern Matching]**
Suppose we are interested in sequences of the form $a^n b^n c^n$ in relation r. The query $answer(X)$ retrieves all such sequences, where the predicate $answer$ is defined by the following clauses, where ϵ is the empty sequence:

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

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

Example 3.2 **[Sequence Restructuring]**
Suppose r is a unary relation containing a set of binary sequences. We want to generate the reverse of every sequence in r. *e.g.*, The reverse of 110000 is 000011. The query $answer(Y)$ generates these sequences, where the predicate $answer$ is defined by the following clauses:

$$
\begin{array}{lcl}
answer(Y) & \leftarrow & r(X),\ reverse(X,Y). \\
reverse(\epsilon,\epsilon) & \leftarrow & true. \\
reverse(X[1{:}N+1], X[N+1] \bullet Y) & \leftarrow & r(X),\ reverse(X[1{:}N], Y).
\end{array}
$$

In this program, the sequences in r act as input for the predicate $reverse(X,Y)$. The third clause recursively scans each input sequence, X, while constructing an output sequence, Y. Each bit in the input sequence is appended to the other end of the output sequence. The clause generates the reverse of each prefix of each sequence in r. The first clause then retrieves the reverse of only the sequences in r. □

The rest of this section presents the syntax and semantics of Sequence Datalog.

3.1 Syntax

SequenceDatalog has two interpreted function symbols for constructing complex terms—one for concatenating sequences and one for extracting subsequences. Intuitively, if X and Y are sequences, then the term $X \bullet Y$ denotes the concatenation of X and Y. Likewise, if I and J are integers, then the term $X[I:J]$ denotes the subsequence of X extending from position I to position J.

To be more precise, the language of terms uses four countable, disjoint sets: the set of non-negative integers, $0, 1, 2, ...$; a set of constant symbols, $a, b, c, ...$, called the *alphabet* and denoted Σ; a set of variables, $R, S, T, ...$, called *sequence variables* and denoted V_Σ; and another set of variables, $I, J, K, ...$, called *index variables* and denoted V_I. A constant sequence (or *sequence*, for short) is an element of Σ^*. From these sets, we construct two kinds of term as follows:

- *index terms* are built from integers, index variables, and the special symbol *end*, by combining them recursively using the binary connectives $+$ and $-$. Thus, if N and M are index variables, then 3, $N+3$, $N-M$, $end-5$ and $end-5+M$ are all index terms;
- *sequence terms* are built from constant sequences, sequence variables and index terms, by combining them recursively into *indexed terms* and *constructive terms*, as follows:
  - If s is a sequence variable and n_1, n_2 are *index terms*, then $s[n_1{:}n_2]$ is an *indexed sequence term*. n_1 and n_2 are called the *indexes* of s. As a shorthand, each sequence term of the form $s[n_i{:}n_i]$ is written $s[n_i]$.
  - If s_1, s_2 are *sequence terms*, then $s_1 \bullet s_2$ is a *constructive sequence term*.

  Thus, if S_1 and S_2 are sequence variables, and N is an index variable, then $S_1[4]$, $S_1[1{:}N]$ and $ccgt \bullet S_1[1{:}end-3] \bullet S_2$ are all sequence terms.

As in most logics, the language of formulas for *SequenceDatalog* includes a countable set of predicate symbols, $p, q, r, ...$, where each predicate symbol has 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 [21]. The head and body of a clause, γ, are denoted $\text{HEAD}(\gamma)$ and $\text{BODY}(\gamma)$, respectively. A clause that contains a constructive term in its head is called a *constructive clause*. A Sequence Datalog *program* is a set of Sequence Datalog

clauses in which constructive terms (terms of the form $s_1 \bullet s_2$) may appear in clause heads, but not in clause bodies.

We say that a variable, X, is *guarded* in a clause if X occurs in the body of the clause as an argument of some predicate. Otherwise, we say that X is *unguarded*. For example, X is guarded in $p(X[1]) \leftarrow q(X)$, whereas it is unguarded in $p(X) \leftarrow q(X[1])$. Because of the active domain semantics, variables in Sequence Datalog clauses need not be guarded.

A term is *ground* if it contains no variables. The set $\mathcal{U}$ containing all the ground terms over Σ is called the *herbrand universe* associated with Σ. It is worth noting that the herbrand universe contains Σ^* and is thus an infinite set.

A finite set of predicate symbols are identified as *base predicates*. A *database* is a finite set of atoms made from base predicates. The semantics of Sequence Datalog does not depend on the notion of base predicates, but some results on query expressibility do.

3.2 Substitutions

A substitution, θ, is a mapping that associates a sequence with each sequence variable in V_Σ, and an integer with each index variable in V_I. Substitutions can be extended to partial mappings on sequence and index *terms* in a straightforward way. Because these terms are interpreted, the result of a substitution is either a sequence or an integer.

Intuitively, $\theta(s[n_1{:}n_2])$ is the contiguous subsequence of $\theta(s)$ extending from position $\theta(n_1)$ to position $\theta(n_2)$. Here, terms such as $s[n+1{:}n]$ are conveniently interpreted as the empty sequence, ϵ. For example,

s	$\theta(s)$
$uvwxy[3:6]$	undefined
$uvwxy[3:5]$	wxy
$uvwxy[3:4]$	wx
$uvwxy[3:3]$	w
$uvwxy[3:2]$	ϵ
$uvwxy[3:1]$	undefined

If the special index term *end* appears in the sequence term $s[n_1{:}n_2]$, then *end* is interpreted as the length of $\theta(s)$. Thus, $\theta(s[n{:}end])$ is a suffix of $\theta(s)$. Finally, $\theta(s_1 \bullet s_2)$ is interpreted as the concatenation of $\theta(s_1)$ and $\theta(s_2)$. The rest of this subsection makes these ideas precise.

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

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

- if s is a sequence in Σ^* or an integer, then $\theta(s) = s$;
- in the context of the sequence term $S[n{:}end]$, we define $\theta(end) = \text{LEN}(\theta(S))$;
- if n_1 and n_2 are index terms, then $\theta(n_1 + n_2) = \theta(n_1) + \theta(n_2)$ and $\theta(n_1 - n_2) = \theta(n_1) - \theta(n_2)$;

- if s is a sequence term of the form $S[n_1{:}n_2]$, then $\theta(s)$ is defined iff $1 \leq \theta(n_1) \leq \theta(n_2) + 1 \leq \text{LEN}(\theta(S)) + 1$, in which case
  - $\theta(s)$ is the contiguous subsequence of $\theta(S)$ extending from position $\theta(n_1)$ to $\theta(n_2)$ if $\theta(n_1) \leq \theta(n_2)$
  - $\theta(s) = \epsilon$ (the empty sequence) if $\theta(n_1) = \theta(n_2) + 1$
- if s_1 and s_2 are sequence terms, then $\theta(s_1 \bullet s_2) = \theta(s_1)\theta(s_2)$ if both $\theta(s_1)$ and $\theta(s_2)$ are defined; otherwise, $\theta(s_1 \bullet s_2)$ is *undefined.*

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

3.3 Fixpoint Semantics

The semantics of clauses is defined in terms of a least fixpoint theory. As in classical logic programming [21], each Sequence Datalog program, P, and database, DB, has an associated operator, $T_{P,\text{DB}}$, that maps Herbrand interpretations to Herbrand interpretations. Each application of $T_{P,\text{DB}}$ may create new atoms, which may contain new sequences. As shown below, $T_{P,\text{DB}}$ is monotonic and continuous, and thus has a least fixpoint that can be computed in a bottom-up, iterative fashion.

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

- The *extended active domain* of database DB is the union of the following three sets: (*i*) the *active domain* of the database, that is, the set of sequences occurring in DB; (*ii*) all contiguous subsequences of sequences in the active domain; and (*iii*) the set of integers $\{0, 1, 2, \ldots, l_{max} + 1\}$, where l_{max} is the maximum length of a sequence in the active domain.
- The *least fixpoint* [21] of operator $T_{P,\text{DB}}$ is computed in a bottom-up fashion, by starting at the empty interpretation and applying $T_{P,\text{DB}}$ repeatedly until a fixpoint is reached. At each step, and for each ground instantiation of each clause in P, if each premise of the clause has been derived, then the head of the clause is added to the set of derived facts. Because $T_{P,\text{DB}}$ is continuous, this process is complete [21]; that is, any atom in the least fixpoint will eventually be derived.
- At each step, if a derived fact contains a new sequence (*i.e.*, a sequence not currently in the extended active domain), then it is added to the active domain. Thus, as the bottom-up computation proceeds, the active domain (and hence the extended active domain) may expand. At each step of the computation, substitutions range over the current value of the extended active domain.

Note that the least fixpoint can be an infinite set. In this case, we say that the semantics of P over DB is *infinite*; otherwise, it is *finite*. Also note that our semantics for sequence creation resembles the semantics of *value invention* in [1] in that sequences are added to the active domain as a side-effect of clause evaluation. In Sequence Datalog, however, the addition is purely declarative and deterministic, since the least fixpoint is unique.

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

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

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

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

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

Lemma 1 *If* $I_1 \subseteq I_2$ *are two interpretations, then* $\mathcal{D}_{I_1}^{ext} \subseteq \mathcal{D}_{I_2}^{ext}$*. If* $\mathcal{I}$ *is a set of interpretations, then their union,* $\bigcup \mathcal{I}$*, is also an interpretation, and* $\mathcal{D}_{\bigcup \mathcal{I}}^{ext} = \bigcup_{I \in \mathcal{I}} \mathcal{D}_I^{ext}$*.*

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

Definition 4 (T-Operator) *The operator* $T_{P,\mathrm{DB}}$ *associated with program P and database* DB *maps interpretations to interpretations. In particular, if I is an interpretation, then* $T_{P,\mathrm{DB}}(I)$ *is the following interpretation:*

$$\{\theta(\textsc{head}(\gamma)) \quad | \quad \theta(\textsc{body}(\gamma)) \subseteq I \text{ for some clause } \gamma \in P \cup \mathrm{DB} \text{ and some substitution } \theta \text{ based on } \mathcal{D}_I^{ext} \text{ and defined at } \gamma\}$$

Lemma 2 (Monotonicity) *The operator* $T_{P,\mathrm{DB}}$ *is* monotonic*; i.e., if* $I_1 \subseteq I_2$ *are two interpretations, then* $T_{P,\mathrm{DB}}(I_1) \subseteq T_{P,\mathrm{DB}}(I_2)$*.*

Proof: Follows immediately from Definition 4 and Lemma 1. □

Lemma 3 (Continuity) *The operator* $T_{P,\mathrm{DB}}$ *is* continuous*; i.e., if* $I_1 \subseteq I_2 \subseteq I_3 \ldots$ *is a (possibly infinite) increasing sequence of interpretations, then* $T_{P,\mathrm{DB}}(\bigcup_i I_i) \subseteq \bigcup_i T_{P,\mathrm{DB}}(I_i)$*.*

Proof: Let $I = \bigcup_i I_i$ and let α be an atom in $T_{P,\mathrm{DB}}(I)$. We must show that α is also in $\bigcup_i T_{P,\mathrm{DB}}(I_i)$. By Definition 4, $\alpha = \theta(\textsc{head}(\gamma))$ and $\theta(\textsc{body}(\gamma)) \subseteq I$ for some clause γ in $P \cup \mathrm{DB}$, and some substitution θ based on $\mathcal{D}_I^{ext}$. In addition, since the interpretations, I_i, are increasing, their extended active domains, $\mathcal{D}_{I_i}^{ext}$, are also increasing, by Lemma 1. With this in mind, we proceed in three steps.

- $\theta(\textsc{body}(\gamma))$ is a finite set of ground atomic formulas, and $\theta(\textsc{body}(\gamma)) \subseteq \bigcup_i I_i$. Thus $\theta(\textsc{body}(\gamma)) \subseteq I_{j_1}$ for some j_1, since the I_i are increasing.
- Let $\mathcal{V}$ be the set of variables in clause γ, and let $\theta(\mathcal{V})$ be the result of applying θ to each variable. Thus $\theta(\mathcal{V})$ is a finite subset of $\mathcal{D}_I^{ext}$, since θ is based on $\mathcal{D}_I^{ext}$. Thus $\theta(\mathcal{V}) \subseteq \bigcup_i \mathcal{D}_{I_i}^{ext}$ by Lemma 1. Thus $\theta(\mathcal{V}) \subseteq \mathcal{D}_{I_{j_2}}^{ext}$ for some j_2, since $\theta(\mathcal{V})$ is finite and the $\mathcal{D}_{I_i}^{ext}$ are increasing.
- Let j be the larger of j_1 and j_2. Then $\theta(\textsc{body}(\gamma)) \subseteq I_j$ and $\theta(\mathcal{V}) \subseteq \mathcal{D}_{I_j}^{ext}$. Let θ' be any substitution based on $\mathcal{D}_{I_j}^{ext}$ that agrees with θ on the variables in $\mathcal{V}$. Then $\theta'(\gamma) = \theta(\gamma)$. Thus, $\theta'(\textsc{body}(\gamma)) \subseteq I_j$. Thus $\theta'(\textsc{head}(\gamma)) \in T_{P,\mathrm{DB}}(I_j)$ by Definition 4. Thus $\alpha \in T_{P,\mathrm{DB}}(I_j)$. Thus $\alpha \in \bigcup_i T_{P,\mathrm{DB}}(I_i)$.

□

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

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

Because $T_{P,\mathrm{DB}}$ is monotonic and continuous, we can invoke the Knaster-Tarsky fixpoint theorem [21]. $T_{P,\mathrm{DB}}$ thus has a least fixpoint, which is equal to $T_{P,\mathrm{DB}} \uparrow \omega$. We say that $T_{P,\mathrm{DB}} \uparrow \omega$ is the *fixpoint semantics* of program P over database DB. That is, $T_{P,\mathrm{DB}} \uparrow \omega$ is the set of facts implied by program P and database DB.

3.4 Model Theory

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

Definition 5 (Models) *An interpretation I is a* model *of a clause γ, written $I \models \gamma$, iff the following is true for each substitution θ based on $\mathcal{D}_I^{ext}$ and defined at γ:*

$$\textit{if} \quad \theta(\textsc{body}(\gamma)) \subseteq I \quad \textit{then} \quad \theta(\textsc{head}(\gamma)) \in I$$

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

Definition 6 (Entailment) *Let P be a Sequence Datalog program,* DB *be a database, and α be a ground atomic formula. Then, P and* DB *entail α, written $P, \mathrm{DB} \models \alpha$, if and only if every model of P and* DB *is also a model of α.*

Lemma 4 *Let P be a Sequence Datalog program,* DB *be a database, and I be an interpretation. Then I is a model of P and* DB *iff it is a fixpoint of $T_{P,\mathrm{DB}}$, i.e., iff $T_{P,\mathrm{DB}}(I) \subseteq I$.*

Proof: Suppose that I is a model of $P \cup \mathrm{DB}$. If $\alpha \in T_{P,\mathrm{DB}}(I)$ then by Definition 4, $\alpha = \theta(\mathrm{HEAD}(\gamma))$ for some clause γ in $P \cup \mathrm{DB}$, and some substitution θ based on $\mathcal{D}_I^{ext}$. Moreover $\theta(\mathrm{BODY}(\gamma)) \subseteq I$. But I is a model of $P \cup \mathrm{DB}$ by hypothesis, and thus of γ. Hence $\theta(\mathrm{HEAD}(\gamma)) \in I$, so $\alpha \in I$. Thus, any atom in $T_{P,\mathrm{DB}}(I)$ is also in I, so $T_{P,\mathrm{DB}}(I) \subseteq I$.

In the other direction, suppose that $T_{P,\mathrm{DB}}(I) \subseteq I$. Let γ be any clause in $P \cup \mathrm{DB}$, and let θ be any substitution based on $\mathcal{D}_I^{ext}$ and defined at γ. If $\theta(\mathrm{BODY}(\gamma)) \subseteq I$ then $\theta(\mathrm{HEAD}(\gamma)) \in T_{P,\mathrm{DB}}(I)$, by Definition 4. Thus $\theta(\mathrm{HEAD}(\gamma)) \in I$, since $T_{P,\mathrm{DB}}(I) \subseteq I$ by hypothesis. Thus, since θ is arbitrary, I is a model of γ. And, since γ is arbitrary, I is a model of $P \cup \mathrm{DB}$. □

The following corollaries are immediate consequences of Lemma 4 and the fixpoint theory of Section 3.3. They show that the model-theoretic semantics and the fixpoint semantics for Sequence Datalog are equivalent.

Corollary 1 *A Sequence Datalog program P and a database* DB *have a unique minimal model, and it is identical to the least fixpoint of the operator $T_{P,\mathrm{DB}}$.*

Corollary 2 *Let P be a Sequence Datalog program,* DB *be a database, and α be a ground atomic formula. Then $P, \mathrm{DB} \models \alpha$ if and only if $\alpha \in T_{P,\mathrm{DB}} \uparrow \omega$.*

4 Expressive Power of Sequence Datalog

In this section, we study the expressive power of Sequence Datalog programs. It turns out that the language has considerable power for manipulating sequences. In fact, it can express any computable sequence function, as the following result shows.

Theorem 1 (Expressive power) *Sequence Datalog expresses exactly the class of computable sequence functions,* i.e. *the class of partial recursive sequence functions.*

Proof: Because of its fixpoint semantics, each sequence function expressible in Sequence Datalog is partial recursive. All that remains is to prove that Sequence Datalog expresses every partial recursive sequence function. Specifically, given a fixed finite alphabet, Σ, we prove that for each partial recursive sequence function $f : \Sigma^* \to \Sigma^*$, there is a Sequence Datalog program, P_f, that expresses f. That is, given the database $\{input(\sigma_{in})\}$, if $f(\sigma_{in})$ is defined, then P_f infers the atom $output(\sigma_{out})$ iff $\sigma_{out} = f(\sigma_{in})$. Moreover, if $f(\sigma_{in})$ is undefined, then P_f infers no atoms of the form $output(X)$.

To construct P_f, let M_f be a Turing machine that computes f. Given an input sequence, σ, if f is defined at σ, then M_f halts and returns $f(\sigma)$ as output; otherwise, M_f does not halt. We construct P_f so that it simulates the computations of M_f. In particular, suppose $M_f = (\Sigma, K, s_0, \delta)$, where Σ is the tape alphabet, K is the set of control *states*, s_0 is the *initial state*, and δ is the *transition function*. We use a special symbol, $\triangleright$, to mark the left-end of the tape, and we assume the machine never tries to overwrite the left-end marker or to move beyond it.

We use sequences to represent the Turing machine tape and to represent a time counter. For the counter, we use a special symbol, #, and a unary encoding, so that the sequence #

represents the first instant of time, ## represents the second instant, ### the third instant, and so on. For the machine configuration, we use a 5-ary predicate, *conf(time,state,σ_l,a,σ_r)*, where:

- *time* is a sequence representing the time of the configuration,
- *state* is the state of the machine's finite control,
- σ_l is a sequence representing the portion of the tape to the *left* of the tape head,
- a is the symbol scanned by the tape head, and
- σ_r is a sequence representing the portion of the tape to the *right* of the tape head.

Thus, the tuple $conf(\#\#\#, q_1, abc, d, ef)$ means that after three steps of computation, the machine is in state q_1, the tape content is $abcdef$, and the tape head is scanning the symbol d.

We can assume that at the beginning of a computation, the machine scans the left-end marker, $\triangleright$. The initial configuration of the machine is thus specified by the following rule:

$$\gamma_1: \quad conf(\#, q_0, \epsilon, \triangleright, X) \leftarrow input(X).$$

The computation itself is specified by a set of rules, one rule for each machine transition. For example, consider the transition $\delta(q, a) = (q', b, -)$. This means that if the machine is currently in state q and scanning tape symbol a, then it goes into state q', writes the symbol b on the tape, and does not move the tape head. This transition is specified by the following rule:

$$\gamma_i: \quad conf(\# \bullet T, q', X_l, b, X_r) \leftarrow conf(T, q, X_l, a, X_r).$$

As another example, consider the transition $\delta(q, a) = (q', b, \leftarrow)$, in which the head writes a b on the tape and then moves one position to the left. This transition is specified by the following rule:

$$\gamma_j: \quad conf(\# \bullet T, q', X_l[1{:}end-1], X_l[end], b \bullet X_r) \leftarrow conf(T, q, X_l, a, X_r).$$

The situation is slightly more complicated if the tape head moves to the right. In this case, we must append blank symbols to the right end of the tape; otherwise, the tape head might move beyond the right end of the tape. By appending blanks, we effectively simulate a tape of unbounded (*i.e.*, infinite) length. To express the transition $\delta(q, a) = (q', b, \rightarrow)$, we use the following rule, where $\sqcup$ denotes the blank symbol:

$$\gamma_{k_1}: \quad conf(\# \bullet T, q', X_l \bullet b, X_r[1], X_r[2{:}end] \bullet \sqcup) \leftarrow conf(T, q, X_l, a, X_r).$$

Finally, the following rule detects the end of a computation, and returns the contents of the machine tape.

$$\gamma_2: \quad output(X_l[2{:}end] \bullet S \bullet X_r) \leftarrow conf(T, q_h, X_l, S, X_r).$$

The body of this rule simply checks whether the machine is in a halting state, q_h. If so, then the contents of the machine tape (minus the left-end marker) are extracted, converted to a single sequence, and passed to the rule head. This sequence is the output of the Turing machine computation.

It is not hard to see that the program, P_f, consisting of the rules above correctly simulates the computation of the Turing machine. Moreover, it expresses the partial function f computed by the machine. □

Note that although Sequence Datalog is function complete, it is not query complete, since it expresses only *monotonic* queries.

5 The Finiteness Problem

In this section, we are interested in studying the behaviour of Sequence Datalog programs with respect to the existence of a finite semantics. Consider the following example.

Example 5.1 [**Infinite Semantics**]
Suppose R is a unary relation containing a set of sequences. For each sequence, X, in R, we want the sequence obtained by repeating each symbol in X twice. For example, given the sequence *abcd*, we want the sequence *aabbccdd*. We call these sequences *echo* sequences. The easiest way to define echo sequences is with the following program:

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

The first rule retrieves every sequence in relation R and its echo, by invoking the predicate $echo(X, Y)$. The last two rules specify what an echo sequence is. For every sequence, X, in the extended active domain, these rules generate its echo sequence, Y. Starting with $X = \epsilon$ and $Y = \epsilon$, they recursively concatenate single characters onto X while concatenating two copies of the same character onto Y. As new sequences are generated, they are added to the active domain, which expands indefinitely. □

The program in Example 5.1 has an infinite semantics over every database that contains a non-empty sequence. This is because the rules defining $echo(X, Y)$ recursively generate longer and longer sequences without bound. For example, suppose the input database contains only one tuple, $\{R(aa)\}$. Its extended active domain consists of the sequences ϵ, a, aa. The table below shows how the inferred facts and the extended domain both grow during a bottom up computation of the least fixpoint. Each row in the table is the result of one additional application of the T operator. In each row, the inferred facts contain one more echo entry, and the extended active domain contains one more sequence, consisting entirely of a's. The least fixpoint of the T operator is therefore infinite, and its extended active domain is the set of all sequences made of a's. Note that the query answer consists of a single atom, $answer(aa, aaaa)$, which is computed during the fourth step. Thus, although the least fixpoint is infinite, the query answer is not.

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

Example 5.1 suggests that:

- the language allows for nonterminating computations;

- nonterminating computations are due to the interaction between recursion and constructive terms, which possibly add new sequences to the active domain.

We say that a program P is *finite* if the semantics of P is finite over every input instance DB. The *finiteness problem* for Sequence Datalog programs is the following: *given a program P, is P finite ?* We can state the following result, which follows from Theorem 1.

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

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

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

Proof: The PTIME lower bound follows because Sequence Datalog includes Datalog as a sublanguage. The PTIME upper bound follows because non-constructive programs have an extended active domain of polynomial size with respect to the size of the input database. In fact, the initial extended domain is polynomial with respect to the size of the database, and it does not grow during the computation. This implies that the fixpoint computation saturates after a polynomial number of iterations and that each iteration can be performed in polynomial time. □

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

6 Generalized Sequence Transducers

Because they cannot construct new sequences, non-constructive programs have weak restructuring capabilities. To increase these capabilities—while preserving finiteness—we use an abstract computational device called a *generalized sequence transducer*. Transducers are low-complexity devices that take sequences as input and produce new sequences as output. They are therefore natural devices for restructuring sequences (see also [12, 16, 13, 26]). Moreover, we can exploit the low complexity of transducers to guarantee finiteness.

6.1 The Machine Model

A *transducer* is usually defined as a machine with n input lines, one output line, and an internal state. The machine sequentially "reads" the input strings, and progressively "computes" the output. At each step of the computation, the current input symbols are read and, based on the current state, a sequence is appended to the output and a new state is chosen. The computation is guaranteed to terminate since at each step at least one input symbol is "consumed". Transducers therefore have very low complexity—essentially linear time. There are therefore many

sequence restructurings that they cannot perform. To allow for more complex restructurings, we introduce a new computational device, which we call a *generalized sequence transducer.*

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

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

To formally define the notion of generalized sequence transducers, we use three special symbols, $\triangleleft$, $\rightarrow$ 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 many of the transducer models proposed in the literature (see for example [12, 26]).

Definition 7 (Generalized Transducers) *A generalized n-ary sequence transducer of order* $k > 0$ *is a 4-tuple* $\langle K, q_0, \Sigma, \delta \rangle$ *where:*

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

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

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

The computation of a generalized sequence transducer over input strings $\langle \sigma_1, \ldots, \sigma_n \rangle$ proceeds as follows. To start, the machine is in its initial state, q_0, each input head scans the first

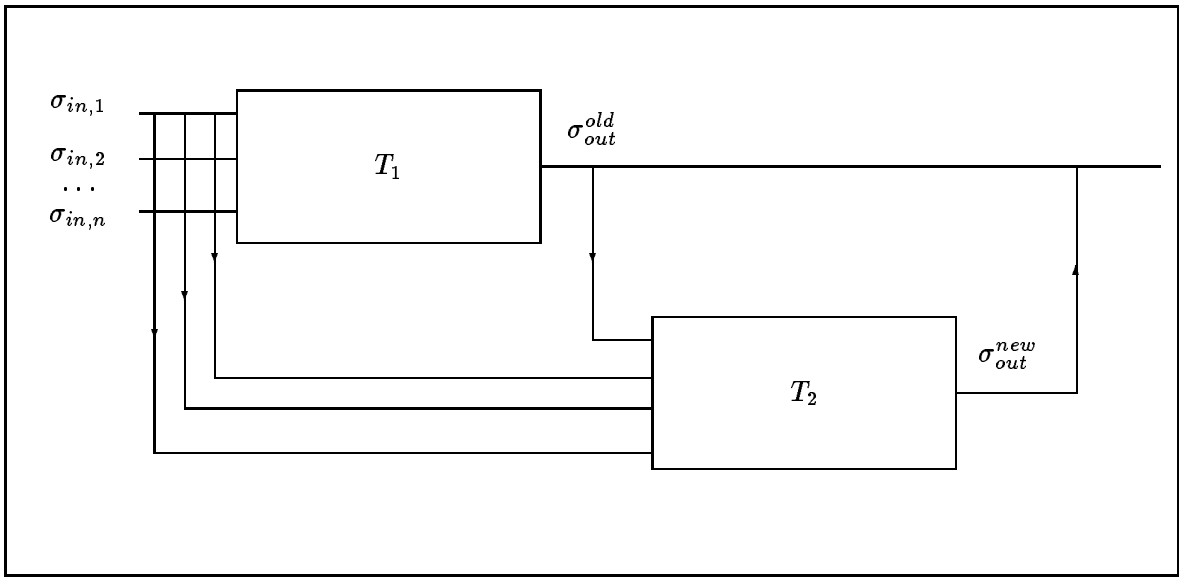

Figure 1: Transducer T_1 calls subtransducer T_2.

(*i.e.*, leftmost) symbol of its tape, and the output tape is empty. At each point of the computation, the internal state and the tape symbols below the input heads determine what transition to perform. If the internal state is q and the tape symbols are $a_1 \ldots a_n$, then the transition is $\delta(q, a_1, \ldots, a_n) = \langle q', c_1, \ldots, c_n, \text{OUT} \rangle$. This transition is carried out as follows:

- If OUT is a symbol in Σ, then it is appended to the output sequence; if $\text{OUT} = \epsilon$, then the output is unchanged. This takes 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. The output of the subtransducer 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. These ideas are illustrated in Figure 1.
- 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 by the following example.

Example 6.1 **[Quadratic Output]**
Let T_{square} be a generalized transducer with one input. At each step of its computation, T_{square} calls a subtransducer, T_{append}, with two inputs (see Figure 2). One input to T_{append} is the input to T_{square}, and the other is the output of T_{square}. T_{append} simply appends its two inputs. The output of T_{append} then becomes the output for T_{square}, overwriting the old output.

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

- At time 1, the two inputs to T_{append} are σ_{in} and the empty sequence, ϵ (since the output of T_{square} is initially empty). Thus, the output of T_{append} at this step is σ_{in}, which becomes the new output for T_{square}.
- At time 2, the two inputs to T_{append} both contain a copy of σ_{in}. Thus, the output of T_{append} at this step is the concatenation of σ_{in} with itself, which is a sequence of length $2n$. This sequence then becomes the new output for T_{square}.
- In general, at time i, for $1 \leq i \leq n$, the two inputs to T_{append} are σ_{in} and the sequence obtained by concatenating σ_{in} with itself $i-1$ times. Thus, the output of T_{append} at this step is the sequence obtained by concatenating σ_{in} with itself i times. This sequence then becomes the new output for T_{square}.

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

6.2 Transducer Networks

Transducers can be combined to form *networks*, in which the output of one transducer is an input to other transducers. Since we are interested in finite computations, we only consider *acyclic networks*, in which the output of a transducer is never fed back to its own input. For each transducer network, some transducer inputs are designated as network inputs, and some transducer outputs are designated as network outputs. Each network then computes a *mapping* from sequence tuples to sequence tuples. When the network has only one output, the network computes a sequence function. This section presents basic results about the complexity of generalized transducer networks. A more detailed analysis is beyond the scope of this paper and will be reported elsewhere [22].

The computational complexity of the sequence function computed by a transducer network depends on two parameters. The first is the *diameter* of the network, *i.e.*, the maximum length of a path in the network. The diameter determines the maximum number of transformations that a sequence will undergo in traveling from the input to the output of a network. The second parameter is the *order* of the network. This is maximum order of any transducer in the network.

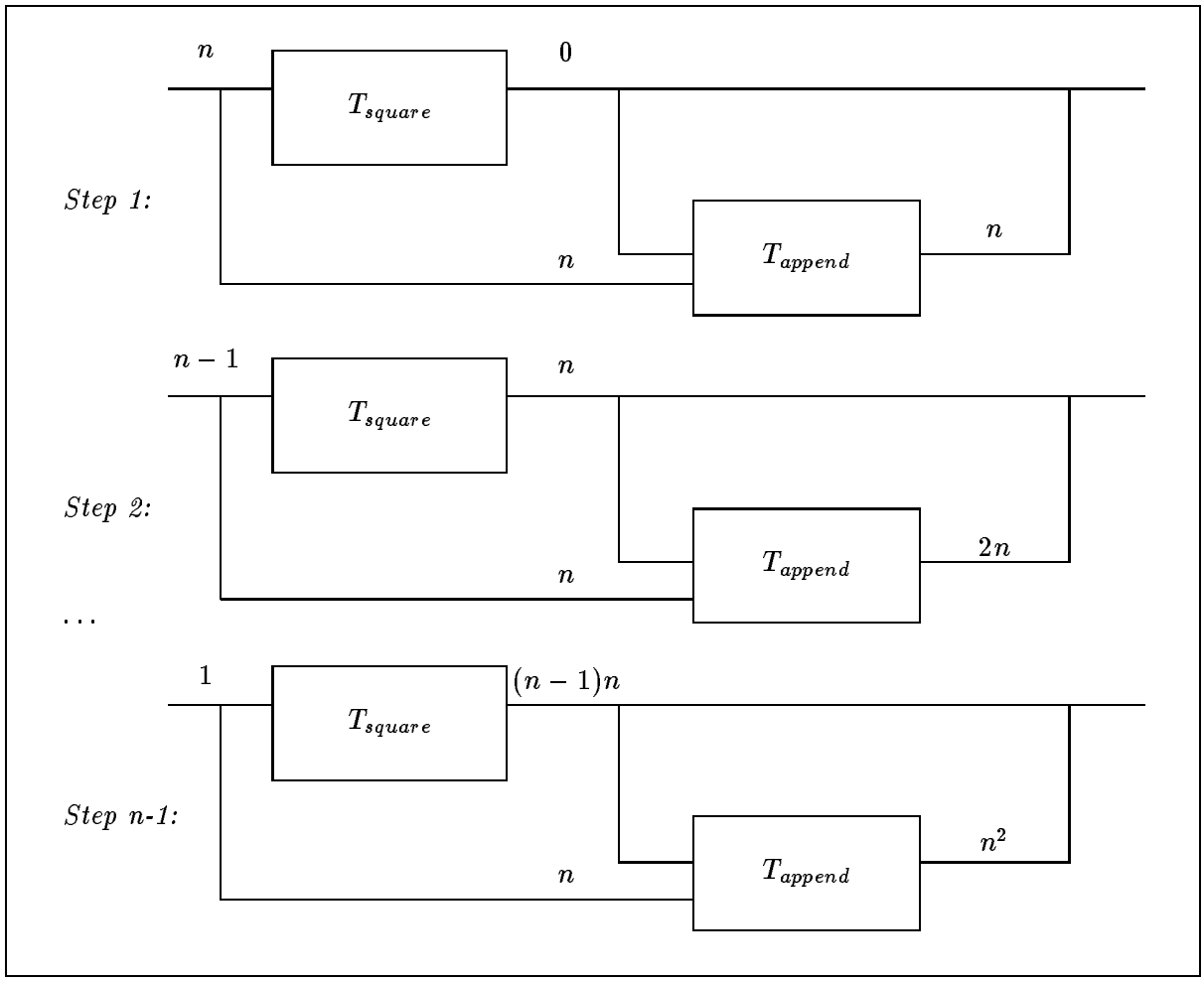

Figure 2: Squaring the input.

This figure illustrates the transducer of Example 6.1 at various stages of computation. The numbers on the input and output lines indicate sequence lengths.

If the set of transducers in the network is a subset of $\mathcal{T}^k$, then the order of the network is at most k. Intuitively, the order of a network is the maximum depth of subtransducer nesting.

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

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

hyp_i is called the hyper-exponential function of level i. The set of *elementary sequence functions* is the set of sequence functions that have hyper-exponential time complexity, that is, the set of sequence functions in $\bigcup_{i\geq 0} \mathrm{DTIME}[hyp_i(\mathcal{O}(n))]$.

The theorems below characterize the complexity and expressibility of two classes of transducer networks, those of order 2 and 3, respectively. Higher order networks will be investigated in [22]. Our first results concern the output size of transducer networks.

Theorem 4 (Complexity of Transducer Networks) *Consider an acyclic network of transducers with m inputs and 1 output.*

- *If the network has order 2, then the length of the output is (at most) polynomial in the sum of input lengths.*
- *If the network has order 3, then the length of the output is (at most) hyper-exponential in the sum of input lengths.*

Moreover, these bounds are tight.

Proof: In the following, $|out_j|$ denotes the *length* of the final output of transducer T_j, and $|in_j|$ denotes the *total length* of its initial inputs, *i.e.*, the sum of the lengths of all input sequences. Superscripts denote particular steps of a computation; *e.g.*, $|out_j^i|$ denotes the output length of transducer T_j at step i of its computation.

First note that the output length of a base transducer is linear with respect to its total input length. In fact, the output can be at most as long as the the concatenation of its inputs. In symbols, $|out_j| \leq |in_j|$. With this in mind, we prove each of the two items of the theorem in turn. In each proof, we first consider a single transducer, *i.e.* a network of diameter 1; and then we extend the proof to networks of arbitrary diameter.

Networks of Order 2. Let T_1 be a transducer of order 2. At step i of its computation, either T_1 may append a symbol to its output, or it calls a subtransducer, T_2, whose output overwrites the output of T_1. Thus, one of the following is true for each i:

$$\begin{cases} |out_1^{i+1}| \leq |out_1^i| + 1 \\ |out_1^{i+1}| = |out_2| \leq |in_1| + |out_1^i| \end{cases}$$

The second inequality follows because the input to T_2 is the initial input to T_1 plus the current output of T_1.

In the worst case, T_1 calls a subtransducer at each step of its computation, and the subtransducer produces an output as long as its total input. In this case, the second inequality above becomes an equality. The following recursive equation thus describes the worst case:

$$\begin{cases} |out_1^0| = 0 \\ |out_1^{i+1}| = |in_1| + |out_1^i| \end{cases}$$

Solving the equation, we get $|out_1^i| = i\,|in_1|$. The computation terminates after n steps, where $n = |in_1|$ is the total input length of T_1. Thus, in the worst case, the final output is quadratic in the initial input, *i.e.*, $|out_1| = n^2$.

Now consider an acyclic network of transducers with diameter d and order 2. Let $T_1, \ldots, T_d$ be the transducers on the longest path in the network. In the worst case, the output of each transducer is quadratic in its total input; and each network input goes to transducer T_1. Thus, if n is the total length of the network input, then the total input length of T_1 is n, and

$$\begin{cases} n^2 \leq |out_1| \leq \mathcal{O}(n^2) \\ n^4 \leq |out_2| \leq \mathcal{O}(n^4) \\ n^8 \leq |out_3| \leq \mathcal{O}(n^8) \\ \qquad \cdots \\ n^{2^d} \leq |out_d| \leq \mathcal{O}(n^{2^d}) \end{cases}$$

The left-hand inequalities arise because at least one copy of the output of T_i forms an input to T_{i+1}. The right hand inequalities arise because T_{i+1} may have several inputs. In any event, since the diameter d is fixed, the final output length is polynomial in n. This proves the first part of the Theorem.

Networks of Order 3. The proof for this case is similar. Let T_1 be a transducer of order 3. At each step of its computation, T_1 may call a subtransducer, T_2, of order 2. In the worst case, T_1 calls T_2 at every step, and T_2 squares its input, as shown above. Thus, at step i of T_1's computation, we have:

$$|out_1^i| = |out_2| = (|in_1| + |out_1^{i-1}|)^2$$

The worst case is therefore described by the following recursive equation:

$$\begin{cases} |out_1^0| = 0 \\ |out_1^i| = (|in_1| + |out_1^{i-1}|)^2 \end{cases}$$

Solving the equation, we get $|out_1^i| = |in_1|^{2^i} + \mathcal{O}(|in_1|^{2^{i-1}})$. The computation terminates after n steps, where $n = |in_1|$ is the total input length of T_1. Thus $|out_1| = n^{2^n} + \mathcal{O}(n^{2^{n-1}})$. Thus $2^{2^n} \leq |out_1| \leq 2^{2^{\mathcal{O}(n)}}$ in the worst case; *i.e.*, the final output is double exponential in the initial input. Intuitively, this comes about because T_1 can use T_2 to square the input length, n, and it can do this n times.

Now consider an acyclic network of transducers with diameter d and order 3. Let $T_1, \ldots, T_d$ be the transducers on the longest path in the network. The output of each transducer is at most double exponential in its total input. Thus, if n is the total length of the network input, then

in the worst case

$$\begin{cases} hyp_2(n) = 2^{2^n} \leq |out_1| \leq 2^{2^{\mathcal{O}(n)}} = hyp_2(\mathcal{O}(n)) \\ hyp_4(n) = 2^{2^{hyp_2(n)}} \leq |out_2| \leq 2^{2^{hyp_2(\mathcal{O}(n))}} = hyp_4(\mathcal{O}(n)) \\ hyp_6(n) = 2^{2^{hyp_4(n)}} \leq |out_3| \leq 2^{2^{hyp_4(\mathcal{O}(n))}} = hyp_6(\mathcal{O}(n)) \\ \qquad\qquad \cdots \\ hyp_{2d}(n) \leq |out_d| \leq hyp_{2d}(\mathcal{O}(n)) \end{cases}$$

This proves the second part of the Theorem. □

Building on Theorem 4, we prove two expressibility theorems for acyclic networks of transducers. Both theorems are concerned with transducer networks that have a single input and a single output. Such networks compute a function from sequences to sequences. Theorem 5 first characterizes the sequence functions computable in polynomial time. (Other characterizations can be found in [10, 6, 18].) Theorem 6 then characterizes the sequence functions computable in elementary time.

Theorem 5 (Expressibility of Order-2 Networks) *Acyclic transducer networks of order 2 express exactly the class of sequence functions computable in* PTIME.

Proof: To prove the upper expressibility bound, suppose we are given a transducer network of order 2. By Theorem 4, the input to each transducer in the network is of polynomial size (polynomial in the size of the network input). Moreover, each transducer in the network is of order 1 or 2, so it consumes its input in polynomial time. Each transducer therefore runs in polynomial time wrt to the size of the network input.

To prove the lower expressibility bound, we must show that every sequence function in PTIME can be computed by an acyclic transducer network of order 2. Any such sequence function is computed by a Turing machine, M, that runs in polynomial time. Without loss of generality, we can assume that M has only one tape and runs in time n^k for some k, where n is the length of its input. We shall simulate the computations of M using a transducer network.

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 transducer is a configuration of M, then the output is the next configuration of M.

To compute a sequence function, our transducer network performs three tasks: it constructs the initial configuration of the Turing machine, it simulates n^k Turing-machine steps, and it extracts the output sequence from the final configuration. These tasks are carried out as follows:

- Given an input sequence $a_1 a_2 \ldots a_n$, a transducer constructs the initial configuration of the Turing machine. 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 2.

- Given the input sequence (of length n), a series of transducers generates a sequence, σ_{count}, of length at least n^k. We shall use this sequence to count time. It is easily generated using $\lceil log_2(k) \rceil$ transducers of order 2, where the output length of each transducer is the square of its input length.

- A transducer T_M of order 2 simulates the computation of Turing machine M. This transducer has two inputs: the counter sequence σ_{count}, and the initial configuration of M. T_M first moves the initial configuration from its input to its output. It then repeatedly calls a subtransducer while consuming its other input. It thus calls the subtransducer at least n^k 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 its input, its output represents the final configuration of machine M.
- A transducer T_{decode} decodes the final configuration of machine M. To do this, we can assume the final configuration has the form $c_1c_2\ldots c_mq_zxxxxxx\ldots x$, where $c_1c_2\ldots c_m$ 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.

□

Theorem 6 (Expressibility of Order-3 Networks) *Acyclic transducer networks of order 3 express exactly the class of sequence functions computable in elementary time.*

Proof: The proof is similar to that of Theorem 5. The main difference is that the sequence σ_{count} must be of hyper-exponential length. This is easily accomplished by a series of transducers of order 3, as shown in the proof of Theorem 4. □

There is a close relationship between the diameter of transducer networks and levels in the hyper-exponential hierarchy. For instance, any sequence function in EXPTIME can be expressed by a single transducer of order 3. These ideas will be further developed in [22].

7 Sequence Datalog with Transducers

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

7.1 Syntax and Semantics

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

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

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

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

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

where T_{append} is a transducer that concatenates its two inputs. As this example shows, constructive terms are not needed in Transducer Datalog, since they can be replaced by transducer terms. Thus, in the sequel, they will not be used in Transducer Datalog programs. In Sequence Datalog, rules with constructive terms in the head are called constructive rules (or clauses). The above example suggests a natural extension of this idea: in Transducer Datalog, rules with transducer terms in the head will also be called *constructive rules.* 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 has order 0.

The semantics of Transducer Datalog is an extension of the semantics of Sequence Datalog. The only change is to extend the interpretation of sequence terms to include transducer terms. This can be done in a natural way. Let θ be a substitution based on a domain $\mathcal{D}$. Thus, $\theta(s)$ is a sequence in $\mathcal{D}$ for any sequence term s. To extend θ to transducer terms, define $\theta(T(s_1, \ldots, s_m))$ to be the output of transducer T on inputs $\theta(s_1), \ldots, \theta(s_m)$, where the symbols of each $\theta(s_i)$ are consumed from left to right. Except for this change, the semantics of Transducer Datalog is identical to that of Sequence Datalog.

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

Example 7.1 **[RNA Transcription]**
A fundamental operation in Molecular Biology is the transcription of DNA into RNA. DNA sequences can be modeled as strings over the alphabet $\{a, c, g, t\}$, where each character represents a *nucleotide.* Likewise, RNA sequences can be modeled as strings over the alphabet $\{a, c, g, u\}$, where each character represents a *ribonucleotide.* Each nucleotide in a DNA sequence is transcribed into a ribonucleotide in a RNA sequence according to the following rules:

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

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

This transformation is easily and naturally expressed as a sequence transducer, $T_{transcribe}$, in which the input is a DNA sequence, and the output is a RNA sequence. Given a relation,

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

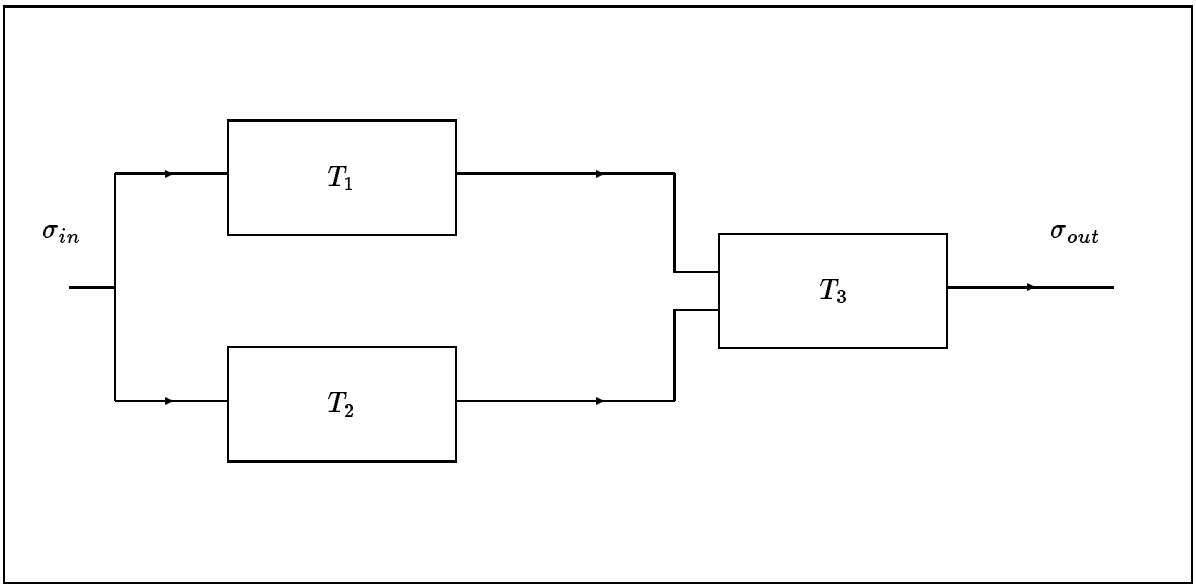

Figure 3: An acyclic transducer network.

dna_seq, containing DNA sequences, the following Transducer Datalog rule transcribes each of the sequences into RNA:

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

□

Although the Transducer Datalog program in Example 7.1 consists of only one rule, two features of this rule are worth noting: (i) all sequence restructurings performed by the program take place "inside" the transducer $T_{transcribe}$; and (ii) the program terminates for every database, since there is no recursion through construction of new sequences.

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.

Example 7.2 [**A Simple Network**] The Transducer Datalog program below corresponds to the network in Figure 3. Rules γ_1 and γ_2 first feed each sequence in relation *input* through transducers T_1 and T_2, respectively. Rule γ_3 then feeds the output of transducers T_1 and T_2 through transducer T_3. Since the program is non-recursive, the corresponding network is acyclic.

$$\begin{array}{lll} \gamma_1 & r(T_1(X)) & \Leftarrow input(X). \\ \gamma_2 & q(T_2(X)) & \Leftarrow input(X). \\ \gamma_3 & p(T_3(X,Y)) & \Leftarrow r(X), q(Y). \end{array}$$

□

Example 7.3 **[Protein Translation]**
Another fundamental operation in Molecular Biology is the translation of RNA into protein. As mentioned above, RNA sequences can be modelled as strings over the alphabet $\{a, c, g, u\}$. Likewise, proteins can be modelled as sequences over a twenty-character alphabet, {A, R, N, D, C, Q, E, G, H, I, L, K, M, F, P, S, T, W, Y, V}, where each character represents an *amino acid*. To translate RNA into protein, ribonucleotides are grouped into triplets, called *codons*, such as *aug*, *acg*, *ggu*, ...[5] Each codon is then translated into a single amino-acid. Different codons may have the same translation. For example, the codons *gau* and *gac* both translate to *aspartic acid*, denoted D in the twenty-letter alphabet. Thus, the RNA sequence *gaugacuuacac* is first grouped into a sequence of four codons, *gau/gac/uua/cac*, and then translated into a sequence of four amino acids, DDLH.[6]

As in Example 7.1, this process is easily and naturally expressed as a sequence transducer, $T_{translate}$, in which the input is a RNA sequence, and the output is a protein sequence. Given a relation, *rna_seq*, containing RNA sequences, the following Transducer Datalog rule translates each of the sequences into protein:

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

The transducer $T_{translate}$ can be combined with the transducer $T_{transcribe}$ from Example 7.1 to form a simple serial network in which the output of $T_{transcribe}$ is the input to $T_{translate}$. This network simulates the transformation of DNA into RNA into protein. This network is represented by the following program:

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

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

7.2 Equivalence to Sequence Datalog

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

Theorem 7 *For any Transducer Datalog program P_{td}, there is a Sequence Datalog program P_{sd} that expresses the same queries. That is, for every database* DB*, and every predicate $p(X_1, \ldots, X_n)$ mentioned in $P_{td} \cup$ DB,*

$$P_{sd}, \text{DB} \models p(\sigma_1, \ldots, \sigma_n) \qquad \textit{iff} \qquad P_{td}, \text{DB} \models p(\sigma_1, \ldots, \sigma_n)$$

Moreover, P_{sd} has a finite semantics over DB *if and only if P_{td} does. That is, P_{sd} preserves finiteness.*

Proof: For each transducer mentioned in P_{td}, we will construct a set of rules for P_{sd} that simulate the transducer's computations. As a side effect, the simulation may create sequences that P_{td}

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

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

does not create. Thus, the minimal models of P_{td} and P_{sd} may not have the same extended active domain. We shall therefore assume that P_{td} is guarded, since guarded programs are insensitive to differences in domain. By Theorem 10 in Appendix A, this assumption does not result in any loss of generality.

To preserve finiteness, our construction has the following property: P_{sd} simulates a transducer on input $\sigma_1, \ldots, \sigma_n$ if and only if P_{td} invokes the transducer on this input. Intuitively, this property means that P_{sd} creates new sequences only when P_{td} does. To see how finiteness is preserved, recall that P_{td} creates new sequences only by invoking transducers and adding the transducer output to the active domain. Each transducer invocation thus adds at most *one* new sequence to the active domain. In contrast, when P_{sd} simulates the computation of a transducer, T, it creates not just the *final* output sequence, but all *intermediate* output sequences as well. It also simulates the computations of any subtransducers invoked by T, and creates all their output sequences. Thus, each transducer simulation in P_{sd} may add *many* new sequences to the active domain. However, because generalized transducers always terminate, they produce only a finite number of intermediate outputs. Thus, for each sequence created by P_{td}, a finite number of sequences are created by P_{sd}. Thus, if P_{td} creates a finite number of new sequences, then so does P_{sd}. In this way, the construction preserves finiteness.

The construction itself has several steps. In the first step, each rule in P_{td} is modified and copied into P_{sd}. In modifying a rule, we replace each transducer term, $T(s_1, \ldots, s_n)$, by a new predicate $p_T(s_1, \ldots, s_n, X)$. Intuitively, this predicate means that X is the output of transducer T on inputs $s_1, \ldots, s_n$. For example, if a rule has a single transducer term, then it has the following form:

$$\gamma: \ p(\ldots, T(s_1, \ldots, s_n), \ldots) \ \Leftarrow \ \text{BODY}(\gamma).$$

For each such rule in P_{td}, we add the following rule to P_{sd}:

$$\gamma': \ p(\ldots, X, \ldots) \ \leftarrow \ \text{BODY}(\gamma), \ p_T(s_1, \ldots, s_n, X).$$

where X is a variable that does not appear in γ. This idea is easily generalized to rules with multiple transducer terms. If a rule in P_{td} has k transducer terms, involving transducers $T_1, \ldots T_k$, then we transform it into a rule whose body invokes the predicates $p_{T_1}, \ldots, p_{T_k}$.

The main step in our construction is defining the predicate p_T for any generalized transducer T. In doing so, we take care to preserve finiteness. In particular, we ensure that $p_T(\sigma_1, \ldots, \sigma_n, x)$ is true only if program P_{td} invokes transducer T on input $\sigma_1, \ldots, \sigma_n$. We do this by defining a predicate called $input_T$ that specifies the set of input tuples for T. This predicate is easily defined. If the function term $T(s_1, \ldots, s_n)$ occurs in rule γ in P_{td}, then we add the following rule to P_{sd}:

$$\gamma'': \ input_T(s_1 \bullet \lhd, \ldots, s_n \bullet \lhd) \ \leftarrow \ \text{BODY}(\gamma).$$

We do this for each occurrence of T in each rule in P_{td}. Note that rule γ'' appends an end-of-tape marker to the end of each input sequence. The model of generalized transducers developed in Section 6.1 assumes that such markers appear at the end of every input tape. In the worst case, appending these markers could double the number of sequences in the active domain; so finiteness is still preserved.

To define the predicate p_T, we write a set of rules in Sequence Datalog that simulate the computations of transducer T on every sequence tuple in $input_T$. We first construct rules for base transducers, and then extend the construction to higher-order transducers.

Base Transducers. For simplicity, let T be a base transducer with two inputs and one output. (The proof can be easily generalized to any number of inputs and outputs.) We encode the transition function of T with the following database predicate:

$$delta_T(state,\ input_1,\ input_2,\ next\_state,\ move_1,\ move_2,\ output)$$

Here, *state* is the state of the finite control, $input_1$ and $input_2$ are the symbols scanned by the two input heads, *next_state* is the next state of the finite control, $move_1$ and $move_2$ describe the motion of the input heads, and *output* is the symbol appended to the output sequence. Given this encoding, the rules below define the predicate $p_T(X, Y, Z)$, which means that Z is the output of transducer T on input X and Y. This predicate is evaluated for every pair of input sequences in the predicate $input_T(X, Y)$. P_T is defined in terms of the more general predicate $comp_T$, which represents a partial computation of transducer T. Intuitively, $comp_T(X, Y, Z, Q)$ means that after consuming inputs X and Y, the output is Z and the control state is Q.

$$\begin{array}{lrcl}
\gamma_1: & p_T(X,Y,Z) & \leftarrow & input_T(X,Y,Z),\\ & & & comp_T(X,Y,Z,Q).\\
\gamma_2: & comp_T(\epsilon,\epsilon,\epsilon,q_0) & \leftarrow & true.\\
\gamma_3: & comp_T(X[1{:}N_1+1],Y[1{:}N_2+1],Z\bullet O,Q') & \leftarrow & input_T(X,Y),\\ & & & comp_T(X[1{:}N_1],Y[1{:}N_2],Z,Q),\\ & & & delta_T(Q,X[N_1+1],Y[N_2+1],Q',\rightarrow,\rightarrow,O).\\
\gamma_4: & comp_T(X[1{:}N_1],Y[1{:}N_2+1],Z\bullet O,Q') & \leftarrow & input_T(X,Y),\\ & & & comp_T(X[1{:}N_1],Y[1{:}N_2],Z,Q),\\ & & & delta_T(Q,X[N_1+1],Y[N_2+1],Q',-,\rightarrow,O).\\
\gamma_5: & comp_T(X[1{:}N_1+1],Y[1{:}N_2],Z\bullet O,Q') & \leftarrow & input_T(X,Y),\\ & & & comp_T(X[1{:}N_1],Y[1{:}N_2],Z,Q),\\ & & & delta_T(Q,X[N_1+1],Y[N_2+1],Q',\rightarrow,-,O).
\end{array}$$

Rule γ_2 initiates a simulation of transducer T. It says that when no input has been consumed, the output is empty and the finite control is in the initial state, q_0. Rules γ_3, γ_4 and γ_5 then simulate the transition of the machine from one state to the next. The three rules simulate three different combinations of tape-head movements. Rule γ_3 simulates the movement of both tape heads, while rules γ_4 and γ_5 simulate the movement of just one tape head. In these rules, the sequence terms $X[1{:}N_1]$ and $Y[1{:}N_2]$ represent the portions of the input sequences consumed so far; and the terms $X[N_1+1]$ and $Y[N_2+1]$ represent the input symbols currently being scanned by the tape heads.

Higher-Order Transducers. Given rules for simulating transducers of order $k-1$, it is not difficult to write rules for simulating transducers of order k. We illustrate the idea on a simple example, one that brings out all the essential elements of the general construction. Let T' be a transducer of order 2 with one input and one output. We encode the transition function of T' in the following database predicate:

$$delta_{T'}(state,\ input,\ next\_state,\ move,\ output)$$

Here, *state* is the state of the finite control, *input* is the symbol scanned by the input head, *next_state* is the next state of the finite control, and *move* describes the motion of the input head. *output* is a tape symbol or the name of a subtransducer: in the former case, the symbol is appended to the output sequence; in the later case, the subtransducer is executed. Given this

encoding, the rules below define the predicate $p_{T'}(X, Y)$, which means that Y is the output of transducer T' on input X. This predicate is evaluated for every input sequence in the predicate $input_{T'}(X)$.

$$\begin{array}{lrcl}
\gamma_1': & p_{T'}(X,Y) & \leftarrow & input_{T'}(X), \\
 & & & comp_{T'}(X,Y,Q). \\
\gamma_2': & comp_{T'}(\epsilon,\epsilon,q_0) & \leftarrow & true. \\
\gamma_3': & comp_{T'}(X[1{:}N+1], Y \bullet O, Q') & \leftarrow & input_{T'}(X), \\
 & & & comp_{T'}(X[1{:}N],Y,Q), \\
 & & & delta_{T'}(Q, X[N+1], Q', \rightarrow, O).
\end{array}$$

In addition, for each subtransducer T invoked by T', we construct the following two rules:

$$\begin{array}{lrcl}
\gamma_4': & comp_{T'}(X[1{:}N+1], Z, Q') & \leftarrow & input_{T'}(X), \\
 & & & comp_{T'}(X[1{:}N],Y,Q), \\
 & & & delta_{T'}(Q, X[N+1], Q', \rightarrow, \mathrm{T}), \\
 & & & p_T(X,Y,Z). \\
\gamma_5': & input_T(X \bullet \lhd, Y \bullet \lhd) & \leftarrow & input_{T'}(X), \\
 & & & comp_{T'}(X[1{:}N],Y,Q), \\
 & & & delta_{T'}(Q, X[N+1], Q', \rightarrow, \mathrm{T}).
\end{array}$$

As before, $p_{T'}$ is defined in terms of the more general predicate $comp_{T'}$. Intuitively, atoms of the form $comp_{T'}(X, Y, Q)$ mean that after consuming input X, the output of T' is Y and the finite control is in state Q. As before, rule γ_2' initiates a simulation of transducer T', and rule γ_3' simulates basic state transitions, in which a symbol is appended to the output. Rules γ_4' and γ_5' are new. Rule γ_4' simulates state transitions involving the subtransducer T (denoted T in the predicate $delta_{T'}$). The body of the rule invokes the subtransducer via the predicate $p_T(X, Y, Z)$. The subtransducer has *two* input sequences. One is X, the initial input to T', and the other is Y, the current output of T'. After the subtransducer has executed, its output, Z, becomes the new output of T'. The actual simulation of the subtransducer is carried out by rules γ_1–γ_5. For the simulation to work, the input tuples for T must be specified in the predicate $input_T(X)$. This is done by rule γ_5'. □

Example 7.4 [**Simulating Transducers**]
The Sequence Datalog program below simulates the Transducer Datalog rule in Example 7.1:

$$\begin{array}{rcl}
rna_seq(R) & \leftarrow & dna_seq(D), \\
 & & transcribe(D,R). \\
 & & \\
transcribe(\epsilon,\epsilon) & \leftarrow & true. \\
transcribe(D[1{:}N+1],\ T \bullet R) & \leftarrow & dna_seq(D), \\
 & & transcribe(D[1{:}N],R), \\
 & & trans(D[N+1],T). \\
 & & \\
trans(a,u) & \leftarrow & true. \\
trans(t,a) & \leftarrow & true. \\
trans(c,g) & \leftarrow & true. \\
trans(g,c) & \leftarrow & true.
\end{array}$$

The first rule transcribes every DNA sequence in the relation *dna_seq* into an RNA sequence, by invoking the predicate *transcribe*. The formula $transcribe(D, R)$ is true iff D is a prefix of a DNA sequence in the database and R is the RNA transcription of D. The two rules defining this predicate simulate the transducer $T_{transcribe}$ in Example 7.1. The second rule initiates the simulation, and the third rule carries it out, by recursively scanning each DNA sequence while constructing an RNA sequence. For each character in a DNA sequence, its transcription, T, is concatenated to the growing RNA sequence. The last four rules specify the transcription of individual characters. *i.e.*, The formula $trans(d, r)$ means that d is a character in the DNA alphabet, and r is its transcription in the RNA alphabet. □

Corollary 3 (Equivalence) *Transducer Datalog and Sequence Datalog are expressively equivalent; i.e., every database query expressible in Transducer Datalog can be expressed in Sequence Datalog, and vice-versa. Moreover, the equivalence preserves finiteness; i.e., if a program in Transducer Datalog has finite semantics, then the equivalent program in Sequence Datalog has finite semantics too, and vice-versa.*

Proof: One direction is proved by Theorem 7. In the other direction, given a Sequence Datalog program, we construct an equivalent Transducer Datalog program by simply replacing each constructive sequence term, $S_1 \bullet S_2$, by the transducer term $T_{append}(S_1, S_2)$, as described in Section 7.1. This transformation clearly preserves finiteness. □

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

8 A Safe Query Language for Sequences

This section develops syntactic restrictions that define a sublanguage of Transducer Datalog, called *Strongly Safe Transducer Datalog*, that is both finite and highly expressive. The restrictions forbid recursion through transducer terms. Intuitively, this ensures that the transducer network corresponding to a program is acyclic. The syntactic restrictions are defined in terms of predicate dependency graphs. These graphs represent dependencies between predicates in rule heads and rule bodies. To keep the development simple, we assume that all programs are guarded. By Theorem 10 in Appendix A, this assumption does not result in any loss of expressibility.

Definition 8 (Dependent Predicates) *Let P be a Transducer Datalog program. A predicate symbol p* depends *on predicate symbol q in program P if for some rule in P, p is the predicate symbol in the head and q is a predicate symbol in the body. If the rule is constructive, then p* depends constructively *on q.*

Definition 9 (Dependency Graph) *Let P be a Transducer Datalog program. The predicate dependency graph of P is a directed graph whose nodes are the predicate symbols in P. There is an 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.*

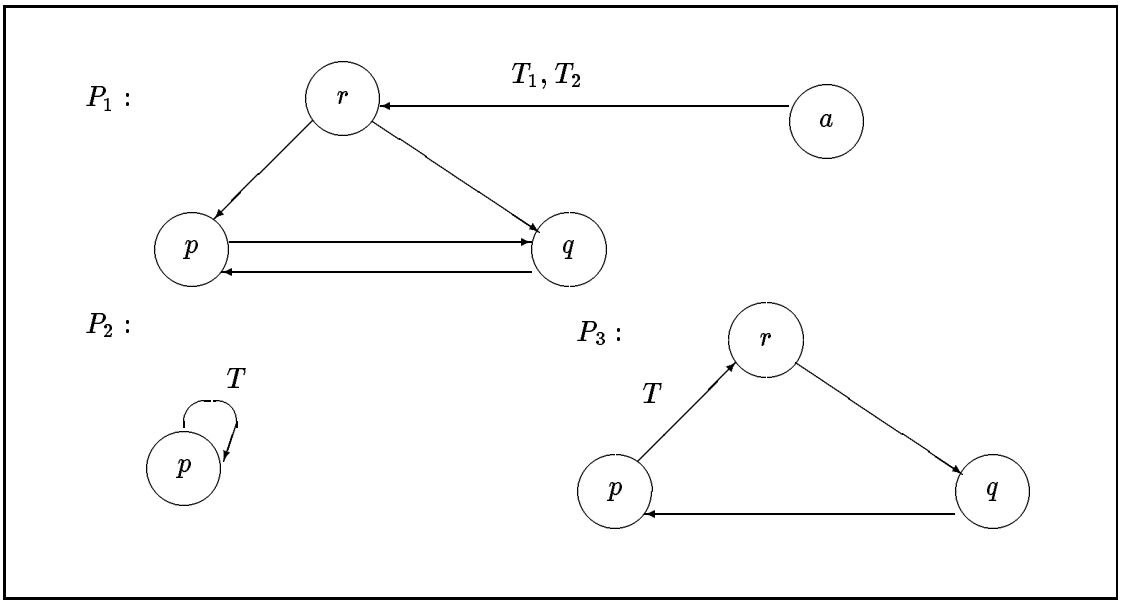

Figure 4: Predicate dependency graphs

These three graphs correspond to the three programs in Example 8.1. Constructive edges are labelled by transducer names.

Definition 10 (Strongly-Safe Programs) *A Transducer Datalog program is* strongly safe *if its predicate dependency graph does not contain any constructive cycles.*

The programs in Examples 7.1 and 7.3 are strongly safe since they are non-recursive, and thus their dependency graphs contain no cycles. In the following example, all the programs are recursive, and one of them is strongly safe.

Example 8.1 Consider the following three Transducer Datalog programs, P_1, P_2 and P_3:

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

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

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

All three programs are recursive, so their predicate dependency graphs all have cycles. (See Figure 4.) The graphs of P_2 and P_3 have constructive cycles, while the graph of P_1 does not. Thus, P_1 is strongly safe, while P_2 and P_3 are not. □

8.1 Finiteness of Strongly Safe Programs

Because strongly safe programs are acyclic wrt transducer calls, their semantics is finite; that is, they have a finite minimal model for every database. We prove this result for programs of order $k \leq 3$. In fact, we establish stronger results than mere finiteness. By considering programs of order 2 and order 3 separately, we establish tight bounds on the size of their minimal models. The proofs can be extended to higher-order programs, but that is beyond the scope of this paper.

Definition 11 (Database Size) *The* size *of a database (or a finite interpretation) is the number of sequences in its extended active domain.*

Theorem 8 *Let P be a Transducer Datalog program that is strongly safe and of order 2. For any database* DB*, the size of the minimal model of P and* DB *is polynomial in the size of* DB*.*

Proof: Without loss of generality, we can assume that transducer terms in P are not nested. Thus, P does not contain rules such as $p(T_1(T_2(X))) \leftarrow q(X)$. Instead, such rules can be decomposed into simpler ones, such as the following:

$$p(T_1(Y)) \leftarrow r(Y) \qquad\qquad r(T_2(X)) \leftarrow q(X)$$

where r is a new predicate symbol. This decomposition can only *increase* the extended active domain of the minimal model. In this example, both the original rule and the decomposed rules contribute ground instances of $T_1(T_2(X))$ to the extended active domain, while the decomposed rules also contribute ground instances of $T_2(X)$. We can therefore assume that all transducer terms in P have the form $T(s_1, \ldots, s_k)$ where each s_i is a non-constructive sequence term.

The rest of 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 $i \in \{1, \ldots, k\}$ 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 $\mathcal{N}_1, \ldots, \mathcal{N}_k$. 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 $P = P_1 \cup P_2 \cup \cdots \cup P_k$.

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 $P_1 \cup \cdots \cup P_i$. 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 extended active domain. Thus, $\mathcal{M}_{i+1}$ and $\mathcal{M}_i'$ have the same extended active domain, and all new sequences are created in computing $\mathcal{M}_i'$ from $\mathcal{M}_i$. We shall show that the size of $\mathcal{M}_i'$ is at most polynomial in the size of $\mathcal{M}_i$. Let n_i' and n_i denote these two sizes, respectively. Also, let l_i' and l_i denote the maximum lengths of any sequence in the extended active domains of $\mathcal{M}_i'$ and $\mathcal{M}_i$, respectively. We first derive upper bounds for l_i' and n_i' in terms of l_i and n_i, from which we derive an upper bound for n_i' in terms of n_i.

To bound l_i', observe that $\mathcal{D}^{ext}_{\mathcal{M}_i'}$ consists of the sequences in $\mathcal{M}_i$, plus the sequences created by transducer terms in P_i^c, plus all their contiguous subsequences. By hypothesis, each transducer in P_i^c has order at most 2. By Theorem 4, such transducers create sequences of at most polynomial length, polynomial in the length of the longest input. Thus, the longest sequence created by any transducer in P_i^c is polynomial in the length of the longest sequence in $\mathcal{M}_i$. Thus $l_i' = l_i^{O(1)}$.

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 $\mathcal{D}^{ext}_{\mathcal{M}_i}$. 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. Each output sequence can contribute up to $O(l_i'^2)$ sequences and subsequences to $\mathcal{D}^{ext}_{\mathcal{M}_i'}$. The transducer term thus contributes at most $n_i^k \cdot O(l_i'^2) \leq n_i^{O(1)} \cdot l_i'^{O(1)}$ sequences to $\mathcal{D}^{ext}_{\mathcal{M}_i'}$. This is true for each transducer term in P_i^c. In addition, $\mathcal{D}^{ext}_{\mathcal{M}_i'}$ contains all the sequences in $\mathcal{D}^{ext}_{\mathcal{M}_i}$. Thus, if P_i^c has m transducer terms, then

$$n_i' \;\leq\; m \cdot n_i^{O(1)} \cdot l_i'^{O(1)} + n_i \;=\; n_i^{O(1)} \cdot l_i'^{O(1)}$$

since m is fixed. Combining this with the upper bound on l_i', we get $n_i' \leq n_i^{O(1)} l_i^{O(1)}$.

An extended active domain includes all the subsequences of its longest sequence, of which there are quadratically many. Thus $n_i \geq O(l_i^2)$ so $l_i \leq O(n_i^{1/2}) \leq n_i^{O(1)}$. Thus $n_i' \leq n_i^{O(1)}$. Each stratum of program P thus increases the size of the minimal model by at most a polynomial. Thus, since the number of strata is fixed, the size of the minimal model of P and DB is at most polynomial in the size of DB. □

A similar result holds for programs of order 3:

Theorem 9 *Let P be a Transducer Datalog program that is strongly safe and of order 3. For any database* DB*, the size of the minimal model of P and* DB *is hyper-exponential in the size of* DB*.*

Proof: The proof is similar to that of Theorem 8. In particular, the following upper bounds are still valid, since they are independent of the order of the transducers:

$$n_i' \leq n_i^{O(1)} \cdot l_i'^{O(1)} \qquad\qquad l_i \leq n_i^{O(1)}$$

When the transducers are of order 3, we get additional bounds. By Theorem 4, such transducers can generate sequences of hyper-exponential length. Thus $l_i' \leq hyp_m(l_i^{O(1)})$ for some integer m. Combining all these inequalities, we get the following upper bound on n_i':

$$n_i' \leq n_i^{O(1)} \cdot hyp_m(l_i^{O(1)}) \leq n_i^{O(1)} \cdot hyp_m(n_i^{O(1)}) = hyp_m(n_i^{O(1)})$$

Each stratum of program P thus increases the size of the minimal model by at most a hyper-exponential. Thus, since the number of strata is fixed, the size of the minimal model of P and DB is at most hyper-exponential in the size of DB. □

We can now state the finiteness result for strongly safe programs.

Corollary 4 (Finiteness) *If a Transducer Datalog program of order at most 3 is strongly safe, then it has a finite semantics.*

Proof: Follows immediately from Theorems 8 and 9, since a model is finite iff its extended active domain in finite. □

8.2 Expressibility of Strongly Safe Programs

Building on Theorems 8 and 9, this section establishes expressibility results for strongly safe programs.

Corollary 5 *Strongly Safe Transducer Datalog programs of order* 2 *express exactly the class of sequence functions computable in* PTIME.

Proof: It is not hard to show that the computation of any acyclic network of transducers of order k can be simulated by a Strongly Safe Transducer Datalog program of order k. When $k = 2$, these programs can express any sequence function in PTIME, by Theorem 5. This proves the lower expressibility bound.

To prove the upper bound, let DB be a database, and let P be a Strongly Safe Transducer Datalog program of order 2. Theorem 9 guarantees that the size of the minimal model $T_{P,\mathrm{DB}} \uparrow \omega$ is polynomial in the size of DB. In general, each application of the operator $T_{P,\mathrm{DB}}$, except the last, adds at least one new atom to the minimal model. Thus, the minimal model is computed after at most polynomially many applications of $T_{P,\mathrm{DB}}$. Moreover, each application of $T_{P,\mathrm{DB}}$ takes polynomial time, since the input and output are of polynomial size, and each ground instance of each transducer term can be evaluated in polynomial time, since the transducers are of order 2. The entire fixpoint computation thus takes at most polynomial time, polynomial in the size of DB. This is true for any database, DB.

Since P computes a sequence function, DB contains just a single atom, $in(\sigma)$, where σ is the input sequence for the function. The size of DB is the number of sequences in its extended active domain. In this case, the size is just the number of subsequences of σ, that is, $O(n^2)$ where n is the length of σ. Thus, the minimal model can be computed in time that is polynomial in n^2, and thus polynomial in n. □

Corollary 6 *Strongly Safe Transducer Datalog programs of order* 3 *expresses exactly the class of elementary sequence functions.*

Proof: Similar to the proof of Corollary 5. In this case, by Theorem 9 and Theorem 4, the size of the minimal model and the running time of each transducer are hyper-exponential. This yields the upper bound. □

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.

References

[1] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Comp. and System Sc.*, 43(1):62–124, August 1991.

[2] A. Albano, L. Cardelli, and R. Orsini. Galileo: a strongly typed interactive conceptual language. *ACM Trans. on Database Syst.*, 10(2):230–260, June 1985.

[3] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and Z. Zdonik. The object-oriented database manifesto. In *First Intern. Conference on Deductive and Object Oriented Databases (DOOD'89), Kyoto, Japan*, 1989.

[4] P. Atzeni, editor. *LOGIDATA+: Deductive Databases with Complex Objects, Lecture Notes in Computer Science 701.* Springer-Verlag, 1993.

[5] 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)*, 1989.

[6] S. Bellantoni and S. Cook. A new recursion-theorethic characterization of the polytime functions. In *ACM Intern. Symposium on Theory of Computing*, 1992.

[7] A. J. Bonner. Hypothetical Datalog: complexity and expressibility. *Theoretical Computer Science*, 76:3–51, 1990.

[8] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Third Intern. Workshop on Database Programming Languages (DBPL'91)*, pages 9–19, 1991.

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

[10] L.S. Colby, E.L. Robertson, L.V. Saxton, and D. Van Gucht. A query language for list-based complex objects. In *Thirteenth ACM SIGMOD Intern. Symposium on Principles of Database Systems (PODS'94)*, pages 179–189, 1994.

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

[12] 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*, pages 293–300, 1992.

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

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

[15] E. Graedel and M. Otto. Inductive definability with counting on finite structures. In *Proc. of Computer Science Logic*, pages 231–247. Lecture Notes in Computer Science 702, 1993.

[16] 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.

[17] S. Grumbach and T. Milo. Towards tractable algebras for bags. In *Twelfth ACM SIGMOD Intern. Symposium on Principles of Database Systems (PODS'93), Washington, DC*, pages 49–58, 1993.

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

[19] 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.

[20] R. Hull and J. Su. On the expressive power of database queries with intermediate types. *Journal of Comp. and System Sc.*, 43(1):219–267, 1993.

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

[22] G. Mecca and A.J. Bonner. Generalized sequence transducers. In preparation.

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

[24] J. Richardson. Supporting lists in a data model (a timely approach). In *Eighteenth International Conference on Very Large Data Bases (VLDB'92), Vancouver, Canada*, pages 127–138, 1992.

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

[26] D. Stott Parker, E. Simon, and P. Valduriez. SVP – a model capturing sets, streams and parallelism. In *Eighteenth International Conference on Very Large Data Bases (VLDB'92), Vancouver, Canada*, pages 115–126, 1992.

[27] The Committee for advanced DBMS functions. Third generation database systems manifesto. *ACM SIGMOD Record*, 19(3):31–44, 1990.

[28] S. Vandenberg and D. De Witt. Algebraic support for complex objects with arrays, identity and inheritance. In *ACM SIGMOD International Conf. on Management of Data*, 1991.

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

[30] X. Wang. *Pattern matching by RS-operations: Towards a unified approach to querying sequence data.* PhD thesis, University of Southern California, 1992.

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

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

A Appendix: Guarded Programs

This appendix uses the declarative semantics developed in Sections 3.3 and 3.4 to prove a basic result about Sequence Datalog and Transducer Datalog. This result, which was used in Section 7.2, allows us to assume that programs are guarded. A program is guarded if all its clauses are guarded, and a clause is guarded if every sequence variable in the clause appears in the body of the clause as an argument of some predicate.

Theorem 10 *In Sequence Datalog and in Transducer Datalog, for any program P, there is guarded program P^G that expresses the same sequence queries. That is, for any database* DB*, and any predicate $p(X_1, \ldots, X_n)$ mentioned in $P \cup$ DB,*

$$P, \text{DB} \models p(\sigma_1, \ldots, \sigma_n) \qquad \textit{iff} \qquad P^G, \text{DB} \models p(\sigma_1, \ldots, \sigma_n)$$

Moreover, P has a finite semantics over DB *if and only if P^G does.*

We shall prove Theorem 10 in a series of short lemmas.

The construction of P^G is simple. The first step is to introduce a new predicate, $dom(X)$. Intuitively, this predicate means that X is a sequence in the extended active domain. Next, for each clause γ in P, we add the following guarded clause to P^G:

$$\text{HEAD}(\gamma) \leftarrow \text{BODY}(\gamma),\ dom(X_1),\ dom(X_2), \ldots, dom(X_n). \tag{1}$$

where $X_1, X_2, \ldots, X_n$ are all the sequence variables in γ.

The predicate $dom(X)$ is defined by guarded clauses in P^G. First, we add the following clause to P^G:

$$dom(X[M, N]) \leftarrow dom(X) \tag{2}$$

This clause ensures that for each sequence in the extent of dom, all its subsequences are also there. Second, if $p(X_1, X_2, \ldots, X_n)$ is a base predicate[7] or a predicate mentioned in P, then we add the following n clauses to P^G:

$$\begin{aligned} dom(X_1) &\leftarrow p(X_1, X_2, \ldots, X_n) \\ dom(X_2) &\leftarrow p(X_1, X_2, \ldots, X_n) \\ &\cdots \\ dom(X_n) &\leftarrow p(X_1, X_2, \ldots, X_n) \end{aligned} \tag{3}$$

[7] Recall from Section 3.1 that the set of base predicates is fixed and finite.

We must now show that P and P^G express the same queries. To do this, we introduce two operations on interpretations. Intuitively, these operations allow us to transform models of P into models of P^G, and vice-versa. This will allow us to establish a correspondance between the models of P and P^G, which is the basis for their expressive equivlance.

Definition 12 *Let I be an interpretation. I^- is an interpretation constructed from I by removing all atoms of the form $dom(\sigma)$. I^+ is an interpretation constructed from I by adding atoms of the form $dom(\sigma)$ for every sequence σ in $\mathcal{D}_I^{ext}$, the extended active domain of I.*

Observe that I^- and I^+ have the following basic properties:

- $\mathcal{D}_{I^-}^{ext} \subseteq \mathcal{D}_I^{ext}$
- $\mathcal{D}_{I^+}^{ext} = \mathcal{D}_I^{ext}$
- $(I^+)^- \subseteq I$
- If $I_1 \subseteq I_2$ then $I_1^- \subseteq I_2^-$ and $I_1^+ \subseteq I_2^+$

The following lemmas establish other properties of these two operations, properties that lead directly to a proof of Theorem 10.

Lemma 5 *I is a model of $P \cup \text{DB}$ if and only if I^+ is a model of $P^G \cup \text{DB}$.*

Proof: First, let γ be any clause of the form (2) or (3), and let θ be any substitution based on $\mathcal{D}_I^{ext}$ and defined at γ. Then $\theta(\text{HEAD}(\gamma))$ has the form $dom(\sigma)$, for some sequence σ in $\mathcal{D}_I^{ext}$. Thus $\theta(\text{HEAD}(\gamma)) \in I^+$, so I^+ is a model of γ. I^+ is therefore a model of clauses (2) and (3).

Next, let γ be any clause in $P \cup \text{DB}$. We must show that I is a model of γ if and only if I^+ is a model of clause (1). To see this, note that $\theta(dom(X)) \in I^+$ for any sequence variable X, and any substitution θ based on $\mathcal{D}_I^{ext}$. Thus, all premises of the form $dom(X)$ in clause (1) are satisfied in I^+. □

Lemma 6 *Suppose I is constructed only from predicates mentioned in $P^G \cup \text{DB}$. If I is a model of $P^G \cup \text{DB}$, then I^- is a model of $P \cup \text{DB}$.*

Proof: I is a model of clauses (2) and (3). By clauses (3), if σ is a sequence in the active domain of I, then $dom(\sigma) \in I$. Thus, by clauses (2), if σ is a sequence in the extended active domain of I, then $dom(\sigma) \in I$. Thus, $\theta(dom(X)) \in I$ for any sequence variable X, and any substitution θ based on $\mathcal{D}_I^{ext}$.

Let γ be a clause in $P \cup \text{DB}$. We must show that I^- is a model of γ. To see this, note that in clause (1), all premises of the form $dom(X)$ are satisfied in I. Thus, I is a model of clause (1) if and only if I^- is a model of γ. But I *is* a model of clause 1, since this clause is in P^G, and I is a model of $P^G \cup \text{DB}$ by hypothesis. □

Lemma 7 *If I is the minimal model of $P^G \cup \text{DB}$, then I^- is the minimal model of $P \cup \text{DB}$, and I and I^- have the same extended active domain.*

Proof: Since I is the minimal model of $P^G \cup \text{DB}$, it is constructed only from predicates mentioned in $P^G \cup \text{DB}$. Thus, I^- is a model of $P \cup \text{DB}$, by Lemma 6. Thus, letting I_0 be the minimal model of $P \cup \text{DB}$, we have

1. $I_0 \subseteq I^-$
2. $I \subseteq I_0^+$, since I_0^+ is a model of $P^G \cup \text{DB}$, by Lemma 5, and I is the minimal model.
3. $I^- \subseteq I_0$, since $I^- \subseteq (I_0^+)^-$ by item 2, and $(I_0^+)^- \subseteq I_0$ in general.
4. $I_0 = I^-$, by items 1 and 3.

This proves the first part of the lemma. To prove the second part, observe that $\mathcal{D}_{I^-}^{ext} \subseteq \mathcal{D}_I^{ext}$. To prove the reverse containment, note that $(I^-)^+$ is a model of $P^G \cup \text{DB}$, by Lemma 5. Thus $I \subseteq (I^-)^+$ since I is the minimal model. Thus $\mathcal{D}_I^{ext} \subseteq \mathcal{D}_{(I^-)^+}^{ext} = \mathcal{D}_{I^-}^{ext}$. □

Theorem 10 follows immediately from Lemma 7 and Corollaries 1 and 2.