Expressiveness within Sequence Datalog

HEBA AAMER, Computer Science, SOFT Lab, Vrije Universiteit Brussel, Brussel, Belgium
JAN HIDDERS*, Birkbeck University of London, London, United Kingdom of Great Britain and Northern Ireland
JAN PAREDAENS, University of Antwerp, Antwerpen, Belgium
JAN VAN DEN BUSSCHE, Data Science Institute, Hasselt University, Hasselt, Belgium

Motivated by old and new applications, we investigate Datalog as a language for sequence databases. We reconsider classical features of Datalog programs, such as negation, recursion, intermediate predicates, and relations of higher arities. We also consider new features that are useful for sequences, notably, equations between path expressions, and "packing". Our goal is to clarify the relative expressiveness of all these different features, in the context of sequences. Towards our goal, we establish a number of redundancy and primitivity results, showing that certain features can, or cannot, be expressed in terms of other features. These results paint a complete picture of the expressiveness relationships among all possible Sequence Datalog fragments that can be formed using the six features that we consider.

CCS Concepts: • **Theory of computation → Database query languages (principles)**.

Additional Key Words and Phrases: path variables, solving word equations, stratified negation

1 Introduction

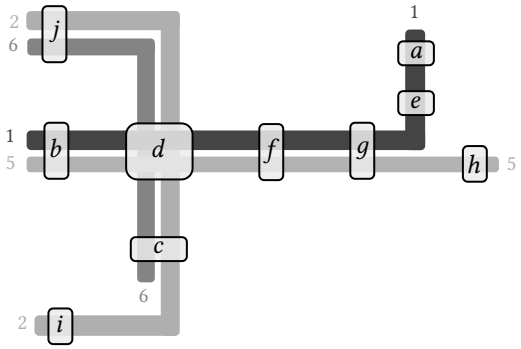
Interest in sequence databases dates back for at least three decades [14]. For clarity, here, by sequence databases, we do not mean relations where the tuples are ordered by some sequence number or timestamp, possibly arriving in a streaming fashion (e.g., [13, 29, 40, 44]). Rather, we mean databases that allow the management of large *collections of sequences*.

Example 1.1. To illustrate the idea of a sequence database, consider the following figure depicting a fragment of the metro network of Brussels.

*Jan Hidders is the corresponding author.

Authors' Contact Information: Heba Aamer, Computer Science, SOFT Lab, Vrije Universiteit Brussel, Brussel, Belgium; e-mail: heba.mohamed@vub.be; Jan Hidders, Birkbeck University of London, London, London, United Kingdom of Great Britain and Northern Ireland; e-mail: j.hidders@bbk.ac.uk; Jan Paredaens, University of Antwerp, Antwerpen, Belgium; e-mail: jan.paredaens@uantwerpen.be; Jan Van den Bussche, Data Science Institute, Hasselt University, Hasselt, Limburg, Belgium; e-mail: jan.vandenbussche@uhasselt.be.

ACM 1557-4644/2025/5-ART
https://doi.org/10.1145/3732283



In the figure, the stations are named $a, \ldots, j$ instead of their actual names for brevity, and the numbers refer to the line numbers of the different metro lines.

This network can be naturally expressed as a binary relation M where the first component represents the line number while the second component represents to the full sequence of stations of that line. The relation M that corresponds to the above depicted network will have the following set of tuples (where the dot represents concatenation).

	M
1	$b \cdot d \cdot f \cdot g \cdot e \cdot a$
2	$j \cdot d \cdot c \cdot i$
5	$b \cdot d \cdot f \cdot g \cdot h$
6	$j \cdot d \cdot c$

In the early years, sequence databases were motivated by applications in object-oriented software engineering [6] and in genomics [10, 27]. While these applications remain relevant, more recent applications of sequence databases include the following.

- Process mining [26] operates on event logs, which are sets of sequences. Thus, sequence databases, and sequence database query languages, can serve as enabling technology for process mining and compliance monitoring [2]. For example, a typical query one may want to be able to support is look for all logs in which every occurrence of 'complete order' is followed by 'receive payment'.
- Graph databases have as main advantage over relational databases that they offer convenient query primitives for retrieving paths. Paths are, of course, sequences. For example, the G-CORE graph query language proposal [30] supports the querying of sequences stored in the database, separately from the graph; these sequences do not even have to correspond to actual paths in the graph. An example query in such a context could be to return the nodes that belong to all paths in a given set of paths. We thus see that a full implementation of G-CORE must be a sequence database!
- JSON Schema [37] is based on the notion of JSON pointers, which are sequences of keys navigating into nested JSON objects. The work on J-Logic [23] has showed that modeling JSON databases as sequence databases is very convenient for defining JSON-to-JSON transformations in a logical, declarative manner. For a simple example, consider a JSON object *Sales* that is a set of key–value pairs, where keys are items; the value for an item is a nested object holding the sales volumes for the item by year. Specifically, the nested object is again a set of key–value pairs, where keys are years and values are numbers. We can naturally view *Sales* as a set of length-3 sequences of the form item–year–value. Restructuring the object to group sales by year, rather than by item, then simply amounts to swapping the first two elements of every sequence. For another example, checking if two (nested) JSON objects are deep-equal amounts to

checking equality of the corresponding sets of sequences. So, again, expressive querying of JSON objects requires a sequence database.
- Logical approaches to information extraction [18, 45] model the result of an information extraction as a sequence database.

Given the importance of sequences in various advanced database applications, our research goal in this paper is to obtain a thorough understanding of the role that different language features play in querying sequence databases. For such an investigation, we need an encompassing query language in which these features are already present, or can be added. For this purpose we adopt Datalog, a logical framework that is well established in database theory research, and that has continued practical relevance [5, 8, 15].

Indeed, Datalog for sequence databases, or Sequence Datalog, was already introduced and studied by Bonner and Mecca in the late 1990s [10, 33]. They showed that, to make Datalog work with sequence databases, all we have to do is to add terms built from sequence variables using the concatenation operator. In our work we refer to such terms as *path expressions* and refer to sequence variables as *path variables*.[1] Bonner and Mecca studied computational completeness, complexity, and termination guarantees for Sequence Datalog, and showed how to combine Sequence Datalog with subcomputations expressed using transducers.

Sequence Datalog was recently also considered for information extraction ("document spanners"), with regular expression matching built-in as a primitive [34, 36]. Such regular expressions may be viewed as very useful syntactic sugar, as they are also expressible using recursion. Adding regular matching directly may be compared to Bonner and Mecca's transducer extensions; the PTIME capturing result reported by Peterfreund et al. [36] may be compared to Corollary 3 of Bonner and Mecca [10].

In the present work, we study the relative expressiveness of query language features in the context of Sequence Datalog. Some of the features we consider are standard Datalog, namely, recursion, stratified negation, and intermediate predicates.[2] The latter feature actually comprises two features, since we distinguish between monadic intermediate predicates and intermediate predicates of higher arities. While we omit regular expression matching as a feature, we consider two further features that are specific to sequences:

- Equalities between path expressions, which we call *equations*, allow for the elegant expression of pattern matching on sequences.
- *Packing*, a feature introduced in J-Logic, is a versatile tool that allows for subsequences to be "bracketed" and temporarily treated as atomic values; they can be unpacked later. Intuitively, sequences with packed values can be seen as nested sequences, in the spirit of nested lists having sub-lists as elements.

The standard Datalog features, whose expressiveness is well understood on classical relational structures [4, 17], need to be re-examined in the presence of sequences; moreover, their interaction with the new features needs to be understood as well. For example, consider recursion versus equations, and the query that checks whether an input sequence $\$x$ consists exclusively of a's. (Path variables are prefixed by a dollar sign.) With an equation we can simply write $\$x \cdot a = a \cdot \$x$ (using the dot for concatenation). Without equations (or other means to simulate equations), however, this query can only be expressed using recursion. For another example, consider monadic versus higher-arity intermediate predicates. Classically, there are well-known arity hierarchies for Datalog [22]. In our setting, however, a unary relation can already hold arbitrary-length sequences, and indeed, using a simple coding trick, we will see that the arity feature is actually redundant.

[1]We actually work with a minor variant of Bonner and Mecca's language; while they additionally introduce index terms, but only allow path expressions in the heads of rules, we allow path expressions also in rule bodies, and additionally introduce atomic variables. The two variants are equivalent in that one can be simulated by the other requiring no additional features such as negation or recursion.
[2]We remark that intermediate predicates are the IDB relations names that are not the output IDB relation name. Thus, they are used by the program internally but do not form the actual output.

In our work, we have chosen to define expressiveness in terms of the baseline class of "flat unary queries", namely, functions from unary relations to unary relations, where both the input and the output are just sets of plain, unpacked sequences. In this way, we avoid trivial tautologies such as "arity is a primitive feature, because without it, we cannot express queries of higher arities". Similarly, we want to avoid a result of the form "packing is a primitive feature, because without it, we cannot create packed sequences". As a matter of fact, we will show that both arity and packing, although they certainly are convenient features, are actually redundant for expressing these flat unary queries. A result in this direction was already stated for packing in the context of J-Logic [23], but the technique used there to simulate packing requires recursion. In the present paper, we show that packing is redundant also in the absence of recursion. Our proof technique leverages associative unification [3], and more specifically, the termination of associative unification for particular cases of word equations [16].

Our further results can be summarized as follows.

(1) At first sight, equations seem to be a redundant feature, at least in the presence of intermediate predicates. Indeed, instead of using an equation $e_1 = e_2$ as a subgoal, we can introduce an auxiliary recursive relation $T(e_1, e_2)$ that axiomatizes the equality relation, and replace the equation by the subgoal $T(e_1, e_2)$. (Our notation here is not precise but hopefully enough to convey the idea.) With negated equations and recursion, however, this simple trick does not work as it violates stratification. We still show, however, that equations are redundant in the presence of both intermediate predicates and negation.
(2) In the absence of intermediate predicates, however, equations are a primitive feature. Indeed, the "only a's" query mentioned above, easily expressed with an equation, is not expressible in the absence of intermediate predicates.
(3) One can also, conversely, simulate intermediate predicates using equations: a simple folding transformation works in the absence of negation and recursion. In the presence of negation or recursion, however, intermediate predicates do add power. This is fairly easy to show for recursion: the squaring query "for every path p in the input, output a^{n^2}, where n is the length of p" requires an intermediate predicate in which the output can be constructed recursively. In the presence of negation, the primitivity of intermediate predicates can be seen to follow from the corresponding result for classical Datalog (by quantifier alternation). Some work has still to be done, however, since the classical proof has to be extended to account for path expressions and equations.
(4) It will not surprise the reader that recursion is primitive in Sequence Datalog. This can be seen in many ways; probably the easiest is to use the above squaring query, and to observe that without recursion, the length of output sequences is at most linear in the length of input sequences. Another proof, that also works for Boolean queries, is by reduction to the classical inexpressibility of graph connectivity in first-order logic. As in the previous paragraph, the reduction must account for the use of path expressions and equations.
(5) A classical fact is that nonrecursive Datalog with stratified negation is equivalent to the relational algebra. We extend the standard relational algebra by allowing path expressions in selection and projection, and adding operators for unpacking and for subsequences. We obtain a language equivalent to nonrecursive Sequence Datalog.

Our results allow us to completely classify the sixteen possible Sequence Datalog fragments in a Hasse diagram with respect to their expressive power, as shown in Figure 1. Some fragments are equivalent, as shown; also, the features for packing and higher-arity intermediate predicates are omitted, since they are redundant independently of the presence or absence of other features.

A conference version of this paper appeared previously [1]. In this version, we remark on the relation between the original definition of Sequence Datalog and our own definition in Section 2.4. Also, we include the following full proofs that were either sketched or entirely omitted previously: Lemmas 4.1, 5.1, 5.4, 5.8, and 7.3; and

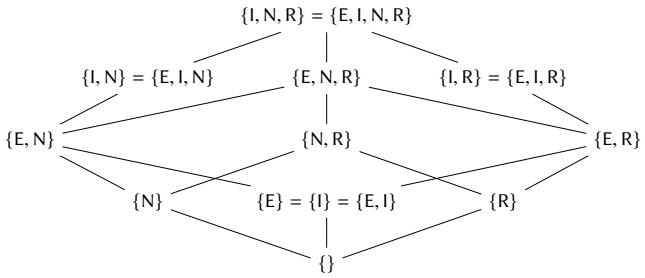


Fig. 1. Relative expressiveness of the different sets of Sequence Datalog features (Negation (N), Equations (E), Intermediate predicates (I), and Recursion (R); features Arity (A) and Packing (P) will turn out to be entirely redundant). An ascending path denotes subsumption; absence of such a path denotes non-subsumption.

Theorems 3.2, 4.17, 5.6, and 7.1. Furthermore, Figure 3 and Examples 1.1, 2.1, 2.4, 3.1, 4.8, 4.9, 4.18, and 4.19 are completely new.

This paper is organized as follows. In Section 2 we define the sequence database model and the syntax and semantics of Sequence Datalog. In Section 3 we introduce the language features and rigorously define what we mean by one fragment (set of features) being subsumed in expressive power by another fragment. Section 4 presents our redundancy (expressibility) results, and Section 5 presents our primitivity (inexpressibility) results. The Hasse diagram of Figure 1 is assembled in Section 6. Section 7 presents the relational algebra for sequence databases. We conclude in Section 8, where we also discuss additional related work.

2 Sequence databases and Sequence Datalog

In this section we formally define the sequence database model and the syntax and semantics of Sequence Datalog. We do assume some familiarity with the basic notions of classical Datalog [4].

2.1 Data model for sequence databases

A *schema* Γ is a finite set of relation names, each name with an associated *arity* (a natural number). We fix a countably infinite universe **dom** of atomic data elements, called *atomic values*. The sets of *packed values*, *values*, and *paths* are defined as the smallest sets satifying the following:

(1) Every atomic value is a value.
(2) Every finite sequence of values is a path. The empty path is denoted by ϵ.
When writing down paths, we will separate the elements by dots, where the $\cdot$ symbol also serves as the usual symbol for concatenation. Recall that concatenation is associative.
(3) If p is a path, then $\langle p \rangle$ is a packed value.
(4) Every packed value is a value.

The set of all paths is denoted by Π.

For example, if a, b and c are atomic values, then $a \cdot b \cdot a$ is a path; $\langle a \cdot b \cdot a \rangle$ is a packed value; and $c \cdot \langle a \cdot b \cdot a \rangle$ is again a path.

An *instance* I of a schema Γ is a function that assigns to each relation name $R \in \Gamma$ a finite n-ary relation on Π, with n the arity of R.

It is natural to identify a value v with the one-length sequence v. In this way, values, in particular atomic values, are also paths. Hence, classical relational database instances are a special case of instances as defined here.

We refer to such instances as *classical*. So, in a classical instance, each relation name R is assigned a finite relation on **dom**.

2.2 Syntax of Sequence Datalog

We assume disjoint supplies of *atomic variables* (ranging over atomic values) and *path variables* (ranging over paths). The set of all variables is also disjoint from **dom**. We indicate atomic variables as $@x$ and path variables as $\$x$. *Path expressions* are defined just like paths, but with variables added in. Formally, we define the set of path expressions to be the smallest set such that:

(1) Every atomic value is a path expression;
(2) Every variable is a path expression;
(3) If e is a path expression, then $\langle e \rangle$ is a path expression;
(4) Every finite sequence of path expressions is a path expression.

A *predicate* is an expression of the form $R(e_1, \ldots, e_n)$, with R a relation name of arity n, and each e_i a path expression. We call e_i the ith component of the predicate. An *equation* is an expression of the form $e_1 = e_2$, with e_1 and e_2 path expressions.

Many of the following definitions adapt well-known Datalog notions to our data model.

An *atom* is a predicate or an equation. A *negated atom* is an expression of the form $\neg A$ with A an atom. We write a negated equation $\neg e_1 = e_2$ also as a nonequality $e_1 \neq e_2$. A *literal* is an atom (also called a positive literal) or a negated atom (a negative literal). A *body* is a finite set of literals (possibly empty). A *rule* is an expression of the form $H \leftarrow B$, where H is a predicate, called the *head* of the rule, and B is a body.

We define the *limited variables* of a rule as the smallest set such that:

(1) every variable occurring in a positive predicate in the body is limited; and
(2) if all variables occurring in one of the sides of a positive equation in the body are limited, then all variables occurring in the other side are also limited.

A rule is called *safe* if all variables occurring in the rule are limited.

Example 2.1. The rule $S(\$x) \leftarrow R(\$x) \wedge \neg P(\$y) \wedge \langle \$x \rangle = \$y \wedge a \cdot \$x = \$z \wedge \$z \cdot \$y = \langle \$u \rangle$ is safe while neither of the following two rules is.

$$S(\$x) \leftarrow R(\$x) \wedge \neg P(\$y) \wedge a \cdot \$x = \$z$$
$$S(\$x) \leftarrow R(\$x) \wedge a \cdot \$x = \$z \wedge \$z \cdot \$y = \langle \$u \rangle$$

A (Sequence Datalog) *program* $\mathbf{P}$ over a schema Γ is a finite set of safe rules such that all the relation names occuring in any of the rules belong to Γ. The relation names occurring in a program are traditionally divided into EDB and IDB relation names. The IDB relation names are the relation names used in the head of some rules; the other relation names are the EDB relation names. Given a program $\mathbf{P}$ over Γ, we use $\mathrm{edb}(\mathbf{P})$ and $\mathrm{idb}(\mathbf{P})$ to refer to the set of EDB relation names and the set of IDB relation names of $\mathbf{P}$, respectively.

A program $\mathbf{P}$ over Γ is called *semipositive* if negated relational atoms are from $\mathrm{edb}(\mathbf{P})$. Finally, in a *stratified* program $\mathbf{P}$ over a schema Γ, the rules can be partitioned into a finite sequence of *strata*, $\mathbf{P_1}, \ldots, \mathbf{P_m}$, such that each *stratum* $\mathbf{P_i}$ is a semipositive program over Γ with $\mathrm{edb}(\mathbf{P_i}) \subseteq \mathrm{edb}(\mathbf{P}) \cup \bigcup_{j<i} \mathrm{idb}(\mathbf{P_j})$ and with $\mathrm{idb}(\mathbf{P_i})$ disjoint from $\mathrm{idb}(\mathbf{P_j})$ for every $i \neq j$. Henceforth, we always use the term program to mean a stratified program unless otherwise specified.

Recall that, in classical Datalog, stratified negation intuitively means that when a negated predicate $\neg R(e_1, \ldots, e_n)$ occurs in some stratum, then no rule in that stratum or later strata can use R in the head predicate. It is easy to see then that classical Datalog programs with stratified negation are a special case of our notion of programs, where the only path expressions used are atomic values or atomic variables.

Example 2.2. An NFA can be represented by a unary relation N (initial states), a ternary relation T (transitions), and a unary relation F (final states). These would be classical relations. Now consider a unary relation R containing paths without packing, i.e., strings of atomic values. Then the following program, consisting of a single stratum, computes in relation A the strings from R that are accepted by the NFA. The program makes use of a ternary relation S that contains the different configurations that the NFA goes through while computing on some string. Thus, if $z \cdot y$ is a sequence of symbols in R, then $S(q, y, z)$ means that after reading the sequence of symbols in z, the NFA is at state q and it remains to read the sequence of symbols in y. Recall that atomic variables are prefixed with @, and path variables with $.

```
S(@q, $x, ϵ) ← R($x), N(@q).
S(@q2, $y, $z·@a) ← S(@q1, @a·$y, $z), T(@q1, @a, @q2).
A($x) ← S(@q, ϵ,$x), F(@q).
```

Example 2.3. Consider unary relations R and S. The following program, again in a single stratum, uses packing and nonequalities to check whether there are at least three different occurrences of a string from S as a substring in strings from R. The Boolean result is computed in the nullary relation A.

```
T($u·<$s>·$v) ← R($u·$s·$v), S($s).
A ← T($x),T($y),T($z), $x≠$y, $x≠$z, $y≠$z.
```

Example 2.4. Consider the binary relation M as described in Example 1.1. The following program, in two strata, checks whether the metro network expressed by relation M is not connected. That is, there are two different stations that are not reachable from each other. The Boolean result is computed in the nullary relation A.

```
S(@s) ← M(@n, $x·@s·$y).
C(@u, @v) ← S(@u), S(@v), M(@n, $x·@u·$y·@v·$z).
C(@u, @v) ← C(@v, @u).
T(@x, @y) ← C(@x, @y).
T(@x, @y) ← T(@x, @z), S(@z, @x).
A ← S(@u), S(@v), ¬T(@u, @v), @u≠@v.
```

Intuitively, in the above program, we compute in relation S that stations that appear in the network. In relation C, we compute pairs of stations that are on the same line. Further, in relation T, we compute the classical reachability relation between the stations of the network. Finally, in A, we check that there network is not connected.

2.3 Semantics

We have defined the notion of instance as an assignment of relations over Π to relation names. A convenient equivalent view of instances is as sets of facts. A *fact* is an expression of the form $R(p_1, \ldots, p_n)$ with R a relation name of arity n, and each p_i a path. An instance I of a schema Γ is viewed as the set of facts $\{R(p_1, \ldots, p_n) \mid R \in \Gamma \text{ and } (p_1, \ldots, p_n) \in I(R)\}$.

A *valuation* ν is a function that maps atomic variables to atomic values and path variables to paths. We say that ν is *appropriate* for a syntactical construct (such as a path expression, a literal, or a rule) if ν is defined on all variables in that syntactical construct. We can apply an appropriate valuation ν to a path expression e by substituting each variable in e by its image under ν and obtain the path $\nu(e)$. Likewise, we can apply an appropriate valuation to a predicate and obtain a fact.

Let L be a literal, ν a valuation appropriate for L, and I an instance. The definition of when I, ν satisfies L is as expected: if L is a predicate, then the fact $\nu(L)$ must be in I; if L is an equation $e_1 = e_2$, then $\nu(e_1)$ and $\nu(e_2)$ must

be the same value; and if L is a negated atom $\neg A$, then I, ν must not satisfy A. A body B is satisfied by I, ν if all its literals are. Now a rule $r = H \leftarrow B$ is satisfied in I if for every valuation ν appropriate for r such that I, ν satisfies B, also I, ν satisfies H.

Let $\mathbf{P}$ be a semipositive program over Γ, and let I be an instance over $\text{edb}(\mathbf{P})$. Then, the output of the program $\mathbf{P}$ on the instance I, denoted $\mathbf{P}(I)$, is the smallest instance over Γ (specifically, $\text{edb}(\mathbf{P}) \cup \text{idb}(\mathbf{P})$) that satisfies all the rules of $\mathbf{P}$, and that agrees with I on $\text{edb}(\mathbf{P})$. Consequently, for a stratified program $\mathbf{P} := \mathbf{P_1}, \ldots, \mathbf{P_m}$ over Γ, we define $\mathbf{P}(I) = \mathbf{P_m}(\cdots \mathbf{P_2}(\mathbf{P_1}(I)))$.

Due to recursion, for some programs or instances, $\mathbf{P}(I)$ may be undefined, since instances are required to be finite. We also say in this case that $\mathbf{P}$ does not *terminate* on I. If, in the course of evaluating a program $\mathbf{P}$ with several strata on an instance I, one of the strata does not terminate, we agree that the entire program $\mathbf{P}$ is undefined on I. As mentioned in the Introduction, Bonner and Mecca have done substantial work on the question of guaranteeing termination for Sequence Datalog programs. In this paper, we only consider programs that always terminate.

Example 2.5. The program from Example 2.2, while recursive, is guaranteed to terminate on every instance. Indeed, this can be easily verified since the sequences of the second component of the S relation are guaranteed to decrease in length upon applying the recursive rule. Thus, the number of applications of the recursive rule is bounded by the length of the sequences in the input relation R. In contrast, the following two-rule program will not terminate on any instance:

```
T(a).
T(a·$x) ← T($x).
```

It worth noting that in the rest of the paper we no longer remark that our programs satisfy the above condition of termination since in most of the cases, the form of the recursive rules is restricted in a way similar to the recursive rule of the program from Example 2.2. That is, one of the components of the recursive predicate is strictly decreasing in the recursive rule. In the other cases, the sequences appearing in the result of the recursive rule are appearing in EDB predicates and hence these sequences are bounded by the input relations.

2.4 Relation to original Sequence Datalog features

Before we investigate our main research question in this paper, we briefly show that the Sequence Datalog language definition presented in this paper is not a restriction of what was originally defined in the works by Bonner and Mecca [10, 33]. They use two constructs in the original definition of Sequence Datalog that we did not mention in our variant of the language. These are the *indexed sequence terms* and *interpreted transducer terms*.

It is well established that every sequence datalog program that uses interpreted transducer terms is equivalent to another that does not use these terms [10]. Thus, we only focus on indexed sequence terms. Given that s is a sequence (or a path variable), an indexed sequence term has the form $s[i : j]$ or $s[i]$, where i and j can be any of the following:

- a numeric constant value such as 1 or 4;
- a position variable such as N or M;
- the keyword *end* which represents that last position of the sequence; or
- a numeric expression built using other (basic) expressions with the operators + and − such as $end - 2 + N$.

Intuitively, $s[i : j]$ denotes the contiguous subsequence of s starting from position i to (and including) position j. Formally, $s[i : j]$ can only be evaluated under valuations ν that assign a path to the path variable s as before, but now also assign natural numbers to position variables in such a way that $\nu(i)$ and $\nu(j)$ are positions in $\nu(s)$ with $\nu(i) \leq \nu(j) + 1$ and with neither $\nu(i)$ nor $\nu(j)$ may exceed the length of the path $\nu(s)$ (otherwise, the

indexed sequence term	equivalent path expression	extra conditions
$s[1]$	$@u$	$s = @u \cdot \$x$
$s[1:3]$	$@u_1 \cdot @u_2 \cdot @u_3$	$s = @u_1 \cdot @u_2 \cdot @u_3 \cdot \$x$
$s[N]$	$@u$	$s = \$x \cdot @u \cdot \$y$
$s[5:end]$	$\$x$	$s = @u_1 \cdot @u_2 \cdot @u_3 \cdot @u_4 \cdot \$x$
$s[N:end-2]$	$\$y$	$s = \$x \cdot \$y \cdot @u_1 \cdot @u_2$
$s[N+1:M]$	$\$y$	$s = \$x \cdot \$y \cdot \$z$

Table 1. Examples of some indexed sequence terms and their equivalents using path expressions.

indexed sequence term is undefined).[3] Moreover, $s[i]$ denotes a subsequence of length one which is the element at position i, so $s[i]$ is a shorthand for $s[i:i]$.

Example 2.6. Let s be the sequence $abcdefg$. Then:

- $s[1]$ evaluates to a.
- $s[1:3]$ evaluates to abc.
- $s[N]$ can evaluate to each of the sequences of $\{a, b, c, d, e, f, g\}$, for the possible values $N = 1, 2, \ldots, 7$.
- $s[5:end]$ evaluates to efg.
- $s[N:end-2]$ can evaluate to each of the sequences of $\{abcde, bcde, cde, de, e, \varepsilon\}$, for the possible values $N = 1, 2, \ldots, 6$.
- $s[N+1:M]$ can evaluate to each of the possible subsequences of $bcdefg$.

Accordingly, we could get each of the results of the evaluated indexed sequences using variables, concatenation, and equations as shown in Table 1.

The simple simulation of position variables using extra path variables, suggested in Table 1, is only sufficient when no position variable is used in different indexed terms.

Example 2.7. Consider the following rule that splits the sequences in the relation R into three partitions:

```
S($x[1:N1], $x[N1+1:N2], $x[N2+1:end]) ← R($x).
```

For this rule, a simple simulation rule works, as we could equivalently rewrite the rule into the following:

```
S($x1, $x2, $x3) ← R($x), $x=$x1·$x2·$x3.
```

or, simply as

```
S($x1, $x2, $x3) ← R($x1·$x2·$x3).
```

The complication happens when we use the same numeric variable in different indexed sequence terms. For example, consider the following rule:

```
S($y[N1:N2]) ←P($x, $y), R($x[1:N1]), R($x[N2:end]), Q($y[1:N1], $y[1:N1+N2]).
```

The two indexed sequence terms `$x[1:N1]` and `$y[1:N1]` in this rule imply an implicit relationship between the lengths of the evaluated sequences. Indeed, we want to ensure that the length of the sequence returned by `$x[1:N1]` is the same as the length of the sequence returned by `$y[1:N1]`.

Using the same path variable to get the prefixes of the sequences `$x` and `$y` with equations, implies that the two prefixes are identical, not only implying the same length constraint. This complication is resolved by

[3]Strictly speaking, we should write $v_s(i)$ and $v_s(j)$, since the value of 'end' always equals the length of $v(s)$.

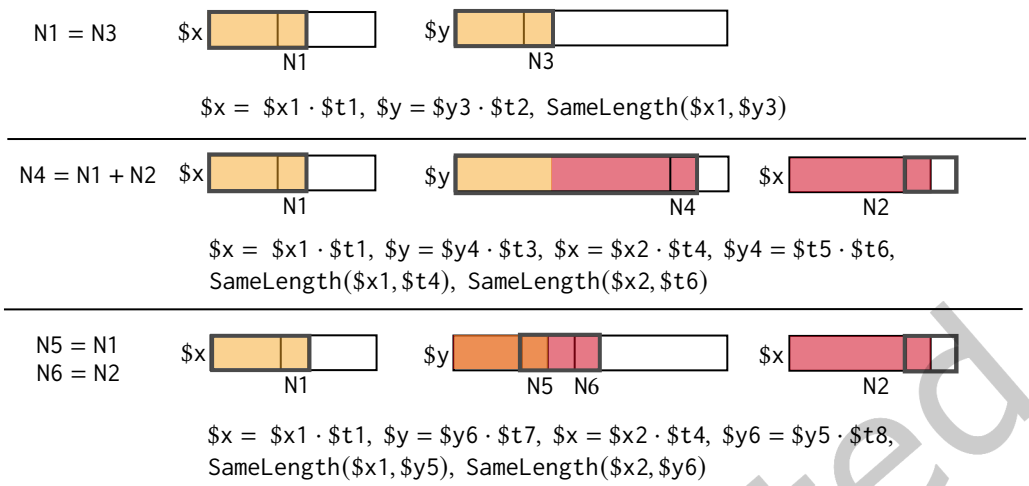


Fig. 2. In this figure, we illustrate what each of equations (mentioned on the left) maps to on the sequences $x and $y. The colored parts are the parts that should have the same length. The bold rectangles surround the parts that should be extracted from the sequences regardless of the length constraints.

introducing a new predicate that determines whether two sequences have the same length. Because our rules are safe, we know that sequences $x and $y must appear entirely in some predicate (in the previous rule, this was the predicate P). Using this information, we can define the required predicate as follows:

```
SameLength(ϵ,ϵ).
SameLength($x1·@u, $y1·@v) ← P($p1·$x1·@u·$x2, $p2·$y1·@v·$y2),
                             SameLength($x1, $y1).
```

Before, using the SameLength predicate in the translation, we remark that it is always possible to rewrite any rule using indexed sequence terms into another equivalent, where the numeric variables used in the rule are unique except in equations. For example, the rule from our running example is equivalent to the following:

```
S($y[N5:N6]) ← P($x, $y), R($x[1:N1]), R($x[N2:end]), Q($y[1:N3], $y[1:N4]),
               N1=N3, N4=N1+N2, N5=N1, N6=N2.
```

These equations are constraints on the lengths of the different subsequences ensured by the SameLength predicate, which is depicted in Figure 2.

Now, we can fully translate the previous rule into the following:

```
S(@u2·$t8) ← P($x, $y), $x=$x1·$t1, $y=$y3·$t2, $y=$y4·$t3, $x=$x2·$t4,
             $y4=$t5·$t6, $y=$y6·$t7, $y6=$y5·$t8, SameLength($x1, $y3),
             SameLength($x1, $t4), SameLength($x2, $t6),
             SameLength($x1, $y5), SameLength($x2, $y6),
             R($x1), $x2=$t9·@u1, R(@u1·$t4), Q($y3, $y4), $y5=$t10·@u2.
```

The correctness of this translation is justified by Figure 2.

The discussion of this section has given a proof of our claim that indexed sequence terms are expressible in the Sequence Datalog variant we use in this work. Of course, one could add the SameLength predicate as syntactic sugar.

Note that our simulations rely heavily on the SameLength predicate, which we conjecture that we cannot express in our Sequence Datalog variant without recursion. However, such a predicate is easily expressible in the original Sequence Datalog language in at least two ways:

```
SameLength(x1, y1) ← P(x, y), x1=x[N1:M1], y1=y[N2:M2], M1-N1=M2-N2.
```

or

```
SameLength(x1, y1) ← P(x, y), x1=x[N1:M1], y1=y[N2:M2],
                     x1[1:L] = x1, y1[1:L] = y1.
```

3 Features, fragments, and queries

In this paper, we consider six possible features that a program may use. These features are exactly what Sequence Datalog adds to unary unions of conjunctive queries, which are indeed what can be formed in the base language that does not use any of the features. Each feature is identified by a letter, spelled out as follows.

Arity A program *uses arity* (has the A-feature) if it contains at least one predicate of arity greater than one.
Recursion A program *uses recursion* (has the R-feature) if there is a cycle in its dependency graph.[4]
Equations A program *uses equations* (has the E-feature) if it contains at least one equation in some rule.
Negation A program *uses negation* (has the N-feature) if it contains at least one negated atom in some rule.
Packing A program *uses packing* (has the P-feature) if a path expression of the form $\langle e \rangle$ occurs in some rule.
Intermediate predicates A program *uses intermediate predicates* (has the I-feature) if it involves at least two different IDB relation names.

Let $\Phi = \{A, I, R, P, E, N\}$ be the set of all features. A subset of Φ is called *a fragment*. A program **P** is said to belong to a fragment F if it uses only features from F.

Example 3.1. The following program belongs to fragment {E}. It computes, in relation S, all paths from R that consist exclusively of a's.

```
S($x) ← R($x), a·$x=$x·a.
```

The following six programs compute the same query, but each belongs to a different fragment. The first one belongs to fragment {A, I, R}:

```
T($x, $x) ← R($x).
T($x, $y) ← T($x, $y·a).
S($x) ← T($x, ϵ).
```

The second alternative program belongs to fragment {I, E, N}:

```
T($x·@w·$y) ← R($x·@w·$y), @w≠a.
S($x) ← R($x), ¬T($x).
```

The third alternative program belongs to fragment {I, R, P}:

```
T(<$x>·<$x>) ← R($x).
T(<$x>·<$y>) ← T(<$x>·<$y·a>).
S($x) ← T(<$x>·<ϵ>).
```

The fourth alternative program belongs to fragment {A, I}:

[4]The nodes of this graph are the IDB relation names, and there is an edge from R_1 to R_2 if R_2 occurs in the body of a rule with R_1 in its head predicate.

```
T(a·$x, $x) ←R($x).
S($x) ←T($x·a, $x).
```

The fifth alternative program belongs to fragment {I, P}:

```
T(<a·$x>·<$x>) ←R($x).
S($x) ←T(<$x·a>·<$x>).
```

Last but not least, the sixth alternative program belongs to fragment {I}:

```
T(a·$x·$x) ←R($x).
S($x) ←R($x), T($x·a·$x).
```

We can even see from the seven different ways that we have always used either feature E or feature I. So an interesting question, is there a program that computes such query without using neither? We can later see that this is not possible, in general.

3.1 Queries and subsumption among fragments and main theorem

Our goal is to compare the different fragments with respect to their power in expressing queries. Our methodology is to do this relative to a baseline class of queries that do not presuppose any feature to begin with. That is, if a query is expected to compute a binary relation over Π, then it is not possible to investigate whether arity is redundant or not since the output must use that feature. Thus, we next formally define the queries we consider in our investigation.

We call a schema *monadic* if each of its relation names has arity zero or one. Also, we call an instance *flat* if it contains no occurrences of packed values.

Given a monadic schema Γ and relation name $S \notin \Gamma$ of arity at most one, a *query* from Γ to S is a total mapping from flat instances over Γ to flat instances over $\{S\}$. A program $\mathbf{P}$ is said to *compute* such a query Q if

(1) $\mathbf{P}$ is over a schema $\Gamma \cup \Gamma'$, with Γ' being disjoint schema from Γ and with $\text{edb}(\mathbf{P}) \subseteq \Gamma$ and $\text{idb}(\mathbf{P}) \subseteq \Gamma'$;
(2) $\mathbf{P}$ terminates on every flat instance of Γ;
(3) S is an IDB relation of $\mathbf{P}$; and
(4) $\mathbf{P}(I)(S)$ equals $Q(I)$ for every flat instance I of Γ.

We now say that fragment F_1 is *subsumed* by fragment F_2, denoted by $F_1 \leq F_2$, if every query computable by a program in F_1 is also computable by a program in F_2. Note that it is possible, for different F_1 and F_2, that $F_1 \leq F_2$ and $F_2 \leq F_1$. Such two fragments are equivalent in expressive power. There will turn out to be 11 equivalence classes; in Section 6 we will prove the following main theorem that characterizes the subsumption relation as shown in Figure 1.

Theorem 3.2 (Main Theorem). *For any fragments F_1 and F_2, we have $F_1 \leq F_2$ if and only if the following five conditions are satisfied:*

(1) $N \in F_1 \Rightarrow N \in F_2$;
(2) $R \in F_1 \Rightarrow R \in F_2$;
(3) $E \in F_1 \Rightarrow (E \in F_2 \vee I \in F_2)$;
(4) $(I \in F_1 \wedge R \notin F_1 \wedge N \notin F_1) \Rightarrow (I \in F_2 \vee E \in F_2)$;
(5) $(I \in F_1 \wedge (R \in F_1 \vee N \in F_1)) \Rightarrow I \in F_2$.

3.2 Redundancy and primitivity

We will explore the subsumption relation by investigating the redundancy or primitivity of the different features with respect to other features. A feature might be redundant in an absolute sense, in that it can be dropped from any fragment without decrease in expressive power. This is a very strong notion of redundancy, and we cannot expect it to hold for most features. Yet a more relative notion of redundancy may hold, meaning that some feature does not contribute to expressive power, on condition that some other features are already present, or are absent. This leads to the following notions.

Definition 3.3 (Redundancy). Let X be a feature and let Y and Z be sets of features.

- X is redundant if $F \leq F - \{X\}$ for every fragment F.
- X is redundant in the presence of Y if $F \leq F - \{X\}$ for every fragment F such that $Y \subseteq F$.
- X is redundant in the absence of Z if $F \leq F - \{X\}$ for every fragment F such that Z is disjoint from F.
- X is redundant in the presence of Y and absence of Z if $F \leq F - \{X\}$ for every fragment F such that $Y \subseteq F$ and Z is disjoint from F.

Similarly, but conversely, a feature might be primitive in an absolute sense, in that dropping it from a fragment always strictly decreases the expressive power. Then again, for other features only more relative notions of primitivity may hold.

Definition 3.4 (Primitivity). Let X be a feature and let Y and Z be sets of features. Recall that Φ is the set of all features.

- X is primitive if $\{X\} \not\leq \Phi - \{X\}$.
- X is primitive in the presence of Y if $\{X\} \cup Y \not\leq \Phi - \{X\}$.
- X is primitive in the absence of Z if $\{X\} \not\leq \Phi - (\{X\} \cup Z)$.

4 Expressibility results

In this section we show various expressibility results that lead to absolute or relative redundancy results for various features.

4.1 Arity

Using a simple encoding trick we can see that arity is redundant. Indeed, let a and b be two different atomic values. For any paths s_1, s_2, s_1' and s_2', we have the following:

Lemma 4.1. *$(s_1, s_2) = (s_1', s_2')$ if and only if $s_1 \cdot a \cdot s_2 \cdot a \cdot s_1 \cdot b \cdot s_2 = s_1' \cdot a \cdot s_2' \cdot a \cdot s_1' \cdot b \cdot s_2'$.*

Proof. The only-if direction is trivial. For the if-direction, we consider

$$s_1 \cdot a \cdot s_2 \cdot a \cdot s_1 \cdot b \cdot s_2 = s_1' \cdot a \cdot s_2' \cdot a \cdot s_1' \cdot b \cdot s_2'$$

and we observe that a appears in the middle of both sequences. Hence,

(a) $s_1 \cdot a \cdot s_2 = s_1' \cdot a \cdot s_2'$ and
(b) $s_1 \cdot b \cdot s_2 = s_1' \cdot b \cdot s_2'$.

For the sake of contradiction, let us assume $|s_1| < |s_1'|$. Then $s_1' = s_1 \cdot x$ for a nonempty sequence x. Thus, equation (a) can be rewritten as $s_1 \cdot a \cdot s_2 = s_1 \cdot x \cdot a \cdot s_2'$, which simplifies to $a \cdot s_2 = x \cdot a \cdot s_2'$. Hence, the sequence x must start with a. In the same way, however, we can deduce from (b) that x must start with b. Hence, the assumption we made is false.

Analogously, $|s_1| > |s_1'|$ can be seen to be false as well, so we know that $|s_1| = |s_1'|$. Then clearly $|s_2| = |s_2'|$ as well. Hence, from (a) and (b) we get that $s_1 = s_1'$ and $s_2 = s_2'$. □

Using this encoding, arities higher than one can be reduced by one. Since we can do this repeatedly, we obtain:

Theorem 4.2. *Arity is redundant.*

Example 4.3. Consider the following program which computes in S the reversals of the paths in R:

```
T($x, ϵ) ⟵ R($x).
T($x, $y·@u) ⟵ T($x·@u, $y).
S($x) ⟵ T(ϵ, $x).
```

The same query can be expressed without arity as follows:

```
T($x·a·a·$x·b) ⟵ R($x).
T($x·a·$y·@u·a·$x·b·$y·@u) ⟵ T($x·@u·a·$y·a·$x·@u·b·$y).
S($x) ⟵ T(a·$x·a·b·$x).
```

4.2 Equations

In the presence of I and A, positive equations are readily seen to be redundant, by introducing an auxiliary intermediate predicate in the program. We only give an example:

Example 4.4. Recall the program from Example 3.1:

```
S($x) ⟵ R($x), a·$x=$x·a.
```

The same query can be computed without equations as follows:

```
T(a·$x, $x) ⟵ R($x).
S($x) ⟵ T($x·a, $x).
```

This simple method works only in the absence of negation, because, when applied to a negated equation in a rule that belongs to a recursive stratum, stratification is violated. However, negated equations can be handled by another method:

Lemma 4.5. *E is redundant in the presence of I, A and N.*

Proof. Positive equations can be handled as above. For each stratum Δ that contains negated equations, we insert a new stratum Δ', right before Δ, consisting of the following rules. Let ρ be a renaming that maps each head relation name in Δ to a fresh relation name; relation names that occur only in bodies in Δ are mapped to themselves by ρ.

For each rule $H \leftarrow B$ in Δ without negated equations, we add the rule $\rho(H) \leftarrow \rho(B)$ to Δ'.

For each rule $r : H \leftarrow B \wedge e_1 \neq e_1' \wedge \ldots \wedge e_n \neq e_n'$ in Δ with n negated equations, we again add $\rho(H) \leftarrow \rho(B)$ to Δ'. Moreover, using a fresh relation name T, we add the following n rules for $i = 1, \ldots, n$:

$$T(v_1, \ldots, v_m) \leftarrow \rho(B) \wedge e_i = e_i'$$

Here, the v's are all variables appearing in B.

Finally, in Δ, we replace r by the following rule:

$$H \leftarrow B \wedge \neg T(v_1, \ldots, v_m).$$ □

Example 4.6. The following program retrieves in S those paths from R that can be written as $a_1 \cdots a_n b_n \cdots b_1$ with $a_i \neq b_i$ for $i = 1, \ldots, n$:

```
U($x, $x) ←R($x).
U($x, $y) ←U($x, @a·$y·@b), @a≠@b.
S($x) ←U($x, ϵ).
```

Applying the method to eliminate negated equations, we obtain:

```
U1($x, $x) ←R($x).
U1($x, $y) ←U1($x, @a·$y·@b).
T($x, $y, @a, @b) ←U1($x, @a·$y·@b), @a=@b.
S1($x) ←U1($x, ϵ).
U($x, $x) ←R($x).
U($x, $y) ←U($x, @a·$y·@b), ¬T($x, $y, @a, @b).
S($x) ←U($x, ϵ).
```

We remark that the rule defining the relation S_1 is not needed and hence can be removed from the program as an optimization step. Nonetheless, we add it to the rewritten program since our rewriting technique discussed in the proof does this.

From the above we conclude that E is redundant in the presence of I and A. Since we already know that arity is redundant, we obtain:

Theorem 4.7. *E is redundant in the presence of I.*

4.3 Packing

In this section we show that packing is redundant. The main task will be to eliminate packing from equations in nonrecursive programs. We will follow the following strategy to achieve this task:

(1) In Section 4.3.3 we show how to eliminate all variables that can hold values with packing. We will call such variables *impure*. The elimination is achieved by "solving" equations involving impure variables. This assumes equations of specific form, called *one-sided nonlinear* equations.
(2) Thereto, we will extend a known method for solving word equations that is guaranteed to terminate on one-sided nonlinear word equations. We begin by recalling this method in Section 4.3.1. In Section 4.3.2 we present the extension to path expressions.
(3) When all variables are pure, equations involving packing can only be satisfiable if the two sides have a similar "shape", called *packing structure*. We formalize this in Section 4.3.4.

The main result concerning packing is then proven in Section 4.3.5.

4.3.1 Solving equations. Consider an equation $e_1 = e_2$ and let X be the set of variables occurring in the equation. A valuation ν on X is called a *solution* if $\nu(e_1)$ and $\nu(e_2)$ are the same path.

Example 4.8. Consider the equation $\$x \cdot a = b \cdot \$y$ with $\$x$ and $\$y$ being distinct path variables and both a and b being atomic values. It is clear that one possible solution to this equation is the valuation ν with $\nu : \{\$x \mapsto b, \$y \mapsto a\}$ since $\nu(\$x \cdot a) = b \cdot a = \nu(b \cdot \$y)$. In general, we can see that for every possible path p, the valuation $\nu : \{\$x \mapsto b \cdot p, \$y \mapsto p \cdot a\}$ constitutes a solution to the equation.

As highlighted by the previous example, one can see that the set of solutions is typically infinite, so we would like a way to represent this set in a finite manner. Thereto one can use variable substitutions: partial functions that map variables to path expressions over X. Such a variable substitution ρ is called a *symbolic solution* to the equation if $\rho(e_1)$ and $\rho(e_2)$ are the same path expression. Every symbolic solution ρ represents a set of solutions

$$[\rho] := \{\nu \circ \rho \mid \nu \text{ a valuation on } X\}.$$

A set R of symbolic solutions is called *complete* if $\bigcup_{\rho \in R}[\rho]$ yields the complete set of solutions to the equation.

Example 4.9. Continuing on Example 4.8, one can verify that each of the following is a symbolic solution to the equation and together they form a complete set of symbolic solutions:

- $\rho_1 : \{\$x \mapsto b, \$y \mapsto a\}$; and
- $\rho_2 : \{\$x \mapsto b \cdot \$x, \$y \mapsto \$x \cdot a\}$.

The classical setting of *word equations* [3] can be seen as a special case of the situation just described. A word equation corresponds to the case where e_1 and e_2 contain no packing, and no atomic variables, i.e., all variables are path variables.

Plotkin's "pig-pug" procedure for associative unification [39] generates a complete set of symbolic solutions to any word equation. However, not every word equation admits a *finite* complete set of symbolic solutions; a simple example is our familiar equation $\$x \cdot a = a \cdot \$x$. Hence, in general, the procedure may not terminate.[5] Nevertheless, pig-pug is guaranteed to terminate on "one-sided nonlinear" equations [16]. These are word equations where all variables that occur more than once in the equation, only occur in one side of the equation.

We briefly review the pig-pug procedure. The procedure constructs a search tree whose nodes are labeled with word equations; the root is labeled with the original word equation. For each node we generate children according to a rewriting relation, $\Rightarrow$, on word equations. Intuitively, at any node, we look to the first symbol from both sides of the equation, and then consider all the matching possibilites between those two symbols. In the most general case of having variables in both side of the equation, it is possible that the two symbols (i.e., variables) have paths of the same length, or that one is longer than the other. For each such possibility, a child node is added to the search tree representing the equation of that possibility. Specifically, we have the following rewrite rules:

(1) Cancellation rule: $(x \cdot w_1 = x \cdot w_2) \Rightarrow (w_1 = w_2)$, for $x \in \mathbf{dom} \cup X$.
(2) Main rules: each one of the rules is associated with a substitution, ρ. Let $\mathbf{x}$ and $\mathbf{y}$ be distinct variables and let a be an atomic value.
   (a) $(\mathbf{x} \cdot w_1 = \mathbf{y} \cdot w_2) \Rightarrow (\mathbf{x} \cdot \rho(w_1) = \rho(w_2))$ with $\rho(\mathbf{x}) = \mathbf{y} \cdot \mathbf{x}$
   (b) $(\mathbf{x} \cdot w_1 = \mathbf{y} \cdot w_2) \Rightarrow (\rho(w_1) = \rho(w_2))$ with $\rho(\mathbf{x}) = \mathbf{y}$
   (c) $(\mathbf{x} \cdot w_1 = \mathbf{y} \cdot w_2) \Rightarrow (\rho(w_1) = \mathbf{y} \cdot \rho(w_2))$ with $\rho(\mathbf{y}) = \mathbf{x} \cdot \mathbf{y}$
   (d) $(\mathbf{x} \cdot w_1 = a \cdot w_2) \Rightarrow (\mathbf{x} \cdot \rho(w_1) = \rho(w_2))$ with $\rho(\mathbf{x}) = a \cdot \mathbf{x}$
   (e) $(\mathbf{x} \cdot w_1 = a \cdot w_2) \Rightarrow (\rho(w_1) = \rho(w_2))$ with $\rho(\mathbf{x}) = a$
   (f) $(a \cdot w_1 = \mathbf{y} \cdot w_2) \Rightarrow (\rho(w_1) = \mathbf{y} \cdot \rho(w_2))$ with $\rho(\mathbf{y}) = a \cdot \mathbf{y}$
   (g) $(a \cdot w_1 = \mathbf{y} \cdot w_2) \Rightarrow (\rho(w_1) = \rho(w_2))$ with $\rho(\mathbf{y}) = a$

When no rule is applicable to an equation, we have reached a leaf node in the search tree. There are three possible cases for such a leaf equation:

(1) $(\epsilon = \epsilon)$.
(2) $(a \cdot w_1 = b \cdot w_2)$, for atomic values $a \neq b$.
(3) $(\epsilon = w)$ or $(w = \epsilon)$, for nonempty w.

The first case is successful, while the other two are not. Each path from the root to a leaf node of the form $(\epsilon = \epsilon)$ yields a symbolic solution, formed by composing the substitutions given by the rewritings along the path. When starting from a one-side nonlinear equation, the tree is finite and we obtain a complete finite set of symbolic

[5]The reader may be interested to know that other means of finite representation (different from a finite set of substitutions) have been discovered, that work for arbitrary word equations [38].

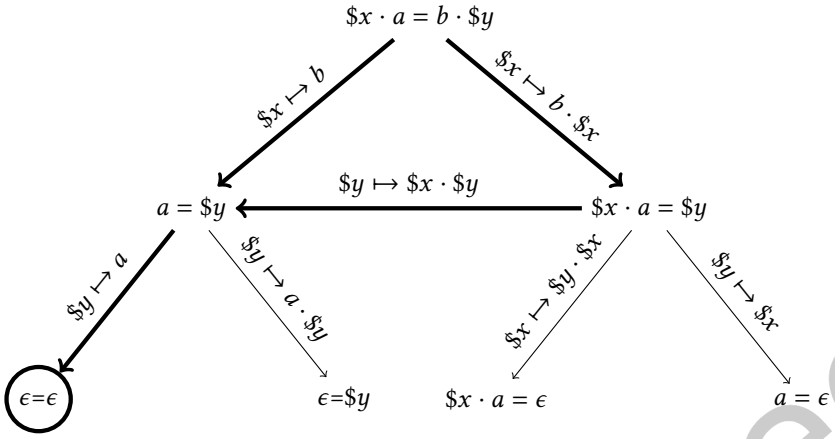


Fig. 3. Pig-pug procedure applied on equation $\$x \cdot a = b \cdot \$y$. Paths from root to leaf of bold edges indicate the successful branches.

solutions.[6] As an illustration, the DAG representation of the search tree of the equation from Example 4.8 is given in Figure 3. The two successful branches represent the two symbolic solutions given in Example 4.9.

4.3.2 Extension to path expressions. Our equations differ from word equations in that path expressions can involve packing as well as atomic variables. To this end, we extend the rewriting system as follows.

(h) Given an equation of the form $(@x \cdot w_1 = @y \cdot w_2)$, the only possibility is for $@x$ and $@y$ to be the same. Thus we add the rule $(@x \cdot w_1 = @y \cdot w_2) \Rightarrow (\rho(w_1) = \rho(w_2))$ with $\rho(@x) = @y$.

(i) An equation of the form $(@x \cdot w_1 = \$y \cdot w_2)$ is not very different from the case where we have a constant instead of $@x$. Thus, we add two rules similar to rules (f) and (g) where the first covers the case when the length of $\$y$ path is strictly larger than one while the second covers the case when the length of $\$y$ path is exactly one:
- $(@x \cdot w_1 = \$y \cdot w_2) \Rightarrow (\rho(w_1) = \$y \cdot \rho(w_2))$ with $\rho(\$y) = @x \cdot \$y$
- $(@x \cdot w_1 = \$y \cdot w_2) \Rightarrow (\rho(w_1) = \rho(w_2))$ with $\rho(\$y) = @x$

(j) Analogously, we add rules similar to rules (d) and (e):
- $(\$x \cdot w_1 = @y \cdot w_2) \Rightarrow (\$x \cdot \rho(w_1) = \rho(w_2))$ with $\rho(\$x) = @y \cdot \$x$
- $(\$x \cdot w_1 = @y \cdot w_2) \Rightarrow (\rho(w_1) = \rho(w_2))$ with $\rho(\$x) = @y$

(k) Given an equation of the form $(\langle w_1 \rangle \cdot w_2 = \langle w_3 \rangle \cdot w_4)$, we work inductively and solve the equation $(w_1 = w_3)$ first. Assuming we can find a finite complete set R of symbolic solutions for this equation, we then add the rules $(\langle w_1 \rangle \cdot w_2 = \langle w_3 \rangle \cdot w_4) \Rightarrow (\rho(w_2) = \rho(w_4))$ for $\rho \in R$.

(l) An equation of the form $(\langle w_1 \rangle \cdot w_2 = \$y \cdot w_3)$ is again not very different from the case where we have a constant instead of $\langle w_1 \rangle$. Thus, we add two rules similar to rules (f) and (g):
- $(\langle w_1 \rangle \cdot w_2 = \$y \cdot w_3) \Rightarrow (\rho(w_2) = \$y \cdot \rho(w_3))$ with $\rho(\$y) = \langle w_1 \rangle \cdot \$y$

[6] It is standard in the literature on word equations to consider only solutions that map variables to nonempty words. The above procedure is only complete under that assumption. However, allowing the empty word can be easily accommodated. For any equation eq on a set of variables X, and any subset Y of X, let eq_Y be the equation obtained from eq by replacing the variables in Y by the empty word. Let R_Y be a complete set of symbolic solutions for eq_Y where we extend each substitution to X by mapping every variable from Y to the empty word. Then the union of the R_Y is a complete set of symbolic solutions for eq, allowing the empty word. If eq is one-sided nonlinear, then eq_Y is too. This remark equally applies to the extension to path expressions presented in Section 4.3.2.

- $(\langle w_1\rangle \cdot w_2 = \$y \cdot w_3) \Rightarrow (\rho(w_2) = \rho(w_3))$ with $\rho(\$y) = \langle w_1\rangle$

(m) Analogously, we again add rules similar to rules (d) and (e):
- $(\$x \cdot w_1 = \langle w_2\rangle \cdot w_3) \Rightarrow (\$x \cdot \rho(w_1) = \rho(w_3))$ with $\rho(\$x) = \langle w_2\rangle \cdot \$x$
- $(\$x \cdot w_1 = \langle w_2\rangle \cdot w_3) \Rightarrow (\rho(w_1) = \rho(w_3))$ with $\rho(\$x) = \langle w_2\rangle$

Furthermore, we now have extra four non-successful cases for leaf equations, namely all equations of the form $(@x \cdot w_1 = \langle w_2\rangle \cdot w_3)$, $(\langle w_2\rangle \cdot w_3 = @y \cdot w_1)$, $(a \cdot w_1 = \langle w_2\rangle \cdot w_3)$, or $(\langle w_2\rangle \cdot w_3 = b \cdot w_1)$.

It remains to argue that on any one-sided nonlinear equation, our extended rewriting system terminates and yields a finite complete set of symbolic solutions. In general, any side of an equation (that is not empty) may begin with an atomic value, an atomic variable, a packed value, or a path variable. The completeness is clear since all possible ways that could match the first two symbols of any of the four aforementioned cases are covered by our rules.

As for the termination, the argument is less clear but it easily extends known arguments [16]. Thereto, suppose that we have a one-sided nonlinear equation where the left side of the equation is linear. Notice that, in this case, the rewriting rule for (k) is equivalent to the following simpler rule: $(\langle w_1\rangle \cdot w_2 = \langle w_3\rangle \cdot w_4) \Rightarrow (w_2 = \rho(w_4))$ for $\rho \in R$ where R is the symbolic solution set of $(w_1 = w_3)$. Indeed, the equivalence follows from the fact that none of the variables that appear in w_2 appear anywhere else in the equation and hence $\rho(w_2) = w_2$. Accordingly, we observe that none of the rewriting rules can increase the number of variables or values (and hence the symbols) in the left side neither can it increase the nesting of the packing in the left side. Moreover, in case the rewriting rule does not strictly decrease the number of the symbols in the left side, the right side then is guaranteed to decrease in that case. In a similar way, we observe that although some of the rules can make the right side of the equation larger, all of those rules make the left side strictly smaller. Hence, the number of times such rules can be executed is bounded by the number of variables and values appearing in the left side.

Example 4.10. Figure 4 shows a DAG representation of the search tree for the equation $\$x \cdot \langle @y \cdot \$z\rangle \cdot @w = \$u \cdot \$v \cdot \$u$. There are four successful branches, so the following substitutions comprise a complete set of symbolic solutions:

$$\{\$x \mapsto @w,\ \$u \mapsto @w,\ \$v \mapsto \langle @y \cdot \$z\rangle\}$$
$$\{\$x \mapsto @w \cdot \$x,\ \$v \mapsto \$x \cdot \langle @y \cdot \$z\rangle,\ \$u \mapsto @w\}$$
$$\{\$x \mapsto \langle @y \cdot \$z\rangle \cdot @w \cdot \$v,\ \$u \mapsto \langle @y \cdot \$z\rangle \cdot @w\}$$
$$\{\$x \mapsto \$x \cdot \langle @y \cdot \$z\rangle \cdot @w \cdot \$v \cdot \$x,\ \$u \mapsto \$x \cdot \langle @y \cdot \$z\rangle \cdot @w\}$$

4.3.3 Pure variables and pure equations. We introduce a syntactic "purity check" on variables, that guarantees that they can only take values that do not contain packed values. Since later we will work stratum per stratum, it is sufficient in what follows to focus on semipositive, nonrecursive programs with only one IDB relation name.

Consider a rule in such a program. When a variable appears in some positive EDB predicate, we call the variable a *source variable* of the rule. Now we inductively define a variable in the rule to be *pure* if

(1) it is a source variable (since we focus on flat input instances); or
(2) it appears in one side of a positive equation, such that
   - all the variables in the other side of the equation are pure, and
   - the other side of the equation has no packing.

By leveraging associative unification, we are going to show that we can always eliminate impure variables. The method is based on a division of the positive equations of a rule into three categories:

Pure equations involve only pure variables.

Half-pure equations have all variables in one side pure, and at least one of the variables in the other side is impure.

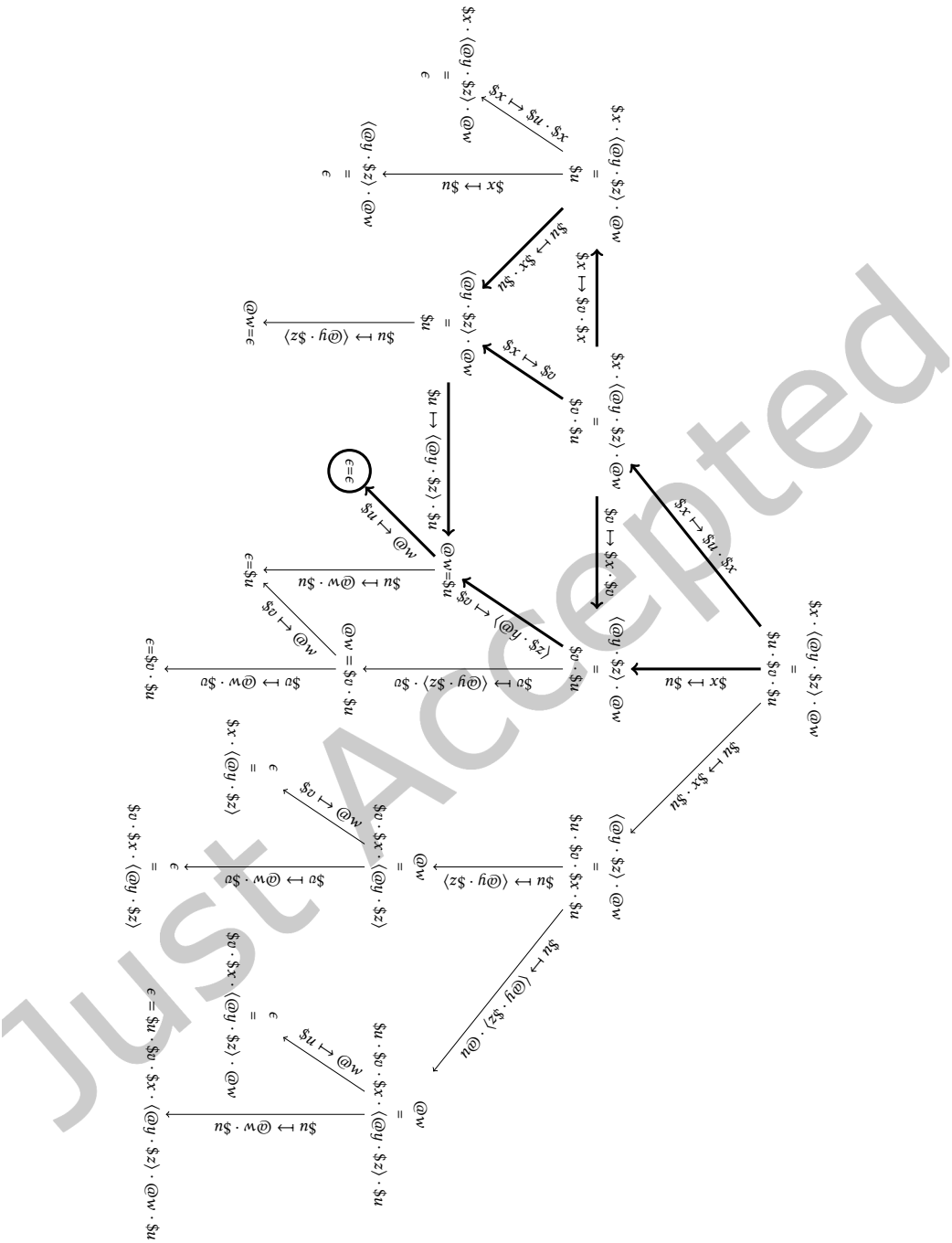


Fig. 4. Associative unification on an equation on path expressions. Bold edges indicate the successful branches.

Fully impure equations have impure variables in both sides.

Example 4.11. The three equations in the rule

$$S(\$x) \leftarrow R(\$x, \$y) \wedge \langle \$x \rangle = \langle \$y \rangle \wedge a \cdot \$x = \$z \wedge \$y = \langle \$u \rangle$$

are pure. The two equations in the rule

$$S(\$x) \leftarrow R(\$x, \$y) \wedge \langle \$y \rangle = \$z \wedge \langle \$x \rangle = \langle \$z \rangle$$

are half-pure. The equation $\langle \$t \rangle = \langle \$z \rangle$ in the rule

$$S(\$x) \leftarrow R(\$x, \$y) \wedge \langle \$t \rangle = \langle \$z \rangle \wedge \$z = \langle \$y \rangle \wedge \$t = \langle \$x \rangle$$

is fully impure.

It is instructive to compare the notion of pure variable with that of limited variable, used to define the notion of safe rule. Indeed, the set of limited variables can be equivalently defined as follows, where we only change the base case of the induction to immediately include all pure variables:

- Every pure variable is limited; and
- If all the variables occurring in one side of the sides of an equation in the rule are limited, then all the variables occurring in the other side are also limited.

Therefore, if there is at least one impure variable in a safe rule, then there must be at least one half-pure equation in the rule. In other words, it is not possible for a rule to have fully impure equations without having half-pure ones.

Lemma 4.12. *Let r be a rule in a semi-positive, nonrecursive program $\mathbf{P}$ with only one IDB relation name. Then there exists a finite set of rules, equivalent to r on flat instances, in which all positive equations are pure.*

Proof. By induction on the number of half-pure equations. Let $r : H \leftarrow B \wedge e_1 = e_2$, where $e_1 = e_2$ is half-pure with e_1 the pure side and e_2 the impure side. Let $u_1, \ldots, u_n$ be the list of all *occurrences* of variables in e_1. Let $v_1, \ldots, v_n$ be n fresh variables, and let e_1' be e_1 with each u_i replaced by v_i. Now replace $e_1 = e_2$ by the following conjunction of $n + 1$ equations:

$$u_1 = v_1 \wedge \ldots \wedge u_n = v_n \wedge e_1' = e_2$$

Here, abusing notation, we use the same notation u_i for the variable that occurs at u_i.

Denote the result of this replacement by r'. The equation $e_1' = e_2$ is one-sided nonlinear; by Section 4.3.2, there exists a finite complete set R of solutions. If we let r'' be r' without $e_1' = e_2$, then clearly r is equivalent to the set of rules $\{\rho(r'') \mid \rho \in R\}$. However, some of these rules may not have strictly less half-pure equations than r, which is necessary for the induction to work.

We can solve this problem as follows. Call $\rho \in R$ *valid* if it maps variables that are pure in r'' to expressions without packing. Since all u_i and v_i are pure in r'', the equations $\rho(u_i) = \rho(v_i)$ in $\rho(r'')$ are all pure, so $\rho(r'')$ does have strictly less half-pure equations than r.

Fortunately, we can restrict attention to the valid $\rho \in R$, so the induction goes through. Indeed, following the definition of pure variable, one can readily verify that for nonvalid ρ, the rule $\rho(r'')$ is unsatisfiable on flat instances. □

4.3.4 Packing structures. By Lemma 4.12, all positive equations can be taken to be pure. We now reduce this further so that all positive equations are free of packing. Thereto we introduce the *packing structure* of a path expression e, denoted by $\delta(e)$, and defined as follows:

- $\delta(\epsilon) = *$.
- $\delta(a) = *$, with a a variable or an atomic value.

- $\delta(\langle e \rangle) = * \cdot \langle \delta(e) \rangle \cdot *$.
- $\delta(e_1 \cdot e_2)$ equals $\delta(e_1) \cdot \delta(e_2)$, in which we replace any consecutive sequence of stars by a single star.

Assume $\delta(e)$ has n stars. Then e can be constructed from $\delta(e)$ by replacing each star by a unique (possibly empty) subexpression of e. We call these subexpressions the components of e. Crucially, they do not use packing.

If e does not use packing, $\delta(e)$ is simply $*$. If e begins or ends with packing, or if some packing in e begins or ends with another packing, then some components will be empty.

Example 4.13. Let $e = @a \cdot \langle \langle \$x \cdot \$y \rangle \cdot \$z \rangle \cdot \langle \epsilon \rangle$. Then $\delta(e) = * \cdot \langle * \cdot \langle * \rangle \cdot * \rangle \cdot * \cdot \langle * \rangle \cdot *$. The seven components of e are $@a$, ϵ, $\$x \cdot \$y$, $\$z$, ϵ, ϵ, and ϵ.

A pure equation $e_1 = e_2$ can only be satisfiable on flat instances if e_1 and e_2 have the same packing structure. Suppose there are n stars in this packing structure. Then, the equation can be replaced by the conjunction of n equations, where we equate the corresponding components of e_1 and e_2. These equations are still pure, and free of packing.

Moreover, when all positive equations are pure, then all variables in the rule are pure, since the rule is safe. Now a negated equation $e_1 \neq e_2$ over pure variables is equivalent to the disjunction of the nonequalities between the corresponding components of e_1 and e_2. Then the rule can be replaced by a set of rules, one for each disjunct, and the component nonequalities are free of packing. We can repeat this for all negated equations.

We have arrived at the following:

Lemma 4.14. *Let r be a rule in a semi-positive, nonrecursive program* **P** *with only one IDB relation name. Then there exists a finite set of rules, equivalent to r on flat instances, in which all variables are pure, and all equations (positive or negated) are free of packing.*

4.3.5 Redundancy of packing. We are now ready for the following result. The proof further leverages packing structures.

Lemma 4.15. *Packing is redundant in the absence of recursion.*

Proof. Consider a query computed by a nonrecursive program **P**. We must show that **P** can be equivalently rewritten without packing. If **P** has only one IDB predicate, Lemma 4.14 gives us what we want. Indeed, by the Lemma, we may assume that equations are already free of packing. Now since the input is a flat instance, any positive (negated) EDB predicate that contains packing may be taken to be always false (true). Also, the result of the query is a flat instance, so IDB predicates containing packing are false as well. We thus obtain a program free of packing as desired.

When **P** uses intermediate predicates, the elimination of packing from IDB predicates requires more work. Since **P** is nonrecursive, we may assume that every stratum involves only one IDB relation name. Since arity is redundant, we may assume that **P** does not use arity, but feel free to use arity in the rewriting of **P**.

Let us consider the first stratum. For every rule, we proceed as follows. Let $R(e)$ be the head of the rule. Let m be the number of stars in $\delta(e)$ and let $e_1, \ldots, e_m$ be the components of e. Replace the head with $R_{\delta(e)}(e_1, \ldots, e_m)$ where $R_{\delta(e)}$ is a fresh relation name.

After this step, the rules in the first stratum no longer contain packing in the head. Of course, R-predicates in rules in later strata must now be updated to call the new relation names. So, assume $R(e)$ appears in the body of some later rule r. For each of the packing structures ps introduced for R, we make a copy of r in which we replace $R(e)$ by the conjunction $R_{ps}(\$e_1, \ldots, \$e_m) \wedge e = e'$, where

- m is the number of stars in ps;
- $\$e_1, \ldots, \$e_m$ are fresh path variables; and
- e' is obtained from the packing structure ps by replacing the ith star by $\$e_i$, for $i = 1, \ldots, m$.

This rewriting introduces equations in later strata, which is necessary because these later strata have not yet been purified per Lemma 4.14.

We do the above for every stratum. So, stratum by stratum, we first remove packing from equations, leaving only pure variables in rules; we replace head predicates; and rewrite calls to these head predicates in later rules.

After this transformation, packing still appears in negated IDB predicates, which have been untouched so far. Fortunately, all rules have pure variables at this point. Thus, a literal $\neg R(e)$, where $\delta(e)$ matches one of the packing structures of R, say ps, with m stars, can now be replaced by $\neg R_{ps}(e_1, \ldots, e_m)$, where e_i is the ith component of e. If $\delta(e)$ does not match any of the packing structures introduced for R, the negative literal is true on flat instances and can be omitted.

Observing that packing in EDB predicates can be handled as in the semipositive case, we are done. □

Example 4.16. Rewriting the program from Example 2.3 without packing yields a program with 28 rules:

```
T($u, $s, $v) ← R($u·$s·$v), S($s).
A ← T($x1,$x2,$x3), T($y1,$y2,$y3), T($z1,$z2,$z3), $xi≠$yi, $xj≠$zj, $yk≠$zk.
% for i=1,2,3, j=1,2,3, k=1,2,3
```

To get from Lemma 4.15 to the following theorem, it remains to show that packing is redundant in the presence of recursion. Building on the flat–flat theorem for J-Logic [23, 24] we can close that gap and we obtain:

Theorem 4.17. *Packing is redundant.*

Proof. It remains to show that P is redundant in the presence of R. Earlier work on J-Logic (flat-flat theorem [23, 24]) is easily adapted to Sequence Datalog and shows that P is redundant in the presence of R and N. The general idea of the rewriting used in that proof is as follows (where, for completeness, we briefly explain the idea with a slight modification):

(1) We add a new stratum at the beginning of the program, where we preprocess the input relations as follows: every path $k_1 \cdot k_2 \cdot \cdots \cdot k_n$ is replaced by its doubled version $k_1 \cdot k_1 \cdot k_2 \cdot k_2 \cdot \cdots \cdot k_n \cdot k_n$.
(2) We modify the program so that it works with doubled EDB and IDB relations. Packing is simulated using a technique of simulated delimiters, which relies on the doubled encoding. Intuitively, since every (flat) path p is doubled as p', using $a \cdot b \cdot p' \cdot b \cdot a$ suffices to represent the path $\langle p \rangle$ by considering $a \cdot b$ and $b \cdot a$ as the delimiters of the packing operator. Precisely, every rule in the original program is rewritten where every atomic variable and every constant is doubled. That is, $@x$ and a will be considered in the new program as $@x \cdot @x$ and $a \cdot a$. Moreover, every occurrence of packing is replaced by the aforementioned delimiters. Every path variable $\$x$ in the original rule is kept as it is in the rewritten rule, but we need to make sure that it encodes a valid (doubled and delimited) subpath. Thereto, suppose that $\$x$ appears in a predicate T, then an extra predicate is added to the rule of the form $Path_T(\$x)$ where $Path_T$ is a relation that is defined recursively to obtain all subpaths of T that are of valid form.
(3) In the last step, we undouble the doubled output.

We remark that steps 1 and 3 as published introduce negation even if the original program does not use negation. We next show that this can be avoided. Instead, we introduce arity, which is harmless as arity is redundant.

We double an EDB relation R into R' as follows:

```
T(ϵ, $x) ← R($x).
T($x·@y·@y, $z) ← T($x, @y·$z).
R'($x) ← T($x, ϵ).
```

We undouble a doubled output relation S' into S as follows:

```
T($x, ϵ) ← S'($x).
T($x, @y·$z) ← T($x·@y·@y, $z).
S($x) ← T(ϵ, $x).
```

□

Example 4.18. Consider the third program of Example 3.1 which is repeated below for convenience.

```
T(<$x>·<$x>) ← R($x).
T(<$x>·<$y>) ← T(<$x>·<$y·a>).
S($x) ← T(<$x>·<ϵ>).
```

Equivalently, the program can be rewritten without packing (over the doubled EDB relations and the doubled output relation) as follows:

```
T(a·b·$x·b·a·a·b·$x·b·a) ← R'($x).
T(a·b·$x·b·a·a·b·$y·b·a) ← T(a·b·$x·b·a·a·b·$y·a·a·b·a), Path_T($x), Path_T($y).
Path_T(ϵ) ← .
Path_T(a·b·$x·b·a) ← T($u·a·b·$x·b·a·$v), Path_T($x).
Path_T(@y·@y·$x) ← T($u·@y·@y·$x·$v), Path_T($x).
Path_T($x·@y·@y) ← T($u·$x·@y·@y·$v), Path_T($x).
S'($x) ← T(a·b·$x·b·a·a·b·b·a), Path_T($x).
```

4.4 Intermediate predicates

The following result is straightforward and uses well-known techniques showing that compositions of nonrecursive rules can be unfolded into a single nonrecursive rule [4, Section 4.3]: intermediate predicates can be eliminated by folding in the bodies of the intermediate rules, using equations to unify calling predicates with intermediate head predicates. This idea is simply illustrated by an example.

Example 4.19. Recall the last program of Example 3.1.

```
T(a·$x·$x) ← R($x).
S($x) ← R($x), T($x·a·$x).
```

This program can be equivalently rewritten without intermediate predicates as follows:

```
S($x) ← R($x), $x·a·$x=a·$y·$y, R($y).
```

We thus obtain:

THEOREM 4.20. *I is redundant in the presence of E and the absence of N and R.*

5 Inexpressibility results

In this section we show various inexpressibility results that lead to absolute or relative primitive results for various features.

5.1 Recursion

To see that recursion is primitive also in the context of Sequence Datalog, we can make the following observation.

Lemma 5.1. *Let Q be a query that can be computed by a nonrecursive program. Then for any input instance I, the lengths of paths in $Q(I)$ are bounded by a linear function of the maximal length of a path in I.*

Proof. Let $\mathbf{P}$ be a nonrecursive program computing a query Q. Let $\mathbf{P}'$ be $\mathbf{P}$ with all negated literals removed. The Q' query computed by $\mathbf{P}'$ contains Q, so if we can prove the claim for Q', it also holds for Q.

By Theorem 4.20, we know that Q' is computable by a program $\mathbf{P}''$ that does not use intermediate predicates. Let n be the number of rules, and for $i = 1, \ldots, n$, let $S(e_i)$ be the head of the ith rule; a_i the number of path variables in e_i; and b_i the number of atomic values and variables in e_i. Then the length of sequences returned by the ith rule is at most $a_i x + b_i$, with x the maximal length of a sequence in the input. The desired linear function can now be taken to be $ax + b$, where $a = \max\{a_i \mid 1 \leq i \leq n\}$ and $b = \max\{b_i \mid 1 \leq i \leq n\}$. □

We immediately get:

Proposition 5.2. *Let a be a fixed atomic value and let Q be any query from $\{R\}$ to S satisfying the property that for every instance I and every natural number n with $R(a^n) \in I$, the string a^{n^2} is a substring of a path in $Q(I)$. Then Q is not expressible without recursion.*

We readily obtain:

Theorem 5.3. *Recursion is primitive.*

Proof. First, we show that R is primitive in the presence of I. Consider the following recursive program $\mathbf{P}$, computing the query Q returning all paths a^{n^2} where n is a natural number such that $R(a^n)$ is in the input:

```
T(ϵ, $x, $x) ← R($x).
T($y·$x, $x, $z) ← T($y, $x, a·$z).
S($y) ← T($y, $x, ϵ).
```

By Proposition 5.2, query Q is not expressible without recursion.

The above program uses intermediate predicates. In the absence of this feature, consider just the program $\mathbf{P}'$ consisting of the first two rules. Strictly, this program does not compute a query, as T is ternary. However, we can turn $\mathbf{P}'$ into a program $\mathbf{P}''$ using the arity simulation technique of Lemma 4.1. Program $\mathbf{P}''$ computes a well-defined query Q'' from $\{R\}$ to T. Although Q'' is not a natural query, Proposition 5.2 applies to it, so it is not expressible without recursion. □

5.1.1 Boolean queries. The above queries showing primitivity of recursion are unary. What about Boolean queries? It turns out that for Boolean queries, in the presence of intermediate predicates, recursion is still primitive. In the absence of intermediate predicates, however, recursion is redundant for Boolean queries, for trivial reasons.

Let us go in a bit more detail. Let R be a binary relation viewed as a directed graph. Let $Q_{\mathrm{a}\to\mathrm{b}?}$ be the Boolean query from $\{R\}$ to S that checks whether b is reachable from a. It is well-known that $Q_{\mathrm{a}\to\mathrm{b}?}$, as a classical relational query, is not computable in classical Datalog without recursion. We can view $Q_{\mathrm{a}\to\mathrm{b}?}$ as a query on sequence databases by encoding edges (a, b) by paths $a \cdot b$ of length two. Under this encoding, the query is clearly computable by a Sequence Datalog program in the fragment {I, R}:

```
T(@x·@y) ← R(@x·@y).
T(@x·@z) ← T(@x·@y),R(@y·@z).
S ← T(a·b).
```

We can now show that $Q_{\mathrm{a}\to\mathrm{b}?}$ is not computable without recursion in Sequence Datalog by showing that, on input instances containing only sequences of length two, any nonrecursive Sequence Datalog program can be simulated by a classical nonrecursive Datalog program. This simulation is similar to the one shown in Lemma 5.4 appearing later. The only added complication is that, due to intermediate predicates, sequences of lengths longer than two can appear. However, since there is no recursion, these lengths are bounded by a constant depending only on the program.

In the absence of the I-feature, we note that any Boolean query, computed by a recursive program without intermediate predicates, is already computed by the nonrecursive rules only. Indeed, if the result of the query is false, then none of the rules is fired. If, on the other hand, the result of the query is true, then at least one rule is fired; however, no recursive rule can be fired before at least one nonrecursive rule is fired.

5.2 Intermediate predicates

It is well known that in classical Datalog, without intermediate predicates, we can not express queries that require universal quantifiers [12]. We can transfer this result to Sequence Datalog by a simulation technique.

Let Γ be a monadic schema and let I be an instance of Γ. We say that I is "two-bounded" if only paths of lengths one or two occur in I. We can encode two-bounded instances by classical instances as follows. Let Γ^c ('c' for classical) be the schema that has two relation names R^1 and R^2 for each $R \in \Gamma$. For I two-bounded as above, we define the classical instance I^c of Γ^c as follows:

- $I^c(R^1) = \{a \in \mathbf{dom} \mid a \in I(R)\}$;
- $I^c(R^2) = \{(a, b) \mid a \cdot b \in I(R)\}$.

Lemma 5.4. *Let* $\mathbf{P}$ *be a program in the fragment* $\{E, N, R\}$*, with IDB relation name* S*, such that the result of* $\mathbf{P}$ *on a two-bounded instance is still two-bounded. Then there exists a semipositive classical Datalog program* $\mathbf{P^c}$ *using only the IDB relation names* S^1 *and* S^2*, such that for every two-bounded instance* I *of* Γ*, we have* $\mathbf{P^c}(I^c) = (\mathbf{P}(I))^{\mathbf{c}}$.

Proof. Our goal is to eliminate path variables as well as concatenations in path expressions. We start with path variables. In any rule containing a head predicate or positive predicate of the form $S(e_1 \cdot \$x \cdot e_2)$ or $R(e_1 \cdot \$x \cdot e_2)$, we can replace $\$x$ either by ϵ, $@x$, or $@x_1 \cdot @x_2$ (splitting the rule in three versions).

Path variables may still occur in equations. By safety, they must appear in positive equations, and inductively we may assume that any remaining path variable $\$x$ occurs in a positive equation $e_1 = e_2$ where e_1 contains no path variables. This equation is then of the form $a_1 \cdots a_n = b_1 \cdots b_m \cdot \$x \cdot e$, where the as and bs are atomic variables or values.

- If $m = n$, replace $\$x$ by the empty path.
- If $m > n$, the equation is unsatisfiable and the rule can be removed.
- If $m < n$, replace $\$x$ by $a_{m+1} \cdots a_i$, for $m < i \leq n$ (splitting the rule in $n - m + 1$ versions).

After these steps, all equations (positive or negated) are of the form $a_1 \cdots a_n = b_1 \cdots b_m$, where the as and bs are atomic variables or values. Such equations can be easily eliminated. Moreover, any predicates, possibly negated, that are of the form $R(e)$ with e empty or strictly longer than two, can be eliminated as well.

We finally replace every remaining predicate (head or body) of the form $R(a)$ by $R^1(a)$ and every predicate of the form $R(a_1 \cdot a_2)$ by $R^2(a_1, a_2)$, and we are done. □

As a consequence, the query computed by the following program, belonging to the fragment {I, N}, cannot be expressed without intermediate predicates:

```
W(@x) ← R(@x·@y), ¬B(@y).
S(@x) ← R(@x·@y), ¬W(@x).
```

Indeed, the classical counterpart of this query is the query asking, on any directed graph where some nodes are "black", for all nodes with only edges to black nodes. That query is well-known not to be expressible in classical semipositive Datalog [12] (recalled in Section 2.2).

We thus obtain:

Theorem 5.5. *I is primitive in the presence of N.*

We also have the following primitivity result in the presence of recursion. The proof merely combines some observations we have already made.

Theorem 5.6. *I is primitive in the presence of R.*

Proof. Recall the squaring query Q from the proof of Theorem 5.3, which is expressible in the fragment {I, R}. Suppose, for the sake of contradiction, that Q can be computed by a program without intermediate predicates. Consider the behavior of this program on the family of singleton instances $I_n = \{R(a^n)\}$, for all natural numbers n. Since $Q(I_n)$ is nonempty, at least one of the rules must fire, which is only possible if at least one of the nonrecursive rules fires. Since there are no intermediate predicates, however, this firing of the nonrecursive rule already produces the (unique) correct output $S(a^{n^2})$. This contradicts Lemma 5.1. Hence, the nonrecursive rule outputs a wrong result, and our supposed program is wrong. □

5.3 Equations

The two theorems in the previous subsection provide counterparts to Theorem 4.20. The following theorem confirms that the presence of equations is necessary for Theorem 4.20, and implies that the fragments {I} and {E} are actually equivalent.

Theorem 5.7. *E is primitive in the absence of I.*

This result follows immediately from the following lemma.

Lemma 5.8. *Let a be an atomic value. The Boolean query that checks if the input relation R contains a path consisting exclusively of a's, cannot be computed by a program that lacks features I and E.*

Proof. By the redundancy of packing and arity, we may ignore these features. Also, in Section 5.1.1, we already noted that in the absence of intermediate predicates, recursion does not help in expressing Boolean queries. Hence, it suffices to show that the query cannot be computed by a program in the fragment {N}. For the sake of contradiction, suppose such a program exists.

Take any rule from the program, and consider the instance J obtained from the positive predicates in the body by "freezing" all variables, i.e., viewing them as atomic values distinct from the atomic values already occurring in the rule. Unless the rule is unsatisfiable (in which case we may ignore it), it will fire on J. So the query is true on J and the body must contain a positive predicate of the form $R(a^\ell)$.

Now consider the instance $I = \{R(a^n)\}$ where n is strictly larger than all values ℓ as above found in the rules. Then no rule can fire on I, but the query is true on I, so we have the desired contradiction. □

Indeed, that query is readily expressed using an equation, as we well know.

6 Putting it all together

The results from the previous two sections allow us to characterize the subsumption relation among fragments (defined in Section 3) and prove Theorem 3.2 (restated below for convenience) as follows.

Theorem 3.2 (Main Theorem). *For any fragments F_1 and F_2, we have $F_1 \leq F_2$ if and only if the following five conditions are satisfied:*

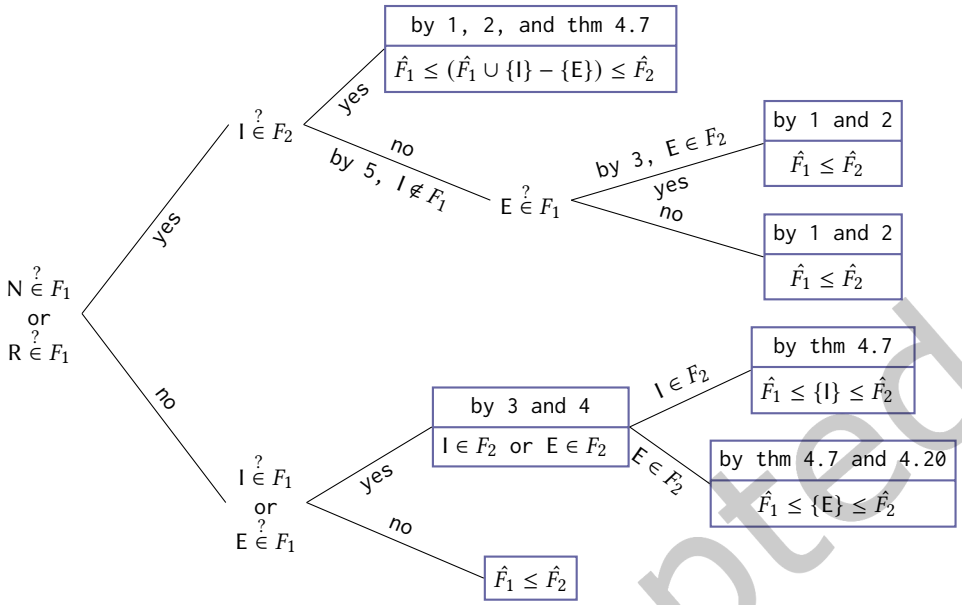


Fig. 5. If-direction of Theorem 3.2.

(1) $N \in F_1 \Rightarrow N \in F_2$*;*
(2) $R \in F_1 \Rightarrow R \in F_2$*;*
(3) $E \in F_1 \Rightarrow (E \in F_2 \vee I \in F_2)$*;*
(4) $(I \in F_1 \wedge R \notin F_1 \wedge N \notin F_1) \Rightarrow (I \in F_2 \vee E \in F_2)$*;*
(5) $(I \in F_1 \wedge (R \in F_1 \vee N \in F_1)) \Rightarrow I \in F_2$.

Proof. For the only-if direction, we verify the five conditions, assuming $F_1 \leq F_2$.

(1) Immediate from the primitivity of negation. We have not stated this primitivity as a theorem because it is so clear (any fragment without negation can express only monotone queries; with negation we can express set difference which is not monotone).
(2) Immediate from primitivity of recursion.
(3) Immediate from Theorem 5.7.
(4) Assume $\mathsf{I} \in F_1 \wedge \mathsf{R} \notin F_1 \wedge \mathsf{N} \notin F_1 \wedge \mathsf{E} \notin F_2 \wedge \mathsf{I} \notin F_2$. By Theorem 4.7, we have $\{\mathsf{E}\} \leq F_1$. Now Theorem 5.7 leads to a contradiction with $F_1 \leq F_2$.
(5) Immediate from Theorems 5.5 and 5.6.

For the if-direction, since arity and packing are redundant, $F_1 \leq F_2$ if and only if $\hat{F}_1 \leq \hat{F}_2$, where $\hat{F} = F - \{\mathsf{A}, \mathsf{P}\}$. Now Figure 5 infers $\hat{F}_1 \leq \hat{F}_2$ from the five conditions and the redundancy results. □

7 Sequence relational algebra

Given the importance of algebraic query plans for database query execution, we show here how to extend the classical relational algebra to obtain a language equivalent to recursion-free Sequence Datalog programs. We note that a similar language, while calculus-based rather than algebra-based, is the language StriQuel proposed by Grahne and Waller [21].

The (unnamed) relational algebra, with operators projection; equality selection; union; difference; and cartesian product, is well known [4, 46]. To extend this algebra to our data model (Section 2.1), we generalize the selection and projection operators and add three extraction operators. In what follows, let R be an n-ary relation on Π, that is, a finite set of tuples $\mathbf{t}$ such that each tuple is viewed as the valuation that maps $\$i$ to $\mathbf{t}_i$ for $i = 1, \ldots, n$ where $\mathbf{t}_i$ is the path at the ith position.

Selection: The classical equality selection $\sigma_{\$i=\$j}(R)$, with $i, j \in \{1, \ldots, n\}$, returns $\{\mathbf{t} \in R \mid \mathbf{t}_i = \mathbf{t}_j\}$. We now allow path expressions α and β over the variables $\$1, \ldots, \$n$ and have the selection operator

$$\sigma_{\alpha=\beta}(R) := \{\mathbf{t} \in R \mid \mathbf{t}(\alpha) = \mathbf{t}(\beta)\}.$$

Projection: For path expressions $\alpha_1, \ldots, \alpha_p$ over variables $\$1, \ldots, \$n$ as above, we define

$$\pi_{\alpha_1,\ldots,\alpha_p}(R) := \{(\mathbf{t}(\alpha_1), \ldots, \mathbf{t}(\alpha_p)) \mid \mathbf{t} \in R\}.$$

Unpacking: For $i \in \{1, \ldots, n\}$, the operator $\mathsf{UNPACK}_i(R)$ returns

$$\{(\mathbf{t}_1, \ldots, \mathbf{t}_{i-1}, s, \mathbf{t}_{i+1}, \ldots, \mathbf{t}_n) \mid (\mathbf{t}_1, \ldots, \mathbf{t}_{i-1}, \langle s \rangle, \mathbf{t}_{i+1}, \ldots, \mathbf{t}_n) \in R\}.$$

Substrings: $\mathsf{SUB}_i(R)$ equals

$$\{(\mathbf{t}_1, \ldots, \mathbf{t}_n, s) \mid (\mathbf{t}_1, \ldots, \mathbf{t}_n) \in R \text{ and } s \text{ is a substring of } \mathbf{t}_i\}.$$

Atoms: $\mathsf{ATOM}_i(R)$ equals

$$\{(\mathbf{t}_1, \ldots, \mathbf{t}_n, a) \mid (\mathbf{t}_1, \ldots, \mathbf{t}_n) \in R \text{ and } a \text{ is an atomic value from } \mathbf{t}_i\}.$$

We could also have defined a more powerful unpacking operator, which extracts components from paths using path expressions, similar to the use of path expressions in Sequence Datalog. Such an operator is useful in practice but can for theoretical purposes be simulated using the given operators, as we will show. First, we give an example of a sequence relational algebra expression that corresponds to the all a's query:

$$\sigma_{\$1 \cdot a = a \cdot \$1}(R).$$

"Sequence relational algebra" expressions over a schema Γ, built up using the above operators from the relation names of Γ and constant relations, are defined as usual. We have, as expected, the following theorem. Note that this result applies for arbitrary instances, not only for flat inputs and flat outputs.

Theorem 7.1. *For every program $\mathbf{P}$ without recursion and every IDB relation name T, there exists a sequence relational algebra expression E such that for every instance I, we have $\mathbf{P}(I)(T) = E(I)$. The converse statement holds as well.*

That sequence relational algebra can be translated to Sequence Datalog is clear. Nonetheless, for completeness the translation is given in Proposition 7.2.

Proposition 7.2. *Let E be a sequence relational algebra expression over a schema Γ. There exists a Sequence Datalog program $\mathbf{P}_E$ and IDB relation name T_E such that for every instance I over Γ, we have $E(I) = \mathbf{P}_E(I)(T_E)$.*

Proof. We establish the proof by structural induction on the sequence relational algebra expression E. As in classical relational algebra, every expression is associated to a particular arity.

The base cases are:

- If $E := \{\mathbf{t}\}$ with $\mathbf{t}$ being an n-ary tuple, then define T_E by the rule

  $$T_E(\mathbf{t}_1, \ldots, \mathbf{t}_n) \leftarrow$$

  and take $\mathbf{P}_E$ to be the program with the above rule.

- If $E := R$ for some n-ary relation R, then define T_E by the rule
$$T_E(\$v_1, \ldots, \$v_n) \leftarrow R(\$v_1, \ldots, \$v_n)$$
and take $\mathbf{P}_E$ to be the program with the above rule.

As for the induction step, assume that E_1 and E_2 are n-ary expressions and that E_3 is an m-ary expression such that we have equivalent rules defining T_{E_1}, T_{E_2}, and T_{E_3} and programs $\mathbf{P}_{E_1}$, $\mathbf{P}_{E_2}$, and $\mathbf{P}_{E_3}$, respectively.

- If $E := \sigma_{\alpha=\beta}(E_1)$ with E_1 being an n-ary expression, then define T_E by the rule
$$T_E(\$v_1, \ldots, \$v_n) \leftarrow T_{E_1}(\$v_1, \ldots, \$v_n), \theta(\alpha) = \theta(\beta)$$
with θ being the obvious mapping from position variables $\$i$ to the corresponding path variables $\$v_i$. Now take $\mathbf{P}_E$ to be the program with the above rule in addition to the rules in $\mathbf{P}_{E_1}$.
- If $E := \pi_{\alpha_1,\ldots,\alpha_p}(E_1)$ with E_1 being an n-ary expression, then define T_E by the rule
$$T_E(\theta(\alpha_1), \ldots, \theta(\alpha_p)) \leftarrow T_{E_1}(\$v_1, \ldots, \$v_n)$$
with θ being the obvious mapping from position variables $\$i$ to the corresponding path variables $\$v_i$. Now take $\mathbf{P}_E$ to be the program with the above rule in addition to the rules in $\mathbf{P}_{E_1}$.
- If $E := \mathsf{UNPACK}_i(E_1)$ with E_1 being an n-ary expression and for $i \in \{1, \ldots, n\}$, then define T_E by the rule
$$T_E(\$v_1, \ldots, \$v_i, \ldots, \$v_n) \leftarrow T_{E_1}(\$v_1, \ldots, \$v_{i-1}, \langle \$v_i \rangle, \$v_{i+1}, \ldots, \$v_n)$$
and take $\mathbf{P}_E$ to be the program with the above rule in addition to the rules in $\mathbf{P}_{E_1}$.
- If $E := \mathsf{SUB}_i(E_1)$ with E_1 being an n-ary expression and for $i \in \{1, \ldots, n\}$, then define T_E by the rule
$$T_E(\$v_1, \ldots, \$v_n, \$y) \leftarrow T_{E_1}(\$v_1, \ldots, \$v_n), \$v_i = \$x \cdot \$y \cdot \$z$$
and take $\mathbf{P}_E$ to be the program with the above rule in addition to the rules in $\mathbf{P}_{E_1}$.
- If $E := \mathsf{ATOM}_i(E_1)$ with E_1 being an n-ary expression and for $i \in \{1, \ldots, n\}$, then define T_E by the rule
$$T_E(\$v_1, \ldots, \$v_n, @y) \leftarrow T_{E_1}(\$v_1, \ldots, \$v_n), \$v_i = \$x \cdot @y \cdot \$z$$
and take $\mathbf{P}_E$ to be the program with the above rule in addition to the rules in $\mathbf{P}_{E_1}$.
- If $E := E_1 \cup E_2$ with both E_1 and E_2 being n-ary expressions, then define T_E by the two rules
$$T_E(\$v_1, \ldots, \$v_n) \leftarrow T_{E_1}(\$v_1, \ldots, \$v_n)$$
$$T_E(\$v_1, \ldots, \$v_n) \leftarrow T_{E_2}(\$v_1, \ldots, \$v_n).$$
Now take $\mathbf{P}_E$ to be the program with the above rules in addition to the rules in $\mathbf{P}_{E_1}$ and the rules in $\mathbf{P}_{E_2}$.
- If $E := E_1 - E_2$ with both E_1 and E_2 being n-ary expressions, then define T_E by the rule
$$T_E(\$v_1, \ldots, \$v_n) \leftarrow T_{E_1}(\$v_1, \ldots, \$v_n), \neg T_{E_2}(\$v_1, \ldots, \$v_n).$$
Now take $\mathbf{P}_E$ to be the program with the above rule in addition to the rules in $\mathbf{P}_{E_1}$ and the rules in $\mathbf{P}_{E_2}$.
- If $E := E_1 \times E_3$ with E_1 being an n-ary expression and E_3 being an m-ary expression, then define T_E by the rule
$$T_E(\$x_1, \ldots, \$x_n, \$y_1, \ldots, \$y_m) \leftarrow T_{E_1}(\$x_1, \ldots, \$x_n), T_{E_3}(\$y_1, \ldots, \$y_m).$$
Now take $\mathbf{P}_E$ to be the program with the above rule in addition to the rules in $\mathbf{P}_{E_1}$ and the rules in $\mathbf{P}_{E_3}$. □

Our approach to translate in the other direction is for the most part standard. We can make use of the following normal form. Afterwards, this normal form is utilized in Proposition 7.4 establishing the second direction of Theorem 7.1.

Lemma 7.3. *Let* $\mathbf{P}$ *be a nonrecursive Sequence Datalog program that does not use equations. Then there is a nonrecursive program* $\mathbf{P}'$ *computing the same query as* $\mathbf{P}$ *where each rule in* $\mathbf{P}'$ *has one of the following six forms:*

(1) $R_1(v_1, \ldots, v_n) \leftarrow R_2(e_1, \ldots, e_m)$;
(2) $R_1(v_1, \ldots, v_n, e) \leftarrow R_2(v_1, \ldots, v_n)$;
(3) $R_1(v_1, \ldots, v_n) \leftarrow R_2(x_1, \ldots, x_k), R_3(y_1, \ldots, y_\ell)$;
(4) $R_1(v_1, \ldots, v_n) \leftarrow R_2(v_1, \ldots, v_n), \neg R_3(v'_1, \ldots, v'_m)$;
(5) $R_1(v'_1, \ldots, v'_m) \leftarrow R_2(v_1, \ldots, v_n)$;
(6) $R(p) \leftarrow$.

The following restrictions apply:

- *In all forms, $v_1, \ldots, v_n$ are distinct variables. Moreover, in forms 2 to 6, each v_i must be a path variable.*
- *In form 3, the x_i and y_j are path variables and $\{v_1, \ldots, v_n\}$ is contained in $\{x_1, \ldots, x_k\} \cup \{y_1, \ldots, y_\ell\}$.*
- *In forms 4 and 5, $v'_1, \ldots, v'_m$ are distinct variables taken from $\{v_1, \ldots, v_n\}$.*
- *In form 6, p is a path (constant relation).*

Proof of Lemma 7.3. The conversion to normal form is best described on a general example. Consider the following one-rule Sequence Datalog program:

```
T(a·b·c, @x·c·$y, $z·$z)← P1($y·$y, $z·a, @u·d), P2($z·@x·c, d),
                          ¬N1(@x·$y·$z, a·@x), ¬N2(a·b, $y).
```

In what follows, we call the rule that we process the main rule and its stratum the main stratum.

Step 1: Get variables from positive literals.

Step 1.1. Replace every occurrence of a positive atom $P(e_1, \ldots, e_m)$ by a new predicate $H(v_1, \ldots, v_n)$ where $\{v_1, \ldots, v_n\}$ is the set of variables used in the atom. For each H add a new rule of the form $H(v_1, \ldots, v_n) \leftarrow P(e_1, \ldots, e_m)$. Note that these set of rules are guaranteed to be form 1. Moreover, every *atomic* variable in the main rule should be replaced by a new *path* variable.

In case the positive atom does not use variables, then we replace every occurrence by a new predicate $H(\$v)$ with a fresh variable $\$v$. To get this H, we add the two rules $H' \leftarrow P(e_1, \ldots, e_m)$ and $H(a) \leftarrow H'$ for a new predicate H' and $a \in$ **dom**. Note that the first rule is of form 1, while the second added rule is of form 2.

```
H1($y, $z, @u)← P1($y·$y, $z·a, @u·d).
H2($z, @x)← P2($z·@x·c, d).
T(a·b·c, $x·c·$y, $z·$z)← H1($y, $z, $u), H2($z, $x),
                          ¬N1($x·$y·$z, a·$x), ¬N2(a·b, $y).
```

Step 1.2.

- When no positive atoms exist in the main rule, then the rule has no variables. Only in this case, we add to the main stratum a new rule of the form $R(a) \leftarrow$, where R is a new relation name and a is some value from the domain. This added rule is of form 6. Moreover, we add $R(\$v)$ to the body of the main rule, where $\$v$ is a fresh path variable.
- Otherwise, this step should be repeated until only one positive atom remains in the main rule. We remove two positive atoms $H_i(x_1, \ldots, x_n)$ and $H_j(y_1, \ldots, y_m)$, and replace them with $H(v_1, \ldots, v_k)$, where H is a fresh predicate name, and the set of variables vs is the union of the set of xs and ys. In addition, we introduce a new rule of the form

$$H(v_1, \ldots, v_k) \leftarrow H_i(x_1, \ldots, x_n), H_j(y_1, \ldots, y_m)$$

in the main stratum. This rule is of form 3.

```
H1($y, $z, @u)← P1($y·$y, $z·a, @u·d).
H2($z, @x)← P2($z·@x·c, d).
H($y, $z, $u, $x)← H1($y, $z, $u), H2($z, $x).
T(a·b·c, $x·c·$y, $z·$z)← H($y, $z, $u, $x),
                          ¬N1($x·$y·$z, a·$x), ¬N2(a·b, $y).
```

Step 2: Separate each negative literal in an intermediate rule.

Step 2.1. Let $H(v_1, \ldots, v_n)$ be the only positive atom in the body of the rule. Every literal $\neg N(e_1, \ldots, e_m)$ is replaced by a predicate $HN(v_1, \ldots, v_n)$, where HN is a new relation name. Moreover, we add a rule of the form

$$HN(v_1, \ldots, v_n) \leftarrow H(v_1, \ldots, v_n), \neg N(e_1, \ldots, e_m)$$

to the main stratum, and we remove $H(v_1, \ldots, v_n)$ from the main rule.

```
H1($y, $z, @u)← P1($y·$y, $z·a, @u·d).
H2($z, @x)← P2($z·@x·c, d).
H($y, $z, $u, $x)← H1($y, $z, $u), H2($z, $x).
HN1($y, $z, $u, $x)← H($y, $z, $u, $x), ¬N1($x·$y·$z, a·$x).
HN2($y, $z, $u, $x)← H($y, $z, $u, $x), ¬N2(a·b, $y).
T(a·b·c, $x·c·$y, $z·$z)← HN1($y, $z, $u, $x),
                          HN2($y, $z, $u, $x).
```

Step 2.2. We do the same as in step 1.2, leaving us in the end with a single positive atom holding the variables from the original rule. All the rules introduced by this step are of form 3.

```
H1($y, $z, @u)← P1($y·$y, $z·a, @u·d).
H2($z, @x)← P2($z·@x·c, d).
H($y, $z, $u, $x)← H1($y, $z, $u), H2($z, $x).
HN1($y, $z, $u, $x)← H($y, $z, $u, $x), ¬N1($x·$y·$z, a·$x).
HN2($y, $z, $u, $x)← H($y, $z, $u, $x), ¬N2(a·b, $y).
HN($y, $z, $u, $x)← HN1($y, $z, $u, $x), HN2($y, $z, $u, $x).
T(a·b·c, $x·c·$y, $z·$z)← HN($y, $z, $u, $x).
```

Step 3: Generate negated expressions. We next work on the rules that were introduced to deal with the negated atoms.

Step 3.1. In step 2.1 we added rules with negative literals:

$$HN(v_1, \ldots, v_n) \leftarrow H(v_1, \ldots, v_n), \neg N(e_1, \ldots, e_m)$$

For each such added rule, we define a sequence of rules in order to generate the values for the expressions e_i. Since our rule is safe from the beginning, we are guaranteed that all the variables used in these expressions are among the vs.

We inductively generate m rules as follows (where the v's are fresh variables) and add them to the main stratum:

(1) $N_1(v_1, \ldots, v_n, e_1) \leftarrow H(v_1, \ldots, v_n)$
(2) for $1 < i \leq m$, the rule

$$N_i(v_1, \ldots, v_n, v'_1, \ldots, v'_{i-1}, e_i) \leftarrow N_{i-1}(v_1, \ldots, v_n, v'_1, \ldots, v'_{i-1}).$$

Each one of the above rules is of form 2. In addition, we replace $H(v_1, \ldots, v_n)$ in the rule under consideration by

$$N_m(v_1, \ldots, v_n, v'_1, \ldots, v'_m).$$

Moreover, we replace $\neg N(e_1, \ldots, e_m)$ by $\neg N(v'_1, \ldots, v'_m)$.

```
H1($y, $z, @u)← P1($y·$y, $z·a, @u·d).
H2($z, @x)← P2($z·@x·c, d).
H($y, $z, $u, $x)← H1($y, $z, $u), H2($z, $x).
N11($y, $z, $u, $x, $x·$y·$z)← H($y, $z, $u, $x).
N21($y, $z, $u, $x, a·b)← H($y, $z, $u, $x).
N12($y, $z, $u, $x, $n11, a·$x)← N11($y, $z, $u, $x, $n11).
N22($y, $z, $u, $x, $n21, $y)← N21($y, $z, $u, $x, $n21).
HN1($y, $z, $u, $x)← N12($y, $z, $u, $x, $n11, $n12),
                     ¬N1($n11, $n12).
HN2($y, $z, $u, $x)← N22($y, $z, $u, $x, $n21, $n22),
                     ¬N2($n21, $n22).
HN($y, $z, $u, $x)← HN1($y, $z, $u, $x), HN2($y, $z, $u, $x).
T(a·b·c, $x·c·$y, $z·$z)← HN($y, $z, $u, $x).
```

Step 3.2. We have now obtained rules of the form

$$HN(v_1, \ldots, v_n) \leftarrow N_m(v_1, \ldots, v_n, v'_1, \ldots, v'_m), \neg N(v'_1, \ldots, v'_m).$$

We now further replace them with

$$HN(v_1, \ldots, v_n) \leftarrow FN(v_1, \ldots, v_n, v'_1, \ldots, v'_m);$$

where FN is a new relation name. Now this rule is of form 5. Moreover, we add the rule

$$FN(v_1, \ldots, v_n, v'_1, \ldots, v'_m) \leftarrow N_m(v_1, \ldots, v_n, v'_1, \ldots, v'_m), \neg N(v'_1, \ldots, v'_m),$$

which is of form 4, to the main stratum.

```
H1($y, $z, @u)← P1($y·$y, $z·a, @u·d).
H2($z, @x)← P2($z·@x·c, d).
H($y, $z, $u, $x)← H1($y, $z, $u), H2($z, $x).
N11($y, $z, $u, $x, $x·$y·$z)← H($y, $z, $u, $x).
N21($y, $z, $u, $x, a·b)← H($y, $z, $u, $x).
N12($y, $z, $u, $x, $n11, a·$x)← N11($y, $z, $u, $x, $n11).
N22($y, $z, $u, $x, $n21, $y)← N21($y, $z, $u, $x, $n21).
FN1($y, $z, $u, $x, $n11, $n12)←
    N12($y, $z, $u, $x, $n11, $n12), ¬N1($n11, $n12).
FN2($y, $z, $u, $x, $n21, $n22)←
    N22($y, $z, $u, $x, $n21, $n22), ¬N2($n21, $n22).
HN1($y, $z, $u, $x)← FN1($y, $z, $u, $x, $n11, $n12).
HN2($y, $z, $u, $x)← FN2($y, $z, $u, $x, $n21, $n22).
HN($y, $z, $u, $x)← HN1($y, $z, $u, $x), HN2($y, $z, $u, $x).
T(a·b·c, $x·c·$y, $z·$z)← HN($y, $z, $u, $x).
```

Step 4: Generate final head expressions. We are now left to work on the final rule which is normalized in a similar way as step 3.1. The final rule is of the form $T(e_1, \ldots, e_m) \leftarrow H(v_1, \ldots, v_n)$, where by safety it is guaranteed that any variable appearing in any of the es is among the vs.

We inductively generate m rules as follows (where the v's are fresh variables):

(1) $T_1(v_1, \ldots, v_n, e_1) \leftarrow H(v_1, \ldots, v_n)$

(2) for $1 < i \leq m$, the rule

$$T_i(v_1, \ldots, v_n, v'_1, \ldots, v'_{i-1}, e_i) \leftarrow T_{i-1}(v_1, \ldots, v_n, v'_1, \ldots, v'_{i-1}).$$

Each one of the above rules is of form 2. The last thing to be done is to update the main rule to

$$T(v'_1, \ldots, v'_m) \leftarrow T_m(v_1, \ldots, v_n, v'_1, \ldots, v'_m).$$

Now, this rule is of form 5.

```
H1($y, $z, @u)← P1($y·$y, $z·a, @u·d).
H2($z, @x)← P2($z·@x·c, d).
H($y, $z, $u, $x)← H1($y, $z, $u), H2($z, $x).
N11($y, $z, $u, $x, $x·$y·$z)← H($y, $z, $u, $x).
N21($y, $z, $u, $x, a·b)← H($y, $z, $u, $x).
N12($y, $z, $u, $x, $n11, a·$x)← N11($y, $z, $u, $x, $n11).
N22($y, $z, $u, $x, $n21, $y)← N21($y, $z, $u, $x, $n21).
FN1($y, $z, $u, $x, $n11, $n12)←
    N12($y, $z, $u, $x, $n11, $n12), ¬N1($n11, $n12).
FN2($y, $z, $u, $x, $n21, $n22)←
    N22($y, $z, $u, $x, $n21, $n22), ¬N2($n21, $n22).
HN1($y, $z, $u, $x)← FN1($y, $z, $u, $x, $n11, $n12).
HN2($y, $z, $u, $x)← FN2($y, $z, $u, $x, $n21, $n22).
HN($y, $z, $u, $x)← HN1($y, $z, $u, $x), HN2($y, $z, $u, $x).
T1($y, $z, $u, $x, a·b·c)← HN($y, $z, $u, $x).
T2($y, $z, $u, $x, $t1, $x·c·$y)← T1($y, $z, $u, $x, $t1).
T3($y, $z, $u, $x, $t1, $t2, $z·$z)←
    T2($y, $z, $u, $x, $t1, $t2).
T($t1, $t2, $t3)← T3($y, $z, $u, $x, $t1, $t2, $t3).
```

□

Observe that the previous lemma is stated for programs without equations, since we know that equations are redundant in the presence of intermediate predicates. Given the normal form, we show the following:

Proposition 7.4. *Let* **P** *be a non-recursive program over* Γ *whose rules are in normal form. Then, for every IDB relation name* T *in* **P**, *there exists a sequence relational algebra expression* E_T *such that for every instance* I *over* Γ, *we have* $\mathbf{P}(I)(T) = E_T(I)$.

Proof. We establish the proof by structural induction on the form of the rules defining the relation names of **P**. Without loss of generality, we assume that each IDB relation name is defined by a single rule. Otherwise, we can always get the union of the different expressions corresponding to the different rules defining a single relation name using $\cup$ operator.

In what follows, we are also assuming that for every EDB relation name R, there is an expression E_R that is simply defined as $E_R := R$; and that T_1 and T_2 are IDB relation names with E_{T_1} and E_{T_2} being their respective equivalent expressions.

- If T is defined by a rule of form (1). Equivalently, and without loss of generality, we can view that the rule has the form

$$T(v_{i_1}, \ldots, v_{i_n}) \leftarrow T_1(e_1, \ldots, e_m)$$

such the set of all variables appearing in the path expressions are $v_1, \ldots, v_\ell, v_{\ell+1}, \ldots, v_{\ell+j}$ with the first ℓ being path variables and the last j being atomic variables, and moreover, each i is a distinct number from 1 to $\ell + j$.
Suppose that the maximum packing depth in any of the path expressions e is k. We then define the following sequence of sequence relational algbebra expressions:

$$\begin{aligned}
E_{depth_0} &:= \pi_{\$1}(E_{T_1}) \cup \pi_{\$2}(E_{T_1}) \cup \cdots \cup \pi_{\$m}(E_{T_1}).\\
E_{subs_0} &:= \pi_{\$2}(\mathsf{SUB}_1(E_{depth_0})).\\
E_{atom_0} &:= \pi_{\$2}(\mathsf{ATOM}_1(E_{depth_0})).\\
E_{depth_1} &:= \pi_{\$2}(\mathsf{UNPACK}_1(E_{subs_0})).\\
E_{subs_1} &:= \pi_{\$2}(\mathsf{SUB}_1(E_{depth_1})).\\
E_{atom_1} &:= \pi_{\$2}(\mathsf{ATOM}_1(E_{depth_1})).\\
E_{depth_2} &:= \pi_{\$2}(\mathsf{UNPACK}_1(E_{subs_1})).\\
&\vdots\\
E_{depth_k} &:= \pi_{\$2}(\mathsf{UNPACK}_1(E_{subs_{k-1}})).\\
E_{subs_k} &:= \pi_{\$2}(\mathsf{SUB}_1(E_{depth_k})).\\
E_{atom_k} &:= \pi_{\$2}(\mathsf{ATOM}_1(E_{depth_k})).\\
E_{all_s} &:= E_{subs_0} \cup \cdots \cup E_{subs_k}.\\
E_{all_a} &:= E_{atom_0} \cup \cdots \cup E_{atom_k}.
\end{aligned}$$

Intuitively, by composing the unpacking and substring operations, we can generate all subpaths (E_{all_s}) and atoms (E_{all_a}) until the maximum packing depth k of the expressions.
Now, take E_T to be the expression

$$\pi_{\$i_1,\ldots,\$i_n}(\sigma_{\$\ell+j+1=\theta(e_1)}(\cdots \sigma_{\$\ell+j+m=\theta(e_m)}(\underbrace{E_{all\_s} \times \cdots \times E_{all\_s}}_{\ell} \times \underbrace{E_{all\_a} \times \cdots \times E_{all\_a}}_{j} \times E_{T_1})))$$

with θ being the obvious mapping from path and atomic variables v_r to the corresponding position variables $\$r$.
- If T is defined by a rule of the form

$$T(v_1, \ldots, v_n, e) \leftarrow T_1(v_1, \ldots, v_n).$$

Take E_T to be the expression $\pi_{\$1,\ldots,\$n,\theta(e)}(E_{T_1})$ where θ is the obvious mapping from path variables v_i to the corresponding position variables $\$i$.
- If T is defined by a rule of form (3). Equivalently, we can view that the rule has the form

$$T(v_{i_1}, \ldots, v_{i_n}) \leftarrow T_1(v_1, \ldots, v_k), T_2(v_{k+1}, \ldots, v_{k+\ell}), v_{j_1} = v_{j_2}, \ldots, v_{j_r} = v_{j_{r+1}}$$

where all v's in $v_1, \ldots, v_{k+\ell}$ are distinct and each i and j is a number from 1 to $k + \ell$ that is distinct in the case of i's. Then, take E_T to be the expression

$$\pi_{\$i_1,\ldots,\$i_n}(\sigma_{\$j_1=\$j_2}(\cdots \sigma_{\$j_r=\$j_{r+1}}(E_{T_1} \times E_{T_2}))).$$

- If T is defined by a rule of form (4). Equivalently, we can view that the rule has the form

$$T(v_1, \ldots, v_n) \leftarrow T_1(v_1, \ldots, v_n), \neg T_2(v_{i_1}, \ldots, v_{i_m})$$

  where all v's in $v_1, \ldots, v_n$ are distinct and each i is a distinct number from 1 to n. Then, take E_T to be the expression

$$\pi_{\$1,\ldots,\$n}(\sigma_{\$i_1=\$i_1+n}(\cdots \sigma_{\$i_m=\$i_m+n}(E_{T_1} \times (\pi_{\$i_1,\ldots,\$i_m}(E_{T_1}) - E_{T_2})))).$$

- If T is defined by a rule of form (5). Equivalently, we can view that the rule has the form

$$T(v_{i_1}, \ldots, v_{i_m}) \leftarrow T_1(v_1, \ldots, v_n).$$

  where all v's in $v_1, \ldots, v_n$ are distinct and each i is a distinct number from 1 to n. Then, take E_T to be the expression

$$\pi_{\$i_1,\ldots,\$i_m}(E_{T_1}).$$

- If T is defined by a rule of the form $T(p) \leftarrow$. Then, take E_T to be the expression $\{(\$1 : p)\}$.

□

8 Conclusion

Sequence databases and sequence query processing (e.g., [43]) were an active research topic twenty years ago or more. We hope our paper can revive interest in the topic, given its continued relevance for advanced database applications. Systems in use today do support sequences one way or another, but often only nominally, without high expressive power or performance. This situation may cause application builders to bypass the database system and solve their problem in an ad-hoc manner.

Of course, to support data science, there is much current research on database systems and query languages for arrays and tensors, e.g., [7, 25, 28, 35, 42]. However, in this domain, applications are typically focused on supporting linear algebra operations [7, 25, 32]. Such applications are qualitatively different from the more generic type of sequence database queries considered in this paper.

We note that other sequence query language approaches, not based on Datalog, deserve attention as well. There have been proposals based on functional programming [31], on structural recursion [41], and on transducers [9, 11, 19, 20]. On the other hand, a proposal very close in spirit to Sequence Datalog can be found in the work by Grahne and Waller [21] already mentioned in Section 7.

Sequence Datalog is also a very useful language for dealing with non-flat instances. In this paper, for reasons we have explained, we focused on queries from flat instances to flat instances. However, using packing, interesting data structures can be represented in a direct manner. For example, a tree with root label a and childtrees $T_1, \ldots, T_n$ can be represented by the path $a \cdot \langle T_1 \rangle \cdots \langle T_n \rangle$ (where each T_i is represented by a path in turn). Thus, Sequence Datalog can be used as an XML-to-XML query language and more.

We conclude by recalling an intriguing theoretical open problem already mentioned before [23]. It can be stated independently of Sequence Datalog. Consider monadic Datalog with stratified negation over sets of *natural numbers,* with natural number constants and variables, and addition as the only operation. Which functions on finite sets of natural numbers are expressible in this language? We came across this question because this is what our setting reduces to when the set of atomic values is a one-letter alphabet.

Acknowledgments

Most of the work was done while Heba Aamer was supported by the Special Research Fund (BOF) (BOF19OWB16). Rest of the work was done while Heba Aamer was supported by either Fonds Wetenschappelijk Onderzoek Vlaanderen (FWO)-grants (1210525N) or (G062721N). Jan Van den Bussche is also partially supported by the Flanders AI Research Programme.

References

[1] H. Aamer, J. Hidders, J. Paredaens, and J. Van den Bussche. 2021. Expressiveness within sequence datalog. In *Proceedings 40th ACM Symposium on Principles of Databases*. ACM, 70–81.

[2] H. Aamer, M. Montali, and J. Van den Bussche. 2023. What Can Database Query Processing Do for Instance-Spanning Constraints?. In *Business Process Management Workshops*, Cristina Cabanillas, Niels Frederik Garmann-Johnsen, and Agnes Koschmider (Eds.). Springer International Publishing, 132–144.

[3] H. Abdulrab and J.-P. Pécuchet. 1989. Solving word equations. *Journal of Symbolic Computation* 8, 5 (1989), 499–521.

[4] S. Abiteboul, R. Hull, and V. Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

[5] M. Alviano and A. Pieris (Eds.). 2019. *Datalog 2.0 2019: Third International Workshop on the Resurgence of Datalog in Academia and Industry*. CEUR Workshop Proceedings, Vol. 2368.

[6] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. 1989. The Object-Oriented Database System Manifesto. In *Proceedings 1st International Conference on Deductive and Object-Oriented Databases*, W. Kim, J.-M. Nicolas, and S. Nishio (Eds.). Elsevier Science Publishers, 40–57.

[7] P. Barceló, N. Higeura, J. Pérez, and B. Suercaseaux. 2020. On the expressiveness of LARA: A unified language for linear and relational algebra. In *Proceedings 23rd International Conference on Database Theory (Leibniz International Proceedings in Informatics, Vol. 155)*, C. Lutz and J.C. Jung (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 6:1–6:20.

[8] P. Barceló and R. Pichler (Eds.). 2012. *Datalog in Academia and Industry: Second International Workshop, Datalog 2.0*. Lecture Notes in Computer Science, Vol. 7494. Springer.

[9] M. Benedikt, L. Libkin, T. Schwentick, and L. Segoufin. 2003. Definable relations and first-order query languages over strings. *J. ACM* 50, 5 (2003), 694–751.

[10] A.J. Bonner and G. Mecca. 1998. Sequences, Datalog, and Transducers. *J. Comput. System Sci.* 57 (1998), 234–259.

[11] A.J. Bonner and G. Mecca. 2000. Querying sequence databases with transducers. *Acta Informatica* 36 (2000), 511–544.

[12] A.K. Chandra and D. Harel. 1982. Structure and complexity of relational queries. *J. Comput. System Sci.* 25, 1 (1982), 99–128.

[13] J. Chomicki. 1994. Temporal query languages: a survey. In *Temporal Logic: ICTL'94 (Lecture Notes in Computer Science, Vol. 827)*, D.M. Gabbay and H.J. Ohlbach (Eds.). Springer-Verlag, 506–534.

[14] The Commitee for Advanced DBMS Function. 1990. Third-Generation Database System Manifesto. *SIGMOD Record* 19, 3 (1990), 31–44.

[15] O. de Moor, G. Gottlob, T. Furche, and A. Sellers (Eds.). 2011. *Datalog Reloaded: First International Workshop, Datalog 2010*. Lecture Notes in Computer Science, Vol. 6702. Springer.

[16] F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, J. Meseguer, and C. Talcott. 2018. Associative unification and symbolic reasoning modulo associativity in Maude. In *Proceedings 12th International Workshop on Rewriting Logic and Its Applications (Lecture Notes in Computer Science, Vol. 11152)*, V. Rusu (Ed.). Springer, 98–114.

[17] H.-D. Ebbinghaus and J. Flum. 1999. *Finite Model Theory* (second ed.). Springer.

[18] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. 2015. Document spanners: A formal approach to information extraction. *J. ACM* 62, 2 (2015), 12:1–12:51.

[19] S. Ginsburg and X.S. Wang. 1998. Regular sequence operations and their use in database queries. *J. Comput. System Sci.* 56, 1 (1998), 1–26.

[20] G. Grahne, M. Nykänen, and E. Ukkonen. 1999. Reasoning about strings in databases. *J. Comput. System Sci.* 59 (1999), 116–162.

[21] G. Grahne and E. Waller. 2000. How to make SQL stand for String Query Language. In *Research Issues in Structured and Semistructured Database Programming (Lecture Notes in Computer Science, Vol. 1949)*, R.C.H. Connor and A.O. Mendelzon (Eds.). Springer, 61–79.

[22] M. Grohe. 1996. Arity hierarchies. *Annals of Pure and Applied Logic* 82, 2 (1996), 103–163.

[23] J. Hidders, J. Paredaens, and J. Van den Bussche. 2017. J-Logic: Logical foundations for JSON querying. In *Proceedings 36th ACM Symposium on Principles of Databases*. ACM, 137–149.

[24] J. Hidders, J. Paredaens, and J. Van den Bussche. 2020. J-Logic: a Logic for Querying JSON. arXiv:2006.04277.

[25] D. Hutchison, B. Howe, and D. Suciu. 2017. LaraDB: A minimalist kernel for linear and relational algebra computation. In *Proceedings 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, F.N. Afrati and J. Sroka (Eds.). 2:1–2:10.

[26] IEEE Task Force on Process Mining. 2011. Process mining manifesto. https://www.tf-pm.org/resources/manifesto

[27] H.V. Jagadish and F. Olken. 2004. Database management for life science research. *SIGMOD Record* 33, 2 (2004), 15–20.

[28] H. Jananthan, Z. Zhou, et al. 2017. Polystore mathematics of relational algebra. In *Proceedings IEEE International Conference on Big Data*, J.-Y. Nie, Z. Obradovic, T. Suzumura, et al. (Eds.). IEEE, 3180–3189.

[29] Y. Law, H. Wang, and C. Zaniolo. 2011. Relational languages and data models for continuous queries on sequcnes and data streams. *ACM Transactions on Database Systems* 36, 2 (2011), 8:1–8:32.

[30] LDBC Graph Query Language Task Force. 2018. G-CORE: A core for future graph query languages. In *Proceedings 2018 International Conference on Management of Data*. ACM, 1421–1432.

[31] L. Libkin, R. Machlin, and L. Wong. 1996. A query language for multidimensional arrays: design, implementations, and optimization techniques. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD Record, Vol. 25:2)*. ACM Press, 228–239.
[32] S. Luo, Z.J. Gao, M.N. Gubanov, L.L. Perez, D. Jankov, and C.M. Jermaine. 2020. Scalable linear algebra on a relational database system. *Commun. ACM* 63, 8 (2020), 93–101.
[33] G. Mecca and A.J. Bonner. 2001. Query languages for sequence databases: Termination and Complexity. *IEEE Transactions on Knowledge and Data Engineering* 13, 3 (2001), 519–525.
[34] Y. Nahshon, L. Peterfreund, and S. Vansummeren. 2019. Incorporating information extraction in the relational database model. In *Proceedings 19th International Conference on Web and Databases*. ACM, 6:1–6:7.
[35] S. Papadopoulos et al. 2016. The TileDB array data storage manager. *Proceedings of the VLDB Endowment* 10, 4 (2016), 349–360.
[36] L. Peterfreund et al. 2019. Recursive programs for document spanners. In *Proceedings 22nd International Conference on Database Theory (LIPIcs, Vol. 127)*, P. Barcelo and M. Calautti (Eds.). Schloss Dagstuhl–Leibniz Center for Informatics, 13:1–13:18.
[37] F. Pezoa, J.L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. 2016. Foundations of JSON Schema. In *Proceedings 25th International Conference on World Wide Web*. 263–273.
[38] W. Plandowski. 2019. On PSPACE generation of a solution set of a word equation and its applications. *Theoretical Computer Science* 792 (2019), 20–61.
[39] G. Plotkin. 1972. Building-in equational theories. In *Machine Intelligence 7*, B. Meltzer and D. Michie (Eds.). Edinburgh University Press, 73–90.
[40] R. Ramakrishnan et al. 1998. SRQL: Sorted relational query language. In *Proceedings 10th International Conference on Scientific and Statistical Database Management*, M. Rafanelli and M. Jarke (Eds.). IEEE Computer Society, 84–95.
[41] E.L. Robertson, L.V. Saxton, D. Van Gucht, and S. Vansummeren. 2009. Structural recursion as a query language on lists and ordered trees. *Theory of Computing Systems* 44 (2009), 590–619.
[42] F. Rusu and Y. Cheng. 2013. A survey on array storage, query languages, and systems. arXiv:1302.0103.
[43] R. Sadri, C. Zaniolo, A. Zarkesh, et al. 2004. Expressing and optimizing sequence queries in database systems. *ACM Transactions on Database Systems* 29, 2 (2004), 282–318.
[44] P. Seshadri, M. Livny, and R. Ramakrishnan. 1995. SEQ: A model for sequence databases. In *Proceedings 11th International Conference on Data Engineering*, P.S. Yu and A.L.P. Chen (Eds.). IEEE Computer Society, 232–239.
[45] W. Shen et al. 2007. Declarative information extraction using Datalog with embedded extraction. In *Proceedings 33th International Conference on Very Large Data Bases*, Ch. Koch et al. (Eds.). ACM, 1033–1044.
[46] J.D. Ullman. 1988. *Principles of Database and Knowledge-Base Systems*. Vol. I. Computer Science Press.

Received 19 January 2024; revised 26 November 2024; accepted 31 March 2025