Expressive Power of Linear Algebra Query Languages

Floris Geerts
University of Antwerp
floris.geerts@uantwerp.be

Thomas Muñoz
PUC Chile and IMFD Chile
tfmunoz@uc.cl

Cristian Riveros
PUC Chile and IMFD Chile
cristian.riveros@uc.cl

Domagoj Vrgoč
PUC Chile and IMFD Chile
dvrgoc@ing.puc.cl

ABSTRACT

Linear algebra algorithms often require some sort of iteration or recursion as is illustrated by standard algorithms for Gaussian elimination, matrix inversion, and transitive closure. A key characteristic shared by these algorithms is that they allow looping for a number of steps that is bounded by the matrix dimension. In this paper we extend the matrix query language MATLANG with this type of recursion, and show that this suffices to express classical linear algebra algorithms. We study the expressive power of this language and show that it naturally corresponds to arithmetic circuit families, which are often said to capture linear algebra. Furthermore, we analyze several sub-fragments of our language, and show that their expressive power is closely tied to logical formalisms on semiring-annotated relations.

CCS CONCEPTS

• **Theory of computation → Database query languages (principles)**; **Circuit complexity**.

KEYWORDS

complexity; query languages; circuit complexity; linear algebra

ACM Reference Format:
Floris Geerts, Thomas Muñoz, Cristian Riveros, and Domagoj Vrgoč. 2021. Expressive Power of Linear Algebra Query Languages. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3452021.3458314

1 INTRODUCTION

Linear algebra-based algorithms have become a key component in data analytic workflows. As such, there is a growing interest in the database community to integrate linear algebra functionalities into relational database management systems [5, 25, 27–29]. In particular, from a query language perspective, several proposals have recently been put forward to unify relational algebra and linear algebra. Two notable examples of this are: LARA [24], a minimalistic language in which a number of atomic operations on associative tables are proposed, and MATLANG, a query language for matrices [7, 8].

Both LARA and MATLANG have been studied by the database theory community, showing interesting connections to relational algebra and logic. For example, fragments of LARA are known to capture first-order logic with aggregation [4], and MATLANG has been recently shown to be equivalent to a restricted version of the (positive) relational algebra on K-relations, RA^+_K [9], where K denotes a semiring. On the other hand, some standard constructions in linear algebra are out of reach for these languages. For instance, it was shown that under standard complexity-theoretic assumptions, LARA can not compute the inverse of a matrix or its determinant [4], and operations such as the transitive closure of a matrix are known to be inexpressible in MATLANG [8]. Given that these are fundamental constructs in linear algebra, one might wonder how to extend LARA or MATLANG in order to allow expressing such properties.

One approach would be to add these constructions explicitly to the language. Indeed, this was done for MATLANG in [8], and LARA in [4]. In these works, the authors have extended the core language with the trace, the inverse, the determinant, or the eigenvectors operators and study the expressive power of these extensions. However, one can argue that there is nothing special about these operators, apart from that they have been used historically in linear algebra textbooks and they extend the expressibility of the core language. The question here is whether these new operators form a sound and natural choice to extend the core language, or are they just some particular queries that we would like to support.

In this paper we take a more principled approach by studying what are the atomic operations needed to define standard linear algebra algorithms. Inspecting any linear algebra textbook, one sees that most linear algebra procedures heavily rely on the use of for-loops in which iterations happen over the dimensions of the matrices involved. To illustrate this, let us consider the example of computing the transitive closure of a graph. This can be done using a modification of the Floyd-Warshall algorithm [11], which takes as its input an $n \times n$ adjacency matrix A representing our graph, and operates according to the following pseudo-code:

```
for k = 1..n do
    for i = 1..n do
        for j = 1..n do
            A[i, j] := A[i, j] + A[i, k] · A[k, j]
```

After executing the algorithm, all of the non zero entries signify an edge in the (irreflexive) transitive closure graph.

By examining standard linear algebra algorithms such as Gaussian elimination, LU-decomposition, computing the inverse of a

PODS '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8381-3/21/06…$15.00
https://doi.org/10.1145/3452021.3458314

matrix, or its determinant, we can readily see that this pattern continues. Namely, we observe that there are two main components to such algorithms: (i) the ability to iterate up to the matrix dimension; and (ii) the ability to access a particular position in our matrix. In order to allow this behavior in a query language, we propose to extend MATLANG with limited recursion in the form of for-loops, resulting in the language for-MATLANG. To simulate the two components of standard linear algebra algorithms in a natural way, we simulate a loop of the form for $i = 1..n$ do by leveraging canonical vectors. In other words, we use the canonical vectors $b_1 = (1, 0, \ldots)$, $b_2 = (0, 1, \ldots), \ldots$, to access specific rows and columns, and iterate over these vectors. In this way, we obtain a language able to compute important linear algebra operators such as LU-decomposition, determinant, matrix inverse, among other things.

Of course, a natural question to ask now is whether this really results in a language suitable for linear algebra? We argue that the correct way to approach this question is to compare our language to arithmetic circuits, which have been shown to capture the vast majority of existing matrix algorithms, from basic ones such as computing the determinant and the inverse, to complex procedures such as discrete Fourier transformation, and Strassen's algorithm (see [1, 33] for an overview of the area), and can therefore be considered to effectively capture linear algebra. In the main technical result of this paper, we show that for-MATLANG indeed computes the same class of functions over matrices as the ones computed by arithmetic circuit families of bounded degree. As a consequence, for-MATLANG inherits all expressiveness properties of circuits, and thus can simulate any linear algebra algorithm definable by circuits.

Having established that for-MATLANG indeed provides a good basis for a linear algebra language, we move to a more fine-grained analysis of the expressiveness of its different fragments. For this, we aim to provide a connection with logical formalisms, similarly as was done by linking LARA and MATLANG to first-order logic with aggregates [4, 8]. As we show, capturing different logics correspond to restricting how matrix variables are updated in each iteration of the for-loops allowed in for-MATLANG. For instance, if we only allow to add some temporary result to a variable in each iteration (instead of rewriting it completely like in any programming language), we obtain a language, called sum-MATLANG, which is equivalent to RA_K^+, directly extending an analogous result shown for MATLANG, mentioned earlier [9]. We then study updating matrix variables based on another standard linear algebra operator, the Hadamard product, resulting in a fragment called sp-MATLANG, which we show to be equivalent to weighted logics [14]. Finally, in prod-MATLANG we update the variables based on the standard matrix product, and link this fragment to the ones discussed previously.

Contribution and outline.

- After we recall MATLANG in Section 2, we show in Section 3 how for-loops can be added to MATLANG in a natural way. We also observe that for-MATLANG strictly extends MATLANG. In addition, we discuss some design decisions behind the definition of for-MATLANG, noting that our use of canonical vectors results in the availability of an order relation.
- In Section 4 we show that for-MATLANG can compute important linear algebra algorithms in a natural way. We provide expressions in for-MATLANG for LU decomposition (used to solve linear systems of equations), the determinant and matrix inversion.
- More generally, in Section 5 we report our main technical contribution. We show that every uniform family of arithmetic circuits of polynomial degree correspond to a for-MATLANG expression, and vice versa, when a for-MATLANG expression has polynomial degree, then there is an equivalent uniform family of arithmetic circuits. As a consequence, for-MATLANG inherits all expressiveness properties of such circuits.
- Finally, we generalize the semantics of for-MATLANG to matrices with values in a semiring K in Section 6, and show that two natural fragments of for-MATLANG, sum-MATLANG, and sp-MATLANG, are equivalent to the (positive) relational algebra and weighted logics on binary K-relations, respectively. We also briefly comment on a minimal fragment of for-MATLANG, based on prod-MATLANG, that is able to compute matrix inversion.

Due to space limitations, we only include some proof sketches. We refer to the full version [18] for more details.

Related work. We already mentioned LARA [24] and MATLANG [8] whose expressive power was further analyzed in [4, 9, 16, 17]. Extensions of SQL for matrix manipulations are reported in [29]. Most relevant is [25] in which a recursion mechanism is added to SQL which resembles for-loops. The expressive power of this extension is unknown, however. Classical logics with aggregation [22] and fixed-point logics with counting [21] can also be used for linear algebra. For-loop extensions of standard first-order logic are considered in [30] but with a different semantics as ours and in which no aggregation is supported. More generally, for the descriptive complexity of linear algebra we refer to [13, 23]. Most of these works require to encode real numbers inside relations, whereas we treat real numbers as atomic values. We refer to relevant papers related to arithmetic circuits and logical formalisms on semiring-annotated relations in the corresponding sections later in the paper.

2 MATLANG

We start by recalling the matrix query language MATLANG, introduced in [8], which serves as our starting point.

Syntax. Let $\mathcal{V} = \{V_1, V_2, \ldots\}$ be a countably infinite set of *matrix variables* and $\mathcal{F} = \bigcup_{k>1} \mathcal{F}_k$ with $\mathcal{F}_k$ a set of *functions* of the form $f : \mathbb{R}^k \to \mathbb{R}$, where $\mathbb{R}$ denotes the set of real numbers. The syntax of MATLANG expressions is defined by the following grammar[1]:

e	::=	$V \in \mathcal{V}$	(matrix variable)
	|	e^T	(transpose)
	|	$\mathbf{1}(e)$	(one-vector)
	|	$\mathtt{diag}(e)$	(diagonalization of a vector)
	|	$e_1 \cdot e_2$	(matrix multiplication)
	|	$e_1 + e_2$	(matrix addition)
	|	$e_1 \times e_2$	(scalar multiplication)
	|	$f(e_1, \ldots, e_k)$	(pointwise application of $f \in \mathcal{F}_k$).

[1]The original syntax also permits the operator let $V = e_1$ in e_2, which replaces every occurrence of V in e_2 with the value of e_1. Since this is just syntactic sugar, we omit this operator. We also explicitly include matrix addition and scalar multiplication, although these can be simulated by pointwise function applications. Finally, we use transposition instead of conjugate transposition since we work with matrices over $\mathbb{R}$.

MATLANG is parametrized by a collection of functions $\mathcal{F}$ but in the remainder of the paper we only make this dependence explicit, and write MATLANG$[\mathcal{F}]$, for some set $\mathcal{F}$ of functions, when these functions are crucial for some results to hold. When we simply write MATLANG, we mean that any function can be used (including not using any function at all).

When working with MATLANG expressions throughout the paper, we assume the expression e to be given by its parse tree. Alternatively, we can assume the expression to be fully parenthesized. This allows us to disambiguate expressions of the form $A + B \cdot C$, since the parse tree (or the parentheses) will tell us whether we are working with $(A) + (B \cdot C)$, or with $(A + B) \cdot C$. For the sake of brevity, throughout the paper we will also sometimes revert to usual operator precedence syntax, assuming matrix and scalar multiplication to have precedence over the sum (the precedence of the other operators is always unambiguous). This means that when we write $A + B \cdot C$, we are actually referring to $(A) + (B \cdot C)$.

Schemas and typing. To define the semantics of MATLANG expressions we need a notion of schema and well-typedness of expressions. A MATLANG *schema* $\mathcal{S}$ is a pair $\mathcal{S} = (\mathcal{M}, \mathtt{size})$, where $\mathcal{M} \subset \mathcal{V}$ is a finite set of matrix variables, and $\mathtt{size} : \mathcal{M} \mapsto \mathtt{Symb} \times \mathtt{Symb}$ is a function that maps each matrix variable in $\mathcal{M}$ to a pair of *size symbols*. The size function helps us determine whether certain matrix operations, such as matrix multiplication, can be performed for matrices adhering to a schema. We denote size symbols by Greek letters α, β, γ. We also assume that $1 \in \mathtt{Symb}$. To help us determine whether a MATLANG expression can always be evaluated, we define the *type* of an expression e, with respect to a schema $\mathcal{S}$, denoted by $\mathtt{type}_{\mathcal{S}}(e)$, inductively as follows:

- $\mathtt{type}_{\mathcal{S}}(V) := \mathtt{size}(V)$, for a matrix variable $V \in \mathcal{M}$;
- $\mathtt{type}_{\mathcal{S}}(e^T) := (\beta, \alpha)$ if $\mathtt{type}_{\mathcal{S}}(e) = (\alpha, \beta)$;
- $\mathtt{type}_{\mathcal{S}}(\mathbf{1}(e)) := (\alpha, 1)$ if $\mathtt{type}_{\mathcal{S}}(e) = (\alpha, \beta)$;
- $\mathtt{type}_{\mathcal{S}}(\mathtt{diag}(e)) := (\alpha, \alpha)$, if $\mathtt{type}_{\mathcal{S}}(e) = (\alpha, 1)$;
- $\mathtt{type}_{\mathcal{S}}(e_1 \cdot e_2) := (\alpha, \gamma)$ if $\mathtt{type}_{\mathcal{S}}(e_1) = (\alpha, \beta)$, and $\mathtt{type}_{\mathcal{S}}(e_2) = (\beta, \gamma)$;
- $\mathtt{type}_{\mathcal{S}}(e_1 + e_2) := (\alpha, \beta)$ if $\mathtt{type}_{\mathcal{S}}(e_1) = \mathtt{type}_{\mathcal{S}}(e_2) = (\alpha, \beta)$;
- $\mathtt{type}_{\mathcal{S}}(e_1 \times e_2) := (\alpha, \beta)$ if $\mathtt{type}_{\mathcal{S}}(e_1) = (1, 1)$ and $\mathtt{type}_{\mathcal{S}}(e_2) = (\alpha, \beta)$; and
- $\mathtt{type}_{\mathcal{S}}(f(e_1, \ldots, e_k)) := (\alpha, \beta)$, whenever $\mathtt{type}_{\mathcal{S}}(e_1) = \cdots = \mathtt{type}_{\mathcal{S}}(e_k) := (\alpha, \beta)$ and $f \in \mathcal{F}_k$.

We call an expression *well-typed* according to the schema $\mathcal{S}$, if it has a defined type. A well-typed expression can be evaluated regardless of the actual sizes of the matrices assigned to matrix variables, as we describe next.

We will usually work with matrices with different sizes (i.e., "rectangular matrices"), but sometimes it will be helpful to restrict to "square matrices". For this we say that $\mathcal{S}$ is a schema over square matrices if there exists α such that each variable has type (α, α), $(\alpha, 1)$, $(1, \alpha)$, or $(1, 1)$. This restriction will be useful when comparing the expressiveness of our proposal with other formalisms (e.g., see Section 6).

Semantics. We use $\mathsf{Mat}[\mathbb{R}]$ to denote the set of all real matrices and for $A \in \mathsf{Mat}[\mathbb{R}]$, $\dim(A) \in \mathbb{N}^2$ denotes its dimensions. A (MATLANG) *instance* $\mathcal{I}$ over a schema$\mathcal{S} = (\mathcal{M}, \mathtt{size})$ is a pair $\mathcal{I} = (\mathcal{D}, \mathtt{mat})$, where $\mathcal{D} : \mathtt{Symb} \mapsto \mathbb{N}$ assigns a value to each size symbol (and thus in turn dimensions to each matrix variable), and $\mathtt{mat} : \mathcal{M} \mapsto \mathsf{Mat}[\mathbb{R}]$ assigns a concrete matrix to each matrix variable $V \in \mathcal{M}$, such that $\dim(\mathtt{mat}(V)) = \mathcal{D}(\alpha) \times \mathcal{D}(\beta)$ if $\mathtt{size}(V) = (\alpha, \beta)$. That is, an instance tells us the dimensions of each matrix variable, and also the concrete matrices assigned to the variable names in $\mathcal{M}$. Notice that having the function $\mathcal{D}$ allows us to specify when two matrix variables have to be assigned matrices of the same dimension. We assume that $\mathcal{D}(1) = 1$, for every instance $\mathcal{I}$. If e is a well-typed expression according to $\mathcal{S}$, then we denote by $[\![e]\!](\mathcal{I})$ the matrix obtained by evaluating e over $\mathcal{I}$, and define it as follows[2]:

- $[\![V]\!](\mathcal{I}) := \mathtt{mat}(V)$, for $V \in \mathcal{M}$;
- $[\![e^T]\!](\mathcal{I}) := [\![e]\!](\mathcal{I})^T$, where A^T is the transpose of matrix A;
- $[\![\mathbf{1}(e)]\!](\mathcal{I})$ is a $n \times 1$ vector with 1 as all of its entries, where $\dim([\![e]\!](\mathcal{I})) = (n, m)$;
- $[\![\mathtt{diag}(e)]\!](\mathcal{I})$ is a diagonal matrix with the vector $[\![e]\!](\mathcal{I})$ on its main diagonal, and zero in every other position;
- $[\![e_1 \cdot e_2]\!](\mathcal{I}) := [\![e_1]\!](\mathcal{I}) \cdot [\![e_2]\!](\mathcal{I})$;
- $[\![e_1 + e_2]\!](\mathcal{I}) := [\![e_1]\!](\mathcal{I}) + [\![e_2]\!](\mathcal{I})$;
- $[\![e_1 \times e_2]\!](\mathcal{I}) := a \times [\![e_2]\!](\mathcal{I})$ with $[\![e_1]\!](\mathcal{I}) = [a]$; and
- $[\![f(e_1, \ldots, e_k)]\!](\mathcal{I})$ is a matrix A of the same size as $[\![e_1]\!](\mathcal{I})$, and where A_{ij} has the value $f([\![e_1]\!](\mathcal{I})_{ij}, \ldots, [\![e_k]\!](\mathcal{I})_{ij})$.

Although MATLANG forms a solid basis for a matrix query language, it is limited in expressive power. Indeed, MATLANG is subsumed by first order logic with aggregates that uses only three variables [8]. Hence, no MATLANG expression exists that can compute the transitive closure of a graph (represented by its adjacency matrix) or can compute the inverse of a matrix. Rather than extending MATLANG with specific linear algebra operators, such as matrix inversion, we next introduce a limited form of recursion in MATLANG.

3 EXTENDING MATLANG WITH FOR LOOPS

To extend MATLANG with recursion, we take inspiration from classical linear algebra algorithms, such as those described in [31]. Many of these algorithms are based on *for-loops* in which the termination condition for each loop is determined by the matrix dimensions. We have seen how the transitive closure of a matrix can be computed using for-loops in the Introduction. Here we add this ability to MATLANG, and show that the resulting language, called for-MATLANG, can compute properties outside of the scope of MATLANG. We see more advanced examples, such as Gaussian elimination and matrix inversion, later in the paper.

3.1 Syntax and semantics of for-MATLANG

The syntax of for-MATLANG is defined just as for MATLANG but with an extra rule in the grammar:

$$\mathtt{for}\ v, X.\ e \quad \text{(canonical for loop, with } v, X \in \mathcal{V}\text{)}.$$

Intuitively, X is a matrix variable which is iteratively updated according to the expression e. We simulate iterations of the form "for $i \in [1..n]$" by letting v loop over the *canonical vectors* $b_1^n, \ldots, b_n^n$ of dimension n. Here, $b_1^n = [1\ 0 \cdots 0]^T$, $b_2^n = [0\ 1\ 0 \cdots 0]^T$, etc. When n is clear from the context we simply write $b_1, b_2, \ldots$. In addition, the expression e in the rule above may depend on v.

[2] We also use easily definable operations as matrix minus $(-)$ and assume the existence of some constants (such as $[1]$) in $\mathcal{M}$. Additionally, several times we omit the $\times$ operator to weigh expressions.

We next make the semantics precise and start by declaring the type of loop expressions. Given a schema $\mathcal{S} = (\mathcal{M}, \mathtt{size})$, the type of a for-MATLANG expression e, denoted $\mathtt{type}_{\mathcal{S}}(e)$, is defined inductively as in MATLANG but with following extra rule:

- $\mathtt{type}_{\mathcal{S}}(\mathtt{for}\, v, X.\, e) := (\alpha, \beta)$, if $\mathtt{type}_{\mathcal{S}}(e) = \mathtt{type}_{\mathcal{S}}(X) = (\alpha, \beta)$ and $\mathtt{type}_{\mathcal{S}}(v) = (\gamma, 1)$.

We note that $\mathcal{S}$ now necessarily includes v and X as variables and assigns size symbols to them. We also remark that in the definition of the type of $\mathtt{for}\, v, X.\, e$, we require that $\mathtt{type}_{\mathcal{S}}(X) = \mathtt{type}_{\mathcal{S}}(e)$ as this expression updates the content of the variable X in each iteration using the result of e. We further restrict the type of v to be a vector, i.e., $\mathtt{type}_{\mathcal{S}}(v) = (\gamma, 1)$, since v will be instantiated with canonical vectors. A for-MATLANG expression e is well-typed over a schema $\mathcal{S} = (\mathcal{M}, \mathtt{size})$ if its type is defined.

For well-typed expressions we next define their semantics. This is done in an inductive way, just as for MATLANG. To define the semantics of $\mathtt{for}\, v, X.\, e$ over an instance $\mathcal{I}$, we need the following notation. Let $\mathcal{I}$ be an instance and $V \in \mathcal{M}$. Then $\mathcal{I}[V_1 := A_1, \ldots, V_l := A_l]$ denotes an instance that coincides with $\mathcal{I}$, except that the value of the matrix variable V_i is given by the matrix A, for $i \in \{1, \ldots, l\}$. Assume that $\mathtt{type}_{\mathcal{S}}(v) = (\gamma, 1)$, and $\mathtt{type}_{\mathcal{S}}(e) = (\alpha, \beta)$ and $n := \mathcal{D}(\gamma)$. Then, $[\![\mathtt{for}\, v, X.\, e]\!](\mathcal{I})$ is defined iteratively, as follows:

- Let $A_0 := \mathbf{0}$ be the zero matrix of size $\mathcal{D}(\alpha) \times \mathcal{D}(\beta)$.
- For $i = 1, \ldots n$, compute $A_i := [\![e]\!](\mathcal{I}[v := b_i^n, X := A_{i-1}])$.
- Finally, set $[\![\mathtt{for}\, v, X.\, e]\!](\mathcal{I}) := A_n$.

For better understanding how for-MATLANG works, we next provide some examples. We start by showing that the one-vector and diag operators are redundant in for-MATLANG.

Example 3.1. We first show how the one-vector operator $\mathbf{1}(e)$ can be expressed using for loops. It suffices to consider the expression

$$e_{\mathbf{1}} := \mathtt{for}\, v, X.\, X + v,$$

with $\mathtt{type}_{\mathcal{S}}(v) = (\alpha, 1) = \mathtt{type}_{\mathcal{S}}(X)$ if $\mathtt{type}_{\mathcal{S}}(e) = (\alpha, \beta)$. This expression is well-typed and is of type $(\alpha, 1)$. When evaluated over some instance $\mathcal{I}$ with $n = \mathcal{D}(\alpha)$, $[\![e_{\mathbf{1}}]\!](\mathcal{I})$ is defined as follows. Initially, $A_0 := \mathbf{0}$. Then $A_i := A_{i-1} + b_i^n$, i.e., the ith canonical vector is added to A_{i-1}. Finally, $[\![e_{\mathbf{1}}]\!](\mathcal{I}) := A_n$ and this now clearly coincides with $[\![\mathbf{1}(e)]\!](\mathcal{I})$. □

Example 3.2. We next show that the diag operator is redundant in for-MATLANG. Indeed, it suffices to consider the expression

$$e_{\mathrm{diag}} := \mathtt{for}\, v, X.\, X + (v^T \cdot e) \times v \cdot v^T,$$

where e is a for-MATLANG expression of type $(\alpha, 1)$. For this expression to be well-typed, v has to be a vector variable of type $\alpha \times 1$ and X a matrix variable of type (α, α). Then, $[\![e_{\mathrm{diag}}]\!](\mathcal{I})$ is defined as follows. Initially, A_0 is the zero matrix of dimension $n \times n$, where $n = \mathcal{D}(\alpha)$. Then, in each iteration $i \in [1..n]$, $A_i := A_{i-1} + ((b_i^n)^T \cdot [\![e]\!](\mathcal{I})) \times (b_i^n \cdot (b_i^n)^T)$. In other words, A_i is obtained by adding the matrix with value $([\![e]\!](\mathcal{I}))_i$ on position (i, i) to A_{i-1}. Hence, $[\![e_{\mathrm{diag}}]\!](\mathcal{I}) := A_n = [\![\mathrm{diag}(e)]\!](\mathcal{I})$. □

These examples illustrate that we can limit for-MATLANG to consist of the following "core" operators: transposition, matrix multiplication and addition, scalar multiplication, pointwise function application, and for-loops. More specific, for-MATLANG is defined by the following simplified syntax:

$$e ::= V \mid e^T \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid e_1 \times e_2 \mid f(e_1, \ldots, e_k) \mid \mathtt{for}\, v, X.\, e$$

Similarly as for MATLANG, we write for-MATLANG$[\mathcal{F}]$ for some set $\mathcal{F}$ of functions when these are required for the task at hand.

As a final example, we show that we can compute whether a graph contains a 4-clique using for-MATLANG.

Example 3.3. To test for 4-cliques it suffices to consider the following expression with for-loops nested four times:

$$\begin{aligned}
&\mathtt{for}\, u, X_1.\ X_1 + \\
&\quad \mathtt{for}\, v, X_2.\ X_2 + \\
&\qquad \mathtt{for}\, w, X_3.\ X_3 + \\
&\qquad\quad \mathtt{for}\, x, X_4.\ X_4 + \\
&\qquad\qquad u^T \cdot V \cdot v \cdot u^T \cdot V \cdot w \cdot u^T \cdot V \cdot x \cdot \\
&\qquad\qquad v^T \cdot V \cdot w \cdot v^T \cdot V \cdot x \cdot w^T \cdot V \cdot x \cdot g(u, v, w, x)
\end{aligned}$$

with $g(u, v, w, x) = f(u, v) \cdot f(u, w) \cdot f(u, x) \cdot f(v, w) \cdot f(v, x) \cdot f(w, x)$ and $f(u, v) = 1 - u^T \cdot v$. Note that $f(b_i^n, b_j^n) = 1$ if $i \neq j$ and $f(b_i^n, b_j^n) = 0$ otherwise. Hence, $g(b_i^n, b_j^n, b_k^n, b_\ell^n) = 1$ if and only if all i, j, k, l are pairwise different. When evaluating the expression on an instance $\mathcal{I}$ such that V is assigned to the adjacency matrix of a graph, the expression above evaluates to a non-zero value if and only if the graph contains a four-clique. □

Given that MATLANG can not check for 4-cliques [8], we easily obtain the following.

Proposition 3.4. *For any collection of functions $\mathcal{F}$,* MATLANG$[\mathcal{F}]$ *is properly subsumed by* for-MATLANG$[\mathcal{F}]$.

3.2 Design decisions behind for-MATLANG

Loop Initialization. As the reader may have observed, in the semantics of for-loops we always initialize A_0 to the zero matrix $\mathbf{0}$ (of appropriate dimensions). It is often convenient to start the iteration given some concrete matrix originating from the result of evaluation a for-MATLANG expression e_0. To make this explicit, we write $\mathtt{for}\, v, X = e_0.\, e$ and its semantics is defined as above with the difference that $A_0 := [\![e_0]\!](\mathcal{I})$. We observe, however, that $\mathtt{for}\, v, X = e_0.\, e$ can already be expressed in for-MATLANG. In other words, we do not loose generality by assuming an initialization of A_0 by $\mathbf{0}$. The key insight is that in for-MATLANG we can check during evaluation whether or not the current canonical vector b_i^n is equal to b_1^n. This is due to the fact that for-loops iterate over the canonical vectors in a fixed order. We discuss this more in the next paragraph. In particular, we can define a for-MATLANG expression min(), which when evaluated on an instance, returns 1 if its input vector is b_1^n, and returns 0 otherwise. Given min(), consider now the for-MATLANG expression

$$\mathtt{for}\, v, X.\, \mathtt{min}(v) \cdot e(v, X/e_0) + (1 - \mathtt{min}(v)) \cdot e(v, X),$$

where we explicitly list v and X as matrix variables on which e potentially depends on, and where $e(v, X/e_0)$ denotes the expression obtained by replacing every occurrence of X in e with e_0. When evaluating this expression on an instance $\mathcal{I}$, A_0 is initial set to the zero matrix, in the first iteration (when $v = b_1^n$ and thus $\mathtt{min}(v) = 1$) we have $A_1 = [\![e]\!](\mathcal{I}[v := b_1^n, X := [\![e_0]\!](\mathcal{I})])$, and for consecutive iterations (when only the part related to $1 - \mathtt{min}(v)$ applies) A_i is

updated as before. Clearly, the result of this evaluation is equal to $[\![\texttt{for}\, v, X = e_0 \,.\, e]\!](\mathcal{I})$.

As an illustration, we consider the Floyd-Warshall algorithm given in the Introduction.

Example 3.5. Consider the following expression:

$$\begin{aligned} e_{FW} := \texttt{for}\, v_k, X_1 = A.\ & X_1 + \\ \texttt{for}\, v_i, X_2.\ & X_2 + \\ \texttt{for}\, v_j, X_3.\ & X_3 + \\ & (v_i^T \cdot X_1 \cdot v_k \cdot v_k^T \cdot X_1 \cdot v_j) \times v_i \cdot v_j^T \end{aligned}$$

The expression e_{FW} simulates the Floyd-Warshall algorithm by updating the matrix A, which is stored in the variable X_1. The inner sub-expression here constructs an $n \times n$ matrix that contains one in the position (i, j) if and only if one can reach the vertex j from i by going through k, and zero elsewhere. If an instance $\mathcal{I}$ assigns to A the adjacency matrix of a graph, then $[\![e_{FW}]\!](\mathcal{I})$ will be equal to the matrix produced by the algorithm given in the Introduction. □

Order. By introducing for-loops we not only extend MATLANG with bounded recursion, we also introduce order information. Indeed, the semantics of the for operator assumes that the canonical vectors $b_1, b_2, \ldots$ are accessed in this order. It implies, among other things, that for-MATLANG expressions are not permutation-invariant. We can, for example, return the bottom right-most entry in a matrix. Indeed, consider the expression $e_{\max} := \texttt{for}\, v, X \,.\, v$ which, for it to be well-typed, requires both v and X to be of type $(\alpha, 1)$. Then, $[\![e_{\max}]\!](\mathcal{I}) = b_n^n$, for $n = \mathcal{D}(\alpha)$, simply because initially, $X = \mathbf{0}$, but X will be overwritten by $b_1^n, b_2^n, \ldots, b_n^n$, in this order. Hence, at the end of the evaluation b_n^n is returned. To extract the bottom right-most entry from a matrix, we now simply use $e_{\max}^T \cdot V \cdot e_{\max}$.

Although the order is implicit in for-MATLANG, we can explicitly use this order in for-MATLANG expressions. More precisely, the order on canonical vectors is made accessible by using the matrix:

$$S_{\leq} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 0 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & 1 \\ 0 & 0 & \cdots & 1 \end{bmatrix}.$$

We observe that $S_{\leq}$ has the property that $b_i^T \cdot S_{\leq} \cdot b_j = 1$, for two canonical vectors b_i and b_j of the same dimension, if and only if $i \leq j$. Otherwise, $b_i^T \cdot S_{\leq} \cdot b_j = 0$. Interestingly, we can build the matrix $S_{\leq}$ with the following for-MATLANG expression:

$$e_{\leq} = \texttt{for}\, v, X \,.\, X + ((X \cdot e_{\max}) + v) \cdot v^T + v \cdot e_{\max}^T,$$

where $e_{\max}$ is as defined above. The intuition behind this expression is that by using the last canonical vector b_n, as returned by $e_{\max}$, we have access to the last column of X (via the product $X \cdot e_{\max}$). We use this column such that after the i-th iteration, this column contains the i-th column of $S_{\leq}$. This is done by incrementing X with $v \cdot e_{\max}^T$. To construct $S_{\leq}$, in the i-th iteration we further increment X with (i) the current last column in X (via $X \cdot e_{\max} \cdot v^T$) which holds the $(i-1)$-th column of $S_{\leq}$; and (ii) the current canonical vector (via $v \cdot v^T$). Hence, after iteration i, X contains the first i columns of $S_{\leq}$ and holds the ith column of $S_{\leq}$ in its last column. It is now readily verified that $X = S_{\leq}$ after the nth iteration.

It should be clear that if we can compute $S_{\leq}$ using $e_{\leq}$, then we can easily define the following predicates and vectors related with the order of canonical vectors:

- $\mathrm{succ}(u, v)$ such that $\mathrm{succ}(b_i^n, b_j^n) = 1$ if $i \leq j$ and 0 otherwise. Similarly, we can define $\mathrm{succ}^+(u, v)$ such that $\mathrm{succ}^+(b_i^n, b_j^n) = 1$ if $i < j$ and 0 otherwise;
- $\min(u)$ such that $\min(b_i^n) = 1$ if $i = 1$ and $\min(b_i^n) = 0$ otherwise;
- $\max(u)$ such that $\max(b_i^n) = 1$ if $i = n$ and $\min(b_i^n) = 0$ otherwise; and
- $e_{\min}$ and $e_{\max}$ such that $[\![e_{\min}]\!](\mathcal{I}) = b_1^n$ and $[\![e_{\max}]\!](\mathcal{I}) = b_n^n$, respectively.

We refer to the full version [18] for details.

Having order information available results in for-MATLANG to be quite expressive. We heavily rely on order information in the next sections to compute the inverse of matrices and more generally to simulate low complexity Turing machines and arithmetic circuits.

4 ALGORITHMS IN LINEAR ALGEBRA

One of our main motivations to introduce for-loops is to be able to express classical linear algebra algorithms in a natural way. We have seen that for-MATLANG is quite expressive as it can check for cliques, compute the transitive closure, and can even leverage a successor relation on canonical vectors. The big question is how expressive for-MATLANG actually is. We will answer this in the next section by connecting for-MATLANG with arithmetic circuits of polynomial degree. Through this connection, one can move back and forth between for-MATLANG and arithmetic circuits, and as a consequence, anything computable by such a circuit can be computed by for-MATLANG as well. When it comes to specific linear algebra algorithms, the detour via circuits can often be avoided. Indeed, in this section we illustrate that for-MATLANG is able to compute LU decompositions of matrices. These decompositions form the basis of many other algorithms, such as solving linear systems of equations. We further show that for-MATLANG is expressive enough to compute matrix inversion and the determinant. We recall that matrix inversion and determinant need to be explicitly added as separate operators in MATLANG [8] and that the LARA language is unable to invert matrices under usual complexity-theoretic assumptions [4].

4.1 LU decomposition

A lower-upper (LU) decomposition factors a matrix A as the product of a lower triangular matrix L and upper triangular matrix U. This decomposition, and more generally LU decomposition with row pivoting (PLU), underlies many linear algebra algorithms and we next show that for-MATLANG can compute these decompositions.

LU decomposition by Gaussian elimination. LU decomposition can be seen as a matrix form of Gaussian elimination in which the columns of A are reduced, one by one, to obtain the matrix U. The reduction of columns of A is achieved as follows. Consider the first column $[A_{11}, \ldots, A_{n1}]^T$ of A and define $c_1 := [0, \alpha_{21}, \ldots, \alpha_{n1}]^T$ with $\alpha_{j1} := -\frac{A_{j1}}{A_{11}}$. Let $T_1 := I + c_1 \cdot b_1^T$ and consider $T_1 \cdot A$. That is, the jth row of $T_1 \cdot A$ is obtained by multiplying the first row of A by α_{j1} and adding it to the jth row of A. As a result, the first column of $T_1 \cdot A$ is equal to $[A_{11}, 0, \ldots, 0]^T$, i.e., all of its entries below the

diagonal are zero. One then iteratively performs a similar computation, using a matrix $T_i := I + c_i \cdot b_i^T$, where c_i now depends on the ith column in $T_{i-1} \cdots T_1 \cdot A$. As a consequence, $T_i \cdot T_{i-1} \cdots T_1 \cdot A$ is upper triangular in its first i columns. At the end of this process, $T_n \cdots T_1 \cdot A = U$ where U is the desired upper triangular matrix. Furthermore, it is easily verified that each T_i is invertible and by defining $L := T_1^{-1} \cdot \cdots \cdot T_n^{-1}$ one obtains a lower triangular matrix satisfying $A = L \cdot U$. The above procedure is only successful when the denominators used in the definition of the vectors c_i are non-zero. When this is the case we call a matrix A *LU-factorizable*.

In case when such a denominator is zero in one of the reduction steps, one can remedy this situation by *row pivoting*. That is, when the ith entry of the ith row in $T_{i-1} \cdots T_1 \cdot A$ is zero, one replaces the ith row by jth row in this matrix, with $j > i$, provided that the ith entry of the jth row is non-zero. If no such row exists, this implies that all elements below the diagonal are zero already in column i and one can proceed with the next column. One can formulate this in matrix terms by stating that there exists a permutation matrix P, which pivots rows, such that $P \cdot A = L \cdot U$. Any matrix A is LU-factorizable *with pivoting*.

Implementing LU decomposition in for-MATLANG. We first assume that the input matrices are LU-factorizable. We deal with general matrices later on. To implement the above procedure, we need to compute the vector c_i for each column i. We do this in two steps. First, we extract from our input matrix its ith column and set all its upper diagonal entries to zero by means of the expression:

$$\texttt{col}(V, y) := \texttt{for}\, v, X.\ \mathrm{succ}^+(y, v) \cdot (v^T \cdot V \cdot y) \cdot v + X.$$

Indeed, when V is assigned to a matrix A and y to b_i, we have that X will be initially assigned $A_0 = \mathbf{0}$ and in consecutive iterations, $A_j = A_{j-1} + b_j^T \cdot A \cdot b_i$ if $j > i$ (because $\mathrm{succ}^+(b_i, b_j) = 1$ if $j > i$) and $A_j = A_{j-1}$ otherwise (because $\mathrm{succ}^+(b_i, b_j) = 0$ for $j \leq i$). The result of this evaluation is the desired column vector. Using $\texttt{col}(V, y)$, we can now compute T_i by the following expression:

$$\texttt{reduce}(V, y) := e_{\mathsf{Id}} + f_/(\texttt{col}(V, y), -(y^T \cdot V \cdot y) \cdot \mathbf{1}(y)) \cdot y^T,$$

where $f_/ : \mathbb{R}^2 \to \mathbb{R} : (x, y) \mapsto x/y$ is the division function[3]. When V is assigned to A and y to b_i, $f_/(\texttt{col}(A, b_i), -(b_i^T \cdot A \cdot b_i) \cdot \mathbf{1}(b_i))$ is equal to the vector c_i used in the definition of T_i. To perform the reduction steps for all columns, we consider the expression:

$$e_U(V) := (\texttt{for}\, y, X = e_{\mathsf{Id}}.\ \texttt{reduce}(X \cdot V, y) \cdot X) \cdot V.$$

That is, when V is assigned A, X will be initially $A_0 = e_{\mathsf{Id}}$, and then $A_i = \texttt{reduce}(A_{i-1} \cdot A, b_i) = T_i \cdot T_{i-1} \cdots T_1 \cdot A$, as desired. One can show, see [18] for details, that because we can obtain the matrices T_i in for-MATLANG and that these are easily invertible, we can also construct an expression $e_L(V)$ which evaluates to L when V is assigned to A. We may thus conclude the following.

Proposition 4.1. *There exists* for-MATLANG$[f_/]$ *expressions* $e_L(V)$ *and* $e_U(V)$ *such that* $[\![e_L]\!](\mathcal{I}) = L$ *and* $[\![e_U]\!](\mathcal{I}) = U$ *form an LU-decomposition of A, where* $\texttt{mat}(V) = A$ *and A is LU-factorizable.* □

We remark that the proposition holds when division is added as a function in $\mathcal{F}$ in for-MATLANG. When row pivoting is needed, we can also obtain a permutation matrix P such that $P \cdot A = L \cdot U$ holds by means of an expression in for-MATLANG, provided that we additionally allow the function $f_{>0}$, where $f_{>0} : \mathbb{R} \to \mathbb{R}$ is such that $f_{>0}(x) := 1$ if $x > 0$ and $f_{>0}(x) := 0$ otherwise.

Proposition 4.2. *There exists expressions* $e_{L^{-1}P}(M)$ *and* $e_U(M)$ *in* for-MATLANG$[f_/, f_{>0}]$ *such that* $L^{-1} \cdot P = [\![e_{L^{-1}P}]\!](\mathcal{I})$ *and* $U = [\![e_U]\!](\mathcal{I})$, *satisfy* $L^{-1} \cdot P \cdot A = U$. □

Intuitively, by allowing $f_{>0}$ a limited form of `if-then-else` is introduced in for-MATLANG, which is needed to continue reducing columns only when the right pivot has been found.

4.2 Determinant and inverse

Other key linear algebra operations include the computation of the determinant and the inverse of a matrix (if the matrix is invertible). As a consequence of the expressibility in for-MATLANG$[f_/, f_{>0}]$ of LU-decompositions with pivoting, it can be shown that the determinant and inverse can be expressed as well.

However, the results in the next section (connecting for-MATLANG with arithmetic circuits) imply that the determinant and inverse of a matrix can already be defined in for-MATLANG$[f_/]$. So instead of using LU decomposition with pivoting for matrix inversion and computing the determinant, we provide an alternative solution.

More specifically, we rely on Csanky's algorithm for computing the inverse of a matrix [12]. This algorithm uses the characteristic polynomial $p_A(x) = \det(xI - A)$ of a matrix. When expanded as a polynomial $p_A(x) = \sum_{i=0}^{n} c_i x^i$ and it is known that $A^{-1} = \frac{-1}{c_n} \sum_{i=0}^{n-1} c_i A^{n-1-i}$ if $c_n \neq 0$. Furthermore, $c_0 = 1$, $c_n = (-1)^n \det(A)$ and the coefficients c_i of $p_A(x)$ are known to satisfy the system of equations $S \cdot c = s$ given by:

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ S_1 & 2 & 0 & \cdots & 0 & 0 \\ S_2 & S_1 & 3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & 0 \\ S_{n-1} & S_{n-2} & S_{n-3} & \cdots & S_1 & n \end{pmatrix} \cdot \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} S_1 \\ S_2 \\ S_3 \\ \vdots \\ S_n \end{pmatrix},$$

with $S_i = \mathrm{tr}(A^i)$. We can construct all ingredients of this system of equations in for-MATLANG$[f_/]$. We refer to [18] for details. By observing that the matrix S is a lower triangular matrix with non-zero elements on its diagonal, we can write it in the form $D_S + (S - D_S) = D_S \cdot (I + D_S^{-1} \cdot (S - D_S))$ with D_S the diagonal matrix consisting of the diagonal entries of S. Hence $S^{-1} = (I + D_S^{-1} \cdot (S - D_S))^{-1} \cdot D_S^{-1}$. We remark D_S^{-1} can simply be obtained by inverting the (non-zero) elements on the diagonal by means of $f_/$ in for-MATLANG$[f_/]$. Furthermore, we observe that $(I + D_S^{-1}(S - D_S))^{-1} = \sum_{i=0}^{n} (D_S^{-1}(S - D_S))^i$ which is something we can compute in for-MATLANG$[f_/]$ as well. Hence, we can invert S and obtain the vector $(c_1, \ldots, c_n)^T$ as $S^{-1} \cdot s$. To compute A^{-1} it now suffices to compute $\frac{-1}{c_n} \sum_{i=0}^{n-1} c_i A^{n-1-i}$. To find the determinant, we compute $(-1)^n c_n$. All this can be done in for-MATLANG$[f_/]$. We may thus conclude:

Proposition 4.3. *There are* for-MATLANG$[f_/]$ *expressions* $e_{\det}(V)$ *and* $e_{\mathsf{inv}}(V)$ *such that* $[\![e_{\det}]\!](\mathcal{I}) = \det(A)$, *and* $[\![e_{\mathsf{inv}}]\!](\mathcal{I}) = A^{-1}$ *when* $\mathcal{I}$ *assigns V to A and A is invertible.* □

[3] We set division by zero to zero.

5 EXPRESSIVENESS OF FOR LOOPS

In this section we explore the expressive power of for-MATLANG. Given that arithmetic circuits [1] capture most standard linear algebra algorithms [32, 33], they seem as a natural candidate for comparison. Intuitively, an arithmetic circuit is similar to a boolean circuit [3], except that it has gates computing the sum and the product function, and processes elements of $\mathbb{R}$ instead of boolean values. To connect for-MATLANG to arithmetic circuits we need a notion of uniformity of such circuits. After all, a for-MATLANG expression can take matrices of arbitrary dimensions as input and we want to avoid having different circuits for each dimension. To handle inputs of different sizes, we thus consider a notion of uniform families of arithmetic circuits, defined via a Turing machine generating a description of the circuit for each input size n.

What we show in the remainder of this section is that any function f which operates on matrices, and is computed by a uniform family of arithmetic circuits of bounded degree, can also be computed by a for-MATLANG expression, and vice versa. In order to keep the notation light, we will focus on for-MATLANG schemas over "square matrices" where each variable has type (α, α), $(\alpha, 1)$, $(1, \alpha)$, or $(1, 1)$, although all of our results hold without these restrictions as well. In what follows, we will write for-MATLANG to denote for-MATLANG$[\emptyset]$, i.e. the fragment of our language with no additional pointwise functions. We begin by defining circuits and then show how circuit families can be simulated by for-MATLANG.

5.1 From arithmetic circuits to for-MATLANG

Let us first recall the definition of arithmetic circuits. An *arithmetic circuit* Φ over a set $X = \{x_1, \ldots, x_n\}$ of input variables is a directed acyclic labeled graph. The vertices of Φ are called *gates* and denoted by $g_1, \ldots, g_m$; the edges in Φ are called *wires*. The children of a gate g correspond to all gates g' such that (g, g') is an edge. The parents of g correspond to all gates g' such that (g', g) is an edge. The *in-degree*, or a *fan-in*, of a gate g refers to its number of children, and the *out-degree* to its number of parents. We will not assume any restriction on the in-degree of a gate, and will thus consider circuits with unbounded fan-in. Gates with in-degree 0 are called *input gates* and are labeled by either a variable in X or a constant 0 or 1. All other gates are labeled by either + or $\times$, and are referred to as *sum gates* or *product gates*, respectively. Gates with out-degree 0 are called *output gates*. When talking about arithmetic circuits, one usually focuses on circuits with n input gates and a single output gate.

The *size* of Φ, denoted by $|\Phi|$, is its number of gates and wires. The *depth* of Φ, denoted by $\mathsf{depth}(\Phi)$, is the length of the longest directed path from any of its output gates to any of the input gates. The *degree* of a gate is defined inductively: an input gate has degree 1, a sum gate has a degree equal to the maximum of degrees of its children, and a product gate has a degree equal to the sum of the degrees of its children. When Φ has a single output gate, the *degree* of Φ, denoted by $\mathsf{degree}(\Phi)$, is defined as the degree of its output gate. If Φ has a single output gate and its input gates take values from $\mathbb{R}$, then Φ corresponds to a polynomial in $\mathbb{R}[X]$ in a natural way. In this case, the degree of Φ equals the degree of the polynomial corresponding to Φ. If $a_1, \ldots, a_n$ are values in $\mathbb{R}$, then the result of the circuit on this input is the value computed by the corresponding polynomial, denoted by $\Phi(a_1, \ldots, a_k)$.

In order to handle inputs of different sizes, we use the notion of uniform circuit families. An *arithmetic circuit family* is a set of arithmetic circuits $\{\Phi_n \mid n = 1, 2, \ldots\}$ where Φ_n has n input variables and a single output gate. An arithmetic circuit family is *uniform* if there exists a LOGSPACE-Turing machine, which on input 1^n, returns an encoding of the arithmetic circuit Φ_n for each n. We observe that uniform arithmetic circuit families are necessarily of polynomial size. Another important parameter is the circuit depth. A circuit family is of logarithmic depth, whenever $\mathsf{depth}(\Phi_n) \in O(\log n)$. We now show that for-MATLANG subsumes uniform arithmetic circuit families that are of logarithmic depth.

Theorem 5.1. *For any uniform arithmetic circuit family $\{\Phi_n \mid n = 1, 2, \ldots\}$ of logarithmic depth there is a* for-MATLANG *schema $\mathcal{S}$ and an expression e_Φ using a matrix variable v, with $\mathsf{type}_\mathcal{S}(v) = (\alpha, 1)$ and $\mathsf{type}_\mathcal{S}(e) = (1, 1)$, such that for any input values $a_1, \ldots, a_n$:*

- *If $\mathcal{I} = (\mathcal{D}, \mathsf{mat})$ is a* MATLANG *instance such that $\mathcal{D}(\alpha) = n$ and $\mathsf{mat}(v) = [a_1 \ldots a_n]^T$.*
- *Then $[\![e_\Phi]\!](\mathcal{I}) = \Phi_n(a_1, \ldots, a_n)$.*

It is important to note that the expression e_Φ does not change depending on the input size, meaning that it is uniform in the same sense as the circuit family being generated by a single Turing machine. The different input sizes for a for-MATLANG instance are handled by the typing mechanism of the language.

Proof sketch. The proof of this Theorem, which is the deepest technical result of the paper, depends crucially on two facts: (i) that any polynomial time Turing machine working within linear space and producing linear size output, can be simulated via a for-MATLANG expression; and (ii) that evaluating an arithmetic circuit Φ_n can be done using two stacks of depth n.

Evaluating Φ_n on input $(a_1, \ldots, a_n)$ can be done in a depth-first manner by maintaining two stacks: the gates-stack that tracks the current gate being evaluated, and the values-stack that stores the value that is being computed for this gate. The idea behind having two stacks is that whenever the number of items on the gates-stack is higher by one than the number of items on the values-stack, we know that we are processing a fresh gate, and we have to initialize its current value (to 0 if it is a sum gate, and to 1 if it is a product gate), and push it to the values-stack. We then proceed by processing the children of the head of the gates-stack one by one, and aggregate the results using sum if we are working with a sum gate, and by using product otherwise.

In order to access the information about the gate we are processing (such as whether it is a sum or a product gate, the list of its children, etc.) we use the uniformity of our circuit family. Namely, we know that we can generate the circuit Φ_n with a LOGSPACE-Turing machine M_Φ by running it on the input 1^n. Using this machine, we can in fact compute all the information needed to run the two-stack algorithms described above. For instance, we can construct a LOGSPACE machine that checks, given two gates g_1 and g_2, whether g_2 is a child of g_1. Similarly, we can construct a machine that, given g_1 and g_2 tells us whether g_2 is the final child of g_1, or the one that produces the following child of g_1 (according to the ordering given by the machine M_Φ). Defining these machines based on M_Φ is similar to the algorithm for the composition of two LOGSPACE transducers, and is commonly used to evaluate arithmetic circuits [1].

To simulate the circuit evaluation algorithm that uses two stacks, in for-MATLANG we can use a binary matrix of size $n \times n$, where n is the number of inputs. The idea here is that the gates-stack corresponds to the first $n-3$ columns of the matrix, with each gate being encoded as a binary number in positions $1, \ldots, n-3$ of a row. The remaining three columns are reserved for the values-stack, the number of elements on the gates stack, and the number of elements on the values stack, respectively. The number of elements is encoded as a canonical vector of size n. Here we crucially depend on the fact that the circuit is of logarithmic depth, and therefore the size of the two stacks is bounded by n (apart from the portion before the asymptotic bound kicks-in, which can be hard-coded into the expression e_Φ). Similarly, given that the circuits are of polynomial size, we can assume that gate ids can be encoded into $n-3$ bits.

This matrix is then updated in the same way as the two-stack algorithm. It processes gates one by one, and using the successor relation for canonical vectors determines whether we have more elements on the gates stack. In this case, a new value is added to the values stack (0 if the gate is a sum gate, and 1 otherwise), and the process continues. Information about the next child, last child, or input value, are obtained using the expression which simulates the Turing machine generating this data about the circuit (the machines used never produce an output longer than their input). Given that the size of the circuit is polynomial, say n^k, we can initialize the matrix with the output gate only, and run the simulation of the two-stack algorithm for n^k steps (by iterating k times over size n canonical vectors). After this, the value in position $(1, n-2)$ (the top of the values stack) holds the final results. □

While Theorem 5.1 gives us an idea on how to simulate arithmetic circuits, it does not tell us which classes of functions over real numbers can be computed by for-MATLANG expressions. In order to answer this question, we note that arithmetic circuits can be used to compute functions over real numbers. Formally, a circuit family $\{\Phi_n \mid n = 1, 2, \ldots\}$ computes a function $f : \bigcup_{n \geq 1} \mathbb{R}^n \mapsto \mathbb{R}$, if for any $a_1, \ldots a_n \in \mathbb{R}$ it holds that $\Phi_n(a_1, \ldots, a_n) = f(a_1, \ldots, a_n)$. To make the connection with for-MATLANG, we need to look at circuit families of bounded degree.

A circuit family $\{\Phi_n \mid n = 1, 2, \ldots\}$ is said to be of *polynomial degree* if $\text{degree}(\Phi_n) \in O(p(n))$, for some polynomial $p(n)$. Note that polynomial size circuit families are not necessarily of polynomial degree. An easy corollary of Theorem 5.1 tells us that all functions computed by uniform family of circuits of polynomial degree and logarithmic depth can be simulated using for-MATLANG expressions. However, we can actually drop the restriction on circuit depth due to the result of Valiant et. al. [35] and Allender et. al. [2] which says that any function computed by a uniform circuit family of polynomial degree (and polynomial depth), can also be computed by a uniform circuit family of logarithmic depth. Using this fact, we can conclude the following:

Corollary 5.2. *For any function f computed by a uniform family of arithmetic circuits of polynomial degree, there is an equivalent* for-MATLANG *formula e_f.*

Note that there is nothing special about circuits that have a single output, and both Theorem 5.1 and Corollary 5.2 also hold for functions $f : \bigcup_{n \geq 1} \mathbb{R}^n \mapsto \mathbb{R}^{s(n)}$, where s is a polynomial. Namely, in this case, we can assume that circuits for f have multiple output gates, and that the depth reduction procedure of [2] is carried out for each output gate separately. Similarly, the construction underlying the proof of Theorem 5.1 can be performed for each output gate independently, and later composed into a single output vector.

5.2 From for-MATLANG to circuits

Now that we know that arithmetic circuits can be simulated using for-MATLANG expressions, it is natural to ask whether the same holds in the other direction. That is, we are asking whether for each for-MATLANG expression e over some schema $\mathcal{S}$ there is a uniform family of arithmetic circuits computing precisely the same result depending on the input size.

In order to handle the fact that for-MATLANG expressions can produce any matrix, and not just a single value, as their output, we need to consider circuits which have multiple output gates. Similarly, we need to encode matrix inputs of a for-MATLANG expression in our circuits. We will write $\Phi(A_1, \ldots, A_k)$, where Φ is an arithmetic circuit with multiple output gates, and each A_i is a matrix of dimensions $\alpha_i \times \beta_i$, with $\alpha_i, \beta_i \in \{n, 1\}$ to denote the input matrices for a circuit Φ. We will also write $\texttt{type}(\Phi) = (\alpha, \beta)$, with $\alpha, \beta \in \{n, 1\}$, to denote the size of the output matrix for Φ. We call such circuits *arithmetic circuits over matrices*. When $\{\Phi_n \mid n = 1, 2, \ldots\}$ is a uniform family of arithmetic circuits over matrices, we will assume that the Turing machine for generating Φ_n also gives us the information about how to access a position of each input matrix, and how to access the positions of the output matrix, as is usually done when handling matrices with arithmetic circuits [32]. The notion of degree is extended to be the sum of the degrees of all the output gates. With this at hand, we can now show the following result.

Theorem 5.3. *Let e be a* for-MATLANG *expression over a schema $\mathcal{S}$, and let $V_1, \ldots, V_k$ be the variables of e such that $\texttt{type}_{\mathcal{S}}(V_i) \in \{(\alpha, \alpha), (\alpha, 1), (1, \alpha), (1, 1)\}$. Then there exists a uniform arithmetic circuit family over matrices $\Phi_n(A_1, \ldots, A_k)$ such that:*

- *For any instance $\mathcal{I} = (\mathcal{D}, \texttt{mat})$ such that $\mathcal{D}(\alpha) = n$ and $\texttt{mat}(V_i) = A_i$ it holds that:*
- *$[\![e]\!](\mathcal{I}) = \Phi_n(A_1, \ldots, A_k)$.*

It is not difficult to see that the proof of Theorem 5.1 can also be extended to support arithmetic circuits over matrices. In order to identify the class of functions computed by for-MATLANG expressions, we need to impose one final restriction: than on the degree of an expression. In particular, we will be interested in for-MATLANG expressions of polynomial degree. Formally, an expression e is of polynomial degree, whenever there is an equivalent circuit family for e of a polynomial degree. For example, all for-MATLANG expressions seen so far have polynomial degree. With this definition, we can now identify the class of functions for which arithmetic circuits and for-MATLANG formulas are equivalent. This is the main technical contribution of the paper.

Corollary 5.4. *Let f be a function with input matrices $A_1, \ldots, A_k$ of dimensions $\alpha \times \beta$, with $\alpha, \beta \in \{n, 1\}$. Then, f is computed by a uniform circuit family over matrices of polynomial degree if and only if there is a* for-MATLANG *expression of polynomial degree for f.*

Note that this result crucially depends on the fact that expressions in for-MATLANG are of polynomial degree. Some for-MATLANG

expressions are easily seen to produce results which are not polynomial. An example of such an expression is, for instance, $e_{\exp} = \text{for } v, X = A.\ X \cdot X$, over a schema $\mathcal{S}$ with $\text{type}_{\mathcal{S}}(v) = (\gamma, 1)$, and $\text{type}_{\mathcal{S}}(X) = (1, 1)$. Over an instance which assigns n to γ this expression computes the function a^{2^n}, for $A = [a]$. Therefore, a natural question to ask then is whether we can determine the degree of a for-MATLANG expression. Unfortunately, as we show in the following proposition this question is in fact undecidable.

Proposition 5.5. *Given a* for-MATLANG *expression e over a schema $\mathcal{S}$, it is undecidable to check whether e is of polynomial degree.*

Of course, one might wonder whether it is possible to define a syntactic subclass of for-MATLANG expressions that are of polynomial degree and can still express many important linear algebra algorithms. We identify one such class in Section 6.1, called sum-MATLANG, and in fact show that this class is powerful enough to capture relational algebra on (binary) K-relations.

5.3 Supporting additional operators

The equivalence of for-MATLANG and arithmetic circuits we proved above assumes that circuits can only use the sum and product gates (note that even without the sum and the product function, for-MATLANG can simulate these operations via matrix sum/product). However, both arithmetic circuits and expressions in for-MATLANG can be allowed to use a multitude of functions over $\mathbb{R}$. The most natural addition to the set of functions is the division operator, which is crucially needed in many linear algebra algorithms, such as, for instance, Gaussian elimination, or LU decomposition (recall Proposition 4.1). Interestingly, the equivalence in this case still holds, mainly due to a surprising result which shows that (almost all) divisions can in fact be removed for arithmetic circuits which allow sum, product, and division gates [1].

More precisely, in [6, 26, 34] it was shown that for any function of the form $f = g/h$, where g and h are relatively prime polynomials of degree d, if f is computed by an arithmetic circuit of size s, then both g and h can be computed by a circuit whose size is polynomial in $s + d$. Given that we can postpone the division without affecting the final result, this, in essence, tells us that division can be eliminated (pushed to the top of the circuit), and we can work with sum-product circuits instead. The degree of a circuit for f, can then be defined as the maximum of degrees of circuits for g and h. Given this fact, we can again use the depth reduction procedure of [2], and extend Corollary 5.4 to circuits with division.

Corollary 5.6. *Let f be a function taking as its input matrices $A_1, \ldots, A_k$ of dimensions $\alpha \times \beta$, with $\alpha, \beta \in \{n, 1\}$. Then, f is computed by a uniform circuit family over matrices of polynomial degree that allows divisions, if and only if there is a* for-MATLANG$[f_/]$ *expression of polynomial degree for f.*

An interesting line of future work here is to see which additional functions can be added to arithmetic circuits and for-MATLANG formulas, in order to preserve their equivalence. Note that this will crucially depend on the fact that these functions have to allow the depth reduction of [2] in order to be supported.

6 RESTRICTING THE POWER OF FOR LOOPS

We conclude the paper by zooming in on some special fragments of for-MATLANG and in which matrices can take values from an arbitrary (commutative) semiring K. In particular, we first consider sum-MATLANG, in which iterations can only perform additive updates, and show that it is equivalent in expressive power to the (positive) relational algebra on K-relations. We then extend sum-MATLANG such that also updates involving pointwise-multiplication (Hadamard product) are allowed. The resulting fragment, sp-MATLANG, is shown to be equivalent in expressive power to weighted logics. Finally, we consider the fragment prod-MATLANG in which updates involving sum and matrix multiplication, and possibly order information, is allowed. From the results in Section 4, we infer that the latter fragment suffices to compute matrix inversion. An overview of the fragments and their relationships are depicted in Figure 1.

6.1 Summation MATLANG and relational algebra

When defining 4-cliques and in several other expressions we have seen so far, we only update X by adding some matrix to it. This restricted form of for-loop proved useful throughout the paper, and we therefore introduce it as a special operator. That is, we define:

$$\Sigma v.e := \text{for } v, X.\ X + e.$$

Here, $e = e(v)$. We define the subfragment of for-MATLANG, called sum-MATLANG, to consist of the Σ operator plus the "core" operators in MATLANG, namely, transposition, matrix multiplication and addition, scalar multiplication, and pointwise function applications.

One property of sum-MATLANG is that it only allows expressions of polynomial degree. Indeed, one can easily show that sum-MATLANG can only create matrix entries that are polynomial in the dimension n of the expression. More precisely, we can show the following:

Proposition 6.1. *Every expression in* sum-MATLANG *is of polynomial degree.*

Interestingly enough, this restricted version of for-loop already allows us to capture the MATLANG operators that are not present in the syntax of sum-MATLANG. More precisely, we see from Examples 3.1 and 3.2 that the one-vector and diag operator are expressible in sum-MATLANG. Combined with the observation that the 4-clique expression of Example 3.3 is in sum-MATLANG, the following result is immediate.

Corollary 6.2. MATLANG *is strictly subsumed by* sum-MATLANG.

What operations over matrices can be defined with sum-MATLANG that is beyond MATLANG? In [9], it was shown that MATLANG is strictly included in the (positive) relational algebra on K-relations, denoted by RA_K^+ [20].[4] It thus seems natural to compare the expressive power of sum-MATLANG with RA_K^+. The main result in this section is that sum-MATLANG and RA_K^+ are equally expressive over binary schemas. To make this equivalence precise, we next give the definition of RA_K^+ [20] and then show how to connect both formalisms.

Let $\mathbb{D}$ be a data domain and $\mathbb{A}$ a set of attributes. A relational signature is a finite subset of $\mathbb{A}$. A relational schema is a function $\mathcal{R}$ on finite set of symbols $\text{dom}(\mathcal{R})$ such that $\mathcal{R}(R)$ is a relation

[4] The algebra used in [9] differs slightly from the one given in [20]. In this paper we work with the original algebra RA_K^+ as defined in [20].

signature for each $R \in \mathrm{dom}(\mathcal{R})$. To simplify the notation, from now on we write R to denote both the symbol R and the relational signature $\mathcal{R}(R)$. Furthermore, we write $R \in \mathcal{R}$ to say that R is a symbol of $\mathcal{R}$. For $R \in \mathcal{R}$, an R-tuple is a function $t : R \to \mathbb{D}$. We denote by $\mathbf{tuples}(R)$ the set of all R-tuples. Given $X \subseteq R$, we denote by $t[X]$ the restriction of t to the set X.

A semiring $(K, \oplus, \odot, \mathbb{0}, \mathbb{1})$ is an algebraic structure where K is a non-empty set, $\oplus$ and $\odot$ are binary operations over K, and $\mathbb{0}, \mathbb{1} \in K$. Furthermore, $\oplus$ and $\odot$ are associative operations, $\mathbb{0}$ and $\mathbb{1}$ are the identities of $\oplus$ and $\odot$ respectively, $\oplus$ is a commutative operation, $\odot$ distributes over $\oplus$, and $\mathbb{0}$ annihilates K (i.e. $\mathbb{0} \odot k = k \odot \mathbb{0} = \mathbb{0}$). As usual, we assume that all semirings in this paper are commutative, namely, $\odot$ is also commutative. We use $\bigoplus_X$ or $\bigodot_X$ for the $\oplus$- or $\odot$-operation over all elements in X, respectively. Typical examples of semirings are the reals $(\mathbb{R}, +, \times, 0, 1)$, the natural numbers $(\mathbb{N}, +, \times, 0, 1)$, and the boolean semiring $(\{0, 1\}, \vee, \wedge, 0, 1)$.

Fix a semiring $(K, \oplus, \odot, \mathbb{0}, \mathbb{1})$ and a relational schema $\mathcal{R}$. A K-relation of $R \in \mathcal{R}$ is a function $r : \mathbf{tuples}(R) \to K$ such that the support $\mathrm{supp}(r) = \{t \in \mathbf{tuples}(R) \mid r(t) \neq \mathbb{0}\}$ is finite. A K-instance $\mathcal{J}$ of $\mathcal{R}$ is a function that assigns relational signatures of $\mathcal{R}$ to K-relations. Given $R \in \mathcal{R}$, we denote by $R^{\mathcal{J}}$ the K-relation associated to R. Recall that $R^{\mathcal{J}}$ is a function and hence $R^{\mathcal{J}}(t)$ is the value in K assigned to t. Given a K-relation r we denote by $\mathbf{adom}(r)$ the active domain of r defined as $\mathbf{adom}(r) = \{t(a) \mid t \in \mathrm{supp}(r) \wedge a \in R\}$. Then the active domain of an K-instance $\mathcal{J}$ of $\mathcal{R}$ is defined as $\mathbf{adom}(\mathcal{J}) = \bigcup_{R \in \mathcal{R}} \mathbf{adom}(R^{\mathcal{J}})$.

An RA^+_K expression Q over $\mathcal{R}$ is given by the following syntax:

$$Q \quad := \quad R \mid Q \cup Q \mid \pi_X(Q) \mid \sigma_X(Q) \mid \rho_f(Q) \mid Q \bowtie Q$$

where $R \in \mathcal{R}$, $X \subseteq \mathbb{A}$ is finite, and $f : X \to Y$ is a one to one mapping with $Y \subseteq \mathbb{A}$. One can extend the schema $\mathcal{R}$ to any expression over $\mathcal{R}$ recursively as follows: $\mathcal{R}(R) = R$, $\mathcal{R}(Q \cup Q') = \mathcal{R}(Q)$, $\mathcal{R}(\pi_X(Q)) = X$, $\mathcal{R}(\sigma_X(Q)) = \mathcal{R}(Q)$, $\mathcal{R}(\rho_f(Q)) = X$ where $f : X \to Y$, and $\mathcal{R}(Q \bowtie Q') = \mathcal{R}(Q) \cup \mathcal{R}(Q')$ for every expressions Q and Q'. We further assume that any expression Q satisfies the following syntactic restrictions: $\mathcal{R}(Q') = \mathcal{R}(Q'')$ whenever $Q = Q' \cup Q''$, $X \subseteq \mathcal{R}(Q')$ whenever $Q = \pi_X(Q')$ or $Q = \sigma_X(Q')$, and $Y = \mathcal{R}(Q')$ whenever $Q = \rho_f(Q')$ with $f : X \to Y$.

Given an RA^+_K expression Q and a K-instance $\mathcal{J}$ of $\mathcal{R}$, we define the semantics $[\![Q]\!]_{\mathcal{J}}$ as a K-relation of $\mathcal{R}(Q)$ as follows. For $X \subseteq \mathbb{A}$, let $\mathrm{Eq}_X(t) = \mathbb{1}$ when $t(a) = t(b)$ for every $a, b \in X$, and $\mathrm{Eq}_X(t) = \mathbb{0}$ otherwise. For every tuple $t \in \mathcal{R}(Q)$:

if $Q = R$, then	$[\![Q]\!]_{\mathcal{J}}(t) = R^{\mathcal{J}}(t)$
if $Q = Q_1 \cup Q_2$, then	$[\![Q]\!]_{\mathcal{J}}(t) = [\![Q_1]\!]_{\mathcal{J}}(t) \oplus [\![Q_2]\!]_{\mathcal{J}}(t)$
if $Q = \pi_X(Q')$, then	$[\![Q]\!]_{\mathcal{J}}(t) = \bigoplus_{t' : t'[X] = t} [\![Q']\!]_{\mathcal{J}}(t')$
if $Q = \sigma_X(Q')$, then	$[\![Q]\!]_{\mathcal{J}}(t) = [\![Q']\!]_{\mathcal{J}}(t) \odot \mathrm{Eq}_X(t)$
if $Q = \rho_f(Q')$, then	$[\![Q]\!]_{\mathcal{J}}(t) = [\![Q']\!]_{\mathcal{J}}(t \circ f)$
if $Q = Q_1 \bowtie Q_2$, then	$[\![Q]\!]_{\mathcal{J}}(t) = [\![Q_1]\!]_{\mathcal{J}}(t[Y]) \odot [\![Q_2]\!]_{\mathcal{J}}(t[Z])$,

where $Y = \mathcal{R}(Q_1)$ and $Z = \mathcal{R}(Q_2)$. It is important to note that the $\bigoplus$-operation in the semantics of $\pi_X(Q')$ is well-defined given that the support of $[\![Q']\!]_{\mathcal{J}}$ is always finite.

We are now ready for comparing sum-MATLANG with RA^+_K . First of all, we need to extend sum-MATLANG from $\mathbb{R}$ to any semiring. Let $\mathrm{Mat}[K]$ denote the set of all K-matrices. Similarly as for MATLANG over $\mathbb{R}$, given a MATLANG schema $\mathcal{S}$, a K-instance $\mathcal{I}$ over $\mathcal{S}$ is a pair $\mathcal{I} = (\mathcal{D}, \mathtt{mat})$, where $\mathcal{D} : \mathtt{Symb} \mapsto \mathbb{N}$ assigns a value to each size symbol, and $\mathtt{mat} : \mathcal{M} \mapsto \mathrm{Mat}[K]$ assigns a concrete K-matrix to each matrix variable. Then it is straightforward to extend the semantics of MATLANG, for-MATLANG, and sum-MATLANG from $(\mathbb{R}, +, \times, 0, 1)$ to $(K, \oplus, \odot, \mathbb{0}, \mathbb{1})$ by switching + with $\oplus$ and $\times$ with $\odot$.

The next step to compare sum-MATLANG with RA^+_K is to represent K-matrices as K-relations. Let $\mathcal{S} = (\mathcal{M}, \mathtt{size})$ be a MATLANG schema. On the relational side we have for each size symbol $\alpha \in \mathtt{Symb} \setminus \{1\}$, attributes α, row_α, and col_α in $\mathbb{A}$. Furthermore, for each $V \in \mathcal{M}$ and $\alpha \in \mathtt{Symb}$ we denote by R_V and R_α its corresponding relation name, respectively. Then, given $\mathcal{S}$ we define the relational schema $\mathrm{Rel}(\mathcal{S})$ such that $\mathrm{dom}(\mathrm{Rel}(\mathcal{S})) = \{R_\alpha \mid \alpha \in \mathtt{Symb}\} \cup \{R_V \mid V \in \mathcal{M}\}$ where $\mathrm{Rel}(\mathcal{S})(R_\alpha) = \{\alpha\}$ and:

$$\mathrm{Rel}(\mathcal{S})(R_V) = \begin{cases} \{\mathrm{row}_\alpha, \mathrm{col}_\beta\} & \text{if } \mathtt{size}(V) = (\alpha, \beta) \\ \{\mathrm{row}_\alpha\} & \text{if } \mathtt{size}(V) = (\alpha, 1) \\ \{\mathrm{col}_\beta\} & \text{if } \mathtt{size}(V) = (1, \beta) \\ \{\} & \text{if } \mathtt{size}(V) = (1, 1). \end{cases}$$

Consider now a matrix instance $\mathcal{I} = (\mathcal{D}, \mathtt{mat})$ over $\mathcal{S}$. Let $V \in \mathcal{M}$ with $\mathtt{size}(V) = (\alpha, \beta)$ and let $\mathtt{mat}(V)$ be its corresponding K-matrix of dimension $\mathcal{D}(\alpha) \times \mathcal{D}(\beta)$. To encode $\mathcal{I}$ as a K-instance in RA^+_K, we use as data domain $\mathbb{D} = \mathbb{N} \setminus \{0\}$. Then we construct the K-instance $\mathrm{Rel}(\mathcal{I})$ such that for each $V \in \mathcal{M}$ we define $R_V^{\mathrm{Rel}(\mathcal{I})}(t) := \mathtt{mat}(V)_{ij}$ whenever $t(\mathrm{row}_\alpha) = i \leq \mathcal{D}(\alpha)$ and $t(\mathrm{col}_\beta) = j \leq \mathcal{D}(\beta)$, and $\mathbb{0}$ otherwise. Furthermore, for each $\alpha \in \mathtt{Symb}$ we define $R_\alpha^{\mathrm{Rel}(\mathcal{I})}(t) := \mathbb{1}$ whenever $t(\alpha) \leq \mathcal{D}(\alpha)$, and $\mathbb{0}$ otherwise. In other words, R_α and R_β encodes the active domain of a matrix variable V with $\mathtt{size}(V) = (\alpha, \beta)$. Given that the RA^+_K framework of [20] represents the "absence" of a tuple in the relation with $\mathbb{0}$, we need to separately encode the indexes in a matrix. This is where $R_\alpha^{\mathrm{Rel}(\mathcal{I})}$ and $R_\beta^{\mathrm{Rel}(\mathcal{I})}$ are used for. We are now ready to state the first connection between sum-MATLANG and RA^+_K by using the previous encoding. The proof of the proposition below is by induction on the structure of expressions.

Proposition 6.3. *For each* sum-MATLANG *expression e over schema $\mathcal{S}$ such that $\mathcal{S}(e) = (\alpha, \beta)$ with $\alpha \neq 1 \neq \beta$, there exists an RA^+_K expression $\Phi(e)$ over relational schema $Rel(\mathcal{S})$ such that $Rel(\mathcal{S})(\Phi(e)) = \{\mathrm{row}_\alpha, \mathrm{row}_\beta\}$ and such that for any instance $\mathcal{I}$ over $\mathcal{S}$,*

$$[\![e]\!](\mathcal{I})_{i,j} = [\![\Phi(e)]\!]_{Rel(\mathcal{I})}(t)$$

for tuple $t(\mathrm{row}_\alpha) = i$ and $t(\mathrm{col}_\beta) = j$. Similarly for when e has schema $\mathcal{S}(e) = (\alpha, 1)$, $\mathcal{S}(e) = (1, \beta)$ or $\mathcal{S}(e) = (1, 1)$, then $\Phi(e)$ has schema $Rel(\mathcal{S})(\Phi(e)) = \{\mathrm{row}_\alpha\}$, $Rel(\mathcal{S})(\Phi(e)) = \{\mathrm{col}_\alpha\}$, or $Rel(\mathcal{S})(\Phi(e)) = \{\}$, respectively.

We now move to the other direction. To translate RA^+_K into sum-MATLANG, we must restrict our comparison to RA^+_K over K-relations with at most two attributes. Given that linear algebra works over vector and matrices, it is reasonable to restrict to unary or binary relations as input. Note that this is only a restriction on the input relations and not on intermediate relations, namely, expressions can create relation signatures of arbitrary size from the binary input relations. Thus, from now we say that a relational schema $\mathcal{R}$ is binary if $|R| \leq 2$ for every $R \in \mathcal{R}$. We also make the assumption that there is an (arbitrary) order, denoted by $<$, on the attributes in $\mathbb{A}$. This is to identify which attributes correspond to rows and

columns when moving to matrices. Then, given that relations will be either unary or binary and there is an order on the attributes, we write $t = (v)$ or $t = (u, v)$ to denote a tuple over a unary or binary relation R, respectively, where u and v is the value of the first and second attribute with respect to $<$.

Consider a binary relational schema $\mathcal{R}$. With each $R \in \mathcal{R}$ we associate a matrix variable V_R such that, if R is a binary relational signature, then V_R represents a (square) matrix, and, if not (i.e. R is unary), then V_R represents a vector. Formally, fix a symbol $\alpha \in \mathsf{Symb} \setminus \{1\}$. Let $\mathrm{Mat}(\mathcal{R})$ denote the MATLANG schema $(\mathcal{M}_{\mathcal{R}}, \mathtt{size}_{\mathcal{R}})$ such that $\mathcal{M}_{\mathcal{R}} = \{V_R \mid R \in \mathcal{R}\}$ and $\mathtt{size}_{\mathcal{R}}(V_R) = (\alpha, \alpha)$ whenever $|R| = 2$, and $\mathtt{size}_{\mathcal{R}}(V_R) = (\alpha, 1)$ whenever $|R| = 1$. Take now a K-instance $\mathcal{J}$ of $\mathcal{R}$ and suppose that $\mathbf{adom}(\mathcal{J}) = \{d_1, \ldots, d_n\}$ is the active domain of $\mathcal{J}$ (the order over $\mathbf{adom}(\mathcal{J})$ is arbitrary). Then we define the matrix instance $\mathrm{Mat}(\mathcal{J}) = (\mathcal{D}_{\mathcal{J}}, \mathtt{mat}_{\mathcal{J}})$ such that $\mathcal{D}_{\mathcal{J}}(\alpha) = n$, $\mathtt{mat}_{\mathcal{J}}(V_R)_{i,j} = R^{\mathcal{J}}((d_i, d_j))$ whenever $|R| = 2$, and $\mathtt{mat}_{\mathcal{J}}(V_R)_i = R^{\mathcal{J}}((d_i))$ whenever $|R| = 1$. Note that, although each K-relation can have a different active domain, we encode them as square matrices by considering the active domain of the K-instance. By again using an inductive proof on the structure of RA^+_K expressions, we obtain the following result.

Proposition 6.4. *Let $\mathcal{R}$ be a binary relational schema. For each RA^+_K expression Q over $\mathcal{R}$ such that $|\mathcal{R}(Q)| = 2$, there exists a* sum-MATLANG *expression $\Psi(Q)$ over* MATLANG *schema $Mat(\mathcal{R})$ such that for any K-instance $\mathcal{J}$ with $\mathbf{adom}(\mathcal{J}) = \{d_1, \ldots, d_n\}$ over $\mathcal{R}$,*

$$[\![Q]\!]_{\mathcal{J}}((d_i, d_j)) = [\![\Psi(Q)]\!](Mat(\mathcal{J}))_{i,j}.$$

Similarly for when $|\mathcal{R}(Q)| = 1$, or $|\mathcal{R}(Q)| = 0$ respectively.

It is important to remark that the expression Q of the previous result can have intermediate expressions that are not necessarily binary, given that the proposition only restricts that the input relation and the schema of Q must have arity at most two. We recall from [9] that MATLANG corresponds to RA^+_K where intermediate expressions are at most ternary, and this underlies, e.g., the inability of MATLANG to check for 4-cliques. In sum-MATLANG, we can deal with intermediate relations of arbitrary arity. In fact, each new attribute can be seen to correspond to an application of the Σ operator. For example, in the 4-clique expression, four Σ operators are needed, in analogy to how 4-clique is expressed in RA^+_K.

Given the previous two propositions we derive the following conclusion which is the first characterization of relational algebra with a (sub)-fragment of linear algebra.

Corollary 6.5. sum-MATLANG *and RA^+_K over binary relational schemas are equally expressive.*

As a direct consequence, we have that sum-MATLANG cannot compute matrix inversion. Indeed, using similar arguments as in [8], i.e., by embedding RA^+_K in (infinitary) first-order logic with counting and by leveraging its locality, one can show that sum-MATLANG cannot compute the transitive closure of an adjacency matrix. By contrast, the transitive closure can be expressed by means of matrix inversion [8]. We also note that the evaluation of the Σ operator is independent of the order in which the canonical vectors are considered. This is because $\oplus$ is commutative. Hence, sum-MATLANG cannot express the order predicates mentioned in Section 3.

6.2 Hadamard product and weighted logics

Similarly to using sum, we can use other operations to update X in the for-loop. The next natural choice is to consider products of matrices. In contrast to matrix sum, we have two options: either we can choose to use matrix product or to use the pointwise matrix product, also called the Hadamard product. We treat matrix product in the next subsection and first explain here the connection of sum and Hadamard product operators to weighted logics.

For the rest of this section, fix a semiring $(K, \oplus, \odot, \mathbb{0}, \mathbb{1})$. The Hadamard product over K-matrices can be defined as the pointwise application of $\odot$ between two matrices of the same size. Formally, we define the expression $e \circ e'$ where e, e' are expressions with respect to $\mathcal{S}$ and $\mathtt{type}_{\mathcal{S}}(e) = \mathtt{type}_{\mathcal{S}}(e')$ for some schema $\mathcal{S} = (\mathcal{M}, \mathtt{size})$. Then the semantics of $e \circ e'$ is the pointwise application of $\odot$, namely, $[\![e \circ e']\!](\mathcal{I})_{ij} = [\![e]\!](\mathcal{I})_{ij} \odot [\![e']\!](\mathcal{I})_{ij}$ for any instance $\mathcal{I}$ of $\mathcal{S}$. This enables us to define, similar as for Σv, the pointwise-product quantifier $\Pi^{\circ} v$ as follows:

$$\Pi^{\circ} v.\ e := \mathtt{for}\ v, X = e_{\mathbb{1}}\ .\ X \circ e.$$

where $e_{\mathbb{1}}$ is the (easily definable) for-MATLANG expression for the matrix with the same type as X and all entries equal to the $\mathbb{1}$-element of K (i.e., we need to initialize X accordingly with the $\odot$-operator). We call sp-MATLANG the subfragment of for-MATLANG that consists of sum-MATLANG extended with $\Pi^{\circ} v$.

Example 6.6. Similar to the trace of a matrix, a useful function in linear algebra is to compute the product of the values on the diagonal. Using the $\Pi^{\circ} v$ operator, this can be easily expressed:

$$e_{\mathrm{dp}}(V) := \Pi^{\circ} v.\ v^T \cdot V \cdot v. \qquad \square$$

Clearly, the inclusion of this new operator extends the expressive power to sum-MATLANG. For example, $[\![e_{\mathrm{dp}}]\!](\mathcal{I})$ can be an exponentially large number in the dimension n of the input. By contrast, one can easily show that all expressions in sum-MATLANG can only return numbers polynomial in n. That is, sp-MATLANG is more expressive than sum-MATLANG and RA^+_K.

To measure the expressive power of sp-MATLANG, we use weighted logics [14] (WL) as a yardstick. Weighted logics extend monadic second-order logic from the boolean semiring to any semiring K. Furthermore, it has been used extensively to characterize the expressive power of weighted automata in terms of logic [15]. We use here the first-order subfragment of weighted logics to suit our purpose and, moreover, we extend its semantics over weighted structures (similar as in [19]).

A relational vocabulary Γ is a finite collection of relation symbols such that each $R \in \Gamma$ has an associated arity, denoted by $\mathrm{arity}(R)$. A K-weighted structure over Γ (or just structure) is a pair $\mathcal{A} = (A, \{R^{\mathcal{A}}\}_{R \in \Gamma})$ such that A is a non-empty finite set (i.e. the domain) and, for each $R \in \Gamma$, $R^{\mathcal{A}} : A^{\mathrm{arity}(R)} \to K$ is a function that associates to each tuple in $A^{\mathrm{arity}(R)}$ a weight in K.

Let X be a set of first-order variables. A K-weighted logic (WL) formula φ over Γ is defined by the following syntax:

$$\varphi \quad := \quad x = y \ \mid\ R(\bar{x}) \ \mid\ \varphi \oplus \varphi \ \mid\ \varphi \odot \varphi \ \mid\ \Sigma x.\varphi \ \mid\ \Pi x.\varphi$$

where $x, y \in X$, $R \in \Gamma$, and $\bar{x} = x_1, \ldots, x_k$ is a sequence of variables in X such that $k = \mathrm{arity}(R)$. As usual, we say that x is a free variable of φ, if x is not below Σx or Πx quantifiers (e.g. x is free in

$\Sigma y.R(x,y)$ but y is not). Given that K is fixed, from now on we talk about structures and formulas without mentioning K explicitly.

An assignment σ over a structure $\mathcal{A} = (A, \{R^{\mathcal{A}}\}_{R\in\Gamma})$ is a function $\sigma : X \to A$. Given $x \in X$ and $a \in A$, we denote by $\sigma[x \mapsto a]$ a new assignment such that $\sigma[x \mapsto a](y) = a$ whenever $x = y$ and $\sigma[x \mapsto a](y) = \sigma(y)$ otherwise. For $\bar{x} = x_1, \ldots, x_k$, we write $\sigma(\bar{x})$ to say $\sigma(x_1), \ldots, \sigma(x_k)$. Given a structure $\mathcal{A} = (A, \{R^{\mathcal{A}}\}_{R\in\Gamma})$ and an assignment σ, we define the semantics $[\![\varphi]\!]_{\mathcal{A}}(\sigma)$ of φ as follows:

$$
\begin{array}{ll}
\text{if } \varphi := x = y, \text{ then} & [\![\varphi]\!]_{\mathcal{A}}(\sigma) = \begin{cases} \mathbb{1} & \text{if } \sigma(x) = \sigma(y) \\ \mathbb{0} & \text{otherwise} \end{cases} \\
\text{if } \varphi := R(\bar{x}), \text{ then} & [\![\varphi]\!]_{\mathcal{A}}(\sigma) = R^{\mathcal{A}}(\sigma(\bar{x})) \\
\text{if } \varphi := \varphi_1 \oplus \varphi_2, \text{ then} & [\![\varphi]\!]_{\mathcal{A}}(\sigma) = [\![\varphi_1]\!]_{\mathcal{A}}(\sigma) \oplus [\![\varphi_2]\!]_{\mathcal{A}}(\sigma) \\
\text{if } \varphi := \varphi_1 \odot \varphi_2, \text{ then} & [\![\varphi]\!]_{\mathcal{A}}(\sigma) = [\![\varphi_1]\!]_{\mathcal{A}}(\sigma) \odot [\![\varphi_2]\!]_{\mathcal{A}}(\sigma) \\
\text{if } \varphi := \Sigma x.\, \varphi', \text{ then} & [\![\varphi]\!]_{\mathcal{A}}(\sigma) = \bigoplus_{a\in A} [\![\varphi']\!]_{\mathcal{A}}(\sigma[x \mapsto a]) \\
\text{if } \varphi := \Pi x.\, \varphi', \text{ then} & [\![\varphi]\!]_{\mathcal{A}}(\sigma) = \bigodot_{a\in A} [\![\varphi']\!]_{\mathcal{A}}(\sigma[x \mapsto a])
\end{array}
$$

When φ contains no free variables, we omit σ and write $[\![\varphi]\!]_{\mathcal{A}}$ instead of $[\![\varphi]\!]_{\mathcal{A}}(\sigma)$.

For comparing the expressive power of sp-MATLANG with WL, we have to show how to encode MATLANG instances into structures and vice versa. For this, we make two assumptions to put both languages at the same level: (1) we restrict structures to relation symbols of arity at most two and (2) we restrict instances to square matrices. The first assumption is for the same reasons as when comparing sum-MATLANG with RA^+_K, and the second assumption is to have a crisp translation between both languages. Indeed, understanding the relation of sp-MATLANG with WL for non-square matrices is slightly more complicated and we leave this for future work.

Let $\mathcal{S} = (\mathcal{M}, \mathtt{size})$ be a schema of square matrices, that is, there exists an α such that $\mathtt{size}(V) \in \{1, \alpha\} \times \{1, \alpha\}$ for every $V \in \mathcal{M}$. Define the relational vocabulary $\mathrm{WL}(\mathcal{S}) = \{R_V \mid V \in \mathcal{M}\}$ such that $\mathrm{arity}(R_V) = 2$ if $\mathtt{size}(V) = (\alpha, \alpha)$, $\mathrm{arity}(R_V) = 1$ if $\mathtt{size}(V) \in \{(\alpha, 1), (1, \alpha)\}$, and $\mathrm{arity}(R_V) = 0$ otherwise. Then given a matrix instance $\mathcal{I} = (\mathcal{D}, \mathtt{mat})$ over $\mathcal{S}$ define the structure $\mathrm{WL}(\mathcal{I}) = (\{1, \ldots, n\}, \{R^{\mathcal{I}}_V\})$ such that $\mathcal{D}(\alpha) = n$ and $R^{\mathcal{I}}_V(i, j) = \mathtt{mat}(V)_{i,j}$ if $\mathtt{size}(V) = (\alpha, \alpha)$, $R^{\mathcal{I}}_V(i) = \mathtt{mat}(V)_i$ if $\mathtt{size}(V) \in \{(\alpha, 1), (1, \alpha)\}$, and $R^{\mathcal{I}}_V = \mathtt{mat}(V)$ if $\mathtt{size}(V) = (1, 1)$.

To encode weighted structures into matrices and vectors, the story is similar as for RA^+_K. Let Γ be a relational vocabulary where $\mathrm{arity}(R) \leq 2$. Define $\mathrm{Mat}(\Gamma) = (\mathcal{M}_\Gamma, \mathtt{size}_\Gamma)$ such that $\mathcal{M}_\Gamma = \{V_R \mid R \in \Gamma\}$ and $\mathtt{size}_\Gamma(V_R)$ is equal to (α, α), $(\alpha, 1)$, or $(1, 1)$ if $\mathrm{arity}(R) = 2$, $\mathrm{arity}(R) = 1$, or $\mathrm{arity}(R) = 0$, respectively, for some $\alpha \in \mathtt{Symb}$. Similarly, let $\mathcal{A} = (A, \{R^{\mathcal{A}}\}_{R\in\Gamma})$ be a structure with $A = \{a_1, \ldots, a_n\}$, ordered arbitrarily. Then we define the matrix instance $\mathrm{Mat}(\mathcal{A}) = (\mathcal{D}, \mathtt{mat})$ such that $\mathcal{D}(\alpha) = n$, $\mathtt{mat}(V_R)_{i,j} = R^{\mathcal{A}}(a_i, a_j)$ if $\mathrm{arity}(R) = 2$, $\mathtt{mat}(V_R)_i = R^{\mathcal{A}}(a_i)$ if $\mathrm{arity}(R) = 1$, and $\mathtt{mat}(V_R) = R^{\mathcal{A}}$ otherwise.

Let $\mathcal{S}$ be a MATLANG schema of square matrices and Γ a relational vocabulary of relational symbols of arity at most 2. We can then show the equivalence of sp-MATLANG and WL as follows.

Proposition 6.7. *Weighted logics over Γ and* sp-MATLANG *over $\mathcal{S}$ have the same expressive power. More specifically,*

- *for each* sp-MATLANG *expression e over $\mathcal{S}$ such that $\mathcal{S}(e) = (1, 1)$, there exists a WL-formula $\Phi(e)$ over $\mathrm{WL}(\mathcal{S})$ such that for every instance $\mathcal{I}$ of $\mathcal{S}$, $[\![e]\!](\mathcal{I}) = [\![\Phi(e)]\!]_{\mathrm{WL}(\mathcal{I})}$.*
- *for each WL-formula φ over Γ without free variables, there exists a* sp-MATLANG *expression $\Psi(\varphi)$ such that for any structure $\mathcal{A}$ over $\mathit{Mat}(\Gamma)$, $[\![\varphi]\!]_{\mathcal{A}} = [\![\Psi(\varphi)]\!](\mathit{Mat}(\mathcal{A}))$.*

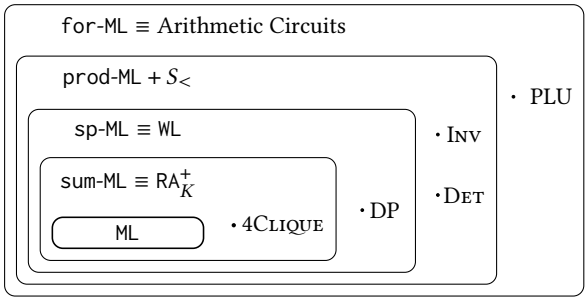

Figure 1: Fragments of for-MATLANG over square matrices and their equivalences. The functions 4Clique, DP (diagonal product), Inv, Det, and PLU decomposition are placed in their fragments.

6.3 Matrix multiplication as a quantifier

In a similar way, we can consider a fragment in which sum and the usual product of matrices can be used in for-loops. Formally, for an expression e we define the operator:

$$\Pi v.\, e = \mathtt{for}\, v, X = e_{\mathsf{Id}}.\, X \cdot e.$$

where e_{Id} is the identity matrix and $e = e(v)$. We call prod-MATLANG the subfragment of for-MATLANG that consists of sum-MATLANG extended with Πv. It is readily verified that $\Pi^\circ v$ can be expressed in terms of Πv. Furthermore, by contrast to the Hadamard product, matrix multiplication is a non-commutative operator. As a consequence, one can formulate expressions that are not invariant under the order in which the canonical vectors are processed.

Proposition 6.8. *Every expression in* sp-MATLANG *can be defined in* prod-MATLANG. *Moreover, there exists an expression that uses the Πv quantifier that cannot be defined in* sp-MATLANG.

What is interesting is that sum-MATLANG extended with Πv suffices to compute the transitive closure, provided that we allow for the $f_{>0}$ function. Indeed, one can use the expression $e_{\mathrm{TC}}(V) := f_{>0}\big(\Pi v.\, (e_{\mathsf{Id}} + V)\big)$ for this purpose because $[\![e_{\mathrm{TC}}]\!](\mathcal{I}) = f_{>0}\big((I + A)^n\big)$ when $\mathcal{I}$ assigns an $n \times n$ adjacency matrix A to V, and non-zero entries in $(I + A)^n$ coincide with non-zero entries in the transitive closure of A. Furthermore, if we extend this fragment with access to the matrix $S_<$, defining the (strict) order on canonical vectors, then Csanky's matrix inversion algorithm becomes expressible (if $f_/$ is allowed). We do not know if the inversion and determinant algorithms are expressible in sp-MATLANG. We leave the study of prod-MATLANG and, in particular, the relationship to full for-MATLANG, for future work.

Finally, in Figure 1 we show a diagram of all the fragments of for-MATLANG over square matrices introduced in this section and their corresponding equivalent formalisms.

7 CONCLUSIONS

We proposed for-MATLANG, an extension of MATLANG with limited recursion, and showed that it is able to capture most of linear algebra due to its connection to arithmetic circuits. We further revealed interesting connections to logics on annotated relations. Our focus was on language design and expressivity. An interesting direction for future work relates to efficient evaluation of (fragments) of for-MATLANG. A possible starting point is [10] in which a general methodology for communication-optimal algorithms for for-loop linear algebra programs is proposed.

Acknowledgements. Muñoz, Riveros and Vrgoč were funded by ANID - Millennium Science Initiative Program - Code ICN17_002.

REFERENCES

[1] Eric Allender. 2004. Arithmetic circuits and counting complexity classes. *Complexity of Computations and Proofs, Quaderni di Matematica* 13 (2004), 33–72.

[2] Eric Allender, Jia Jiao, Meena Mahajan, and V. Vinay. 1998. Non-Commutative Arithmetic Circuits: Depth Reduction and Size Lower Bounds. *Theor. Comput. Sci.* 209, 1-2 (1998), 47–86. https://doi.org/10.1016/S0304-3975(97)00227-2

[3] Sanjeev Arora and Boaz Barak. 2009. Complexity theory: A modern approach.

[4] Pablo Barceló, Nelson Higuera, Jorge Pérez, and Bernardo Subercaseaux. 2020. On the Expressiveness of LARA: A Unified Language for Linear and Relational Algebra. In *Proceedings of the 23rd International Conference on Database Theory (ICDT)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:20. https://doi.org/10.4230/LIPIcs.ICDT.2020.6

[5] Matthias Boehm, Arun Kumar, and Jun Yang. 2019. *Data Management in Machine Learning Systems*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00895ED1V01Y201901DTM057

[6] Allan Borodin, Joachim von zur Gathem, and John Hopcroft. 1982. Fast parallel matrix and GCD computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (SFCS)*. IEEE, 65–71. https://doi.org/10.1109/SFCS.1982.17

[7] Robert Brijder, Floris Geerts, Jan Van den Bussche, and Timmy Weerwag. 2018. On the Expressive Power of Query Languages for Matrices. In *Proceeding of the 21st International Conference on Database Theory (ICDT) (LIPIcs, Vol. 98)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:17. https://doi.org/10.4230/LIPIcs.ICDT.2018.10

[8] Robert Brijder, Floris Geerts, Jan Van den Bussche, and Timmy Weerwag. 2019. On the Expressive Power of Query Languages for Matrices. *ACM Trans. Database Syst.* 44, 4 (2019), 15:1–15:31. https://doi.org/10.1145/3331445

[9] Robert Brijder, Marc Gyssens, and Jan Van den Bussche. 2020. On Matrices and K-Relations. In *Foundations of Information and Knowledge Systems (FoIKS)*. 42–57.

[10] Michael Christ, James Demmel, Nicholas Knight, Thomas Scanlon, and Katherine A. Yelick. 2013. Communication Lower Bounds and Optimal Algorithms for Programs that Reference Arrays - Part 1. https://arxiv.org/abs/1308.0068

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms* (2nd ed.). The MIT Press.

[12] L. Csanky. 1976. Fast Parallel Matrix Inversion Algorithms. *SIAM J. Comput.* 5, 4 (1976), 618–623. https://doi.org/10.1137/0205040

[13] Anuj Dawar, Martin Grohe, Bjarki Holm, and Bastian Laubner. 2009. Logics with Rank Operators. In *Proceedings of the 24th Annual IEEE Symposium on Logic In Computer Science (LICS)*. IEEE, 113–122. https://doi.org/10.1109/LICS.2009.24

[14] Manfred Droste and Paul Gastin. 2005. Weighted Automata and Weighted Logics. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP) (Lecture Notes in Computer Science, Vol. 3580)*. 513–525. https://doi.org/10.1007/11523468_42

[15] Manfred Droste, Werner Kuich, and Heiko Vogler. 2009. *Handbook of weighted automata*. Springer Science & Business Media.

[16] Floris Geerts. 2019. On the Expressive Power of Linear Algebra on Graphs. In *Proceedings of the 22nd International Conference on Database Theory (ICDT)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:19. https://doi.org/10.4230/LIPIcs.ICDT.2019.7

[17] Floris Geerts. 2020. When Can Matrix Query Languages Discern Matrices?. In *Proceedings of the 23rd International Conference on Database Theory (ICDT) (LIPIcs, Vol. 155)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 12:1–12:18. https://doi.org/10.4230/LIPIcs.ICDT.2020.12

[18] Floris Geerts, Thomas Muñoz, Cristian Riveros, and Domagoj Vrgoč. 2020. Expressive power of linear algebra query languages. https://arxiv.org/abs/2010.13717

[19] Erich Grädel and Val Tannen. 2017. Semiring Provenance for First-Order Model Checking. http://arxiv.org/abs/1712.01980

[20] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. 31–40. https://doi.org/10.1145/1265530.1265535

[21] Martin Grohe and Wied Pakusa. 2017. Descriptive complexity of linear equation systems and applications to propositional proof complexity. In *Proceedings of the 32nd Annual IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 1–12. https://doi.org/10.1109/LICS.2017.8005081

[22] Lauri Hella, Leonid Libkin, Juha Nurmonen, and Limsoon Wong. 2001. Logics with Aggregate Operators. *J. ACM* 48, 4 (2001), 880–907. https://doi.org/10.1145/502090.502100

[23] Bjarki Holm. 2010. *Descriptive Complexity of Linear Algebra*. Ph.D. Dissertation. University of Cambridge.

[24] Dylan Hutchison, Bill Howe, and Dan Suciu. 2017. LaraDB: A Minimalist Kernel for Linear and Relational Algebra Computation. In *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond (BeyondMR)*. ACM, 2:1–2:10. https://doi.org/10.1145/3070607.3070608

[25] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J. Gao. 2019. Declarative Recursive Computation on an RDBMS. *Proc. VLDB Endow.* 12, 7 (2019), 822–835. http://www.vldb.org/pvldb/vol12/p822-jankov.pdf

[26] Erich Kaltofen. 1988. Greatest common divisors of polynomials given by straight-line programs. *J. ACM* 35, 1 (1988), 231–264. https://doi.org/10.1145/42267.45069

[27] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. AC/DC: In-Database Learning Thunderstruck. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*. ACM, 8:1–8:10. https://doi.org/10.1145/3209889.3209896

[28] Andreas Kunft, Alexander Alexandrov, Asterios Katsifodimos, and Volker Markl. 2016. Bridging the Gap: Towards Optimization Across Linear and Relational Algebra. In *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond (BeyondMR)*. 1:1–1:4. http://doi.acm.org/10.1145/2926534.2926540

[29] Shangyu Luo, Zekai J. Gao, Michael Gubanov, Luis L. Perez, and Christopher Jermaine. 2018. Scalable Linear Algebra on a Relational Database System. *SIGMOD Rec.* 47, 1 (2018), 24–31. http://doi.acm.org/10.1145/3277006.3277013.

[30] Frank Neven, Martin Otto, Jerzy Tyszkiewicz, and Jan Van den Bussche. 2001. Adding For-Loops to First-Order Logic. *Inf. Comput.* 168, 2 (2001), 156–186.

[31] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 1992. *Numerical Recipes in C, 2nd Edition*. Cambridge University Press. http://numerical.recipes

[32] Ran Raz. 2003. On the Complexity of Matrix Product. *SIAM J. Comput.* 32, 5 (2003), 1356–1369. https://doi.org/10.1137/S0097539702402147

[33] Amir Shpilka and Amir Yehudayoff. 2010. Arithmetic Circuits: A survey of recent results and open questions. *Foundations and Trends in Theoretical Computer Science* 5, 3-4 (2010), 207–388. http://dx.doi.org/10.1561/0400000039

[34] Volker Strassen. 1973. Vermeidung von Divisionen. *Journal für die reine und angewandte Mathematik* 264 (1973), 184–202. https://doi.org/10.1515/crll.1973.264.184

[35] Leslie G Valiant and Sven Skyum. 1981. Fast parallel computation of polynomials using few processors. In *Proceedings of the International Symposium on Mathematical Foundations of Computer Science (MFCS) (Lecture Notes in Computer Science, Vol. 118)*. 132–139. https://doi.org/10.1007/3-540-10856-4_79