On the Expressive Power of Query Languages for Matrices

Peer-reviewed author version

BRIJDER, Robert; GEERTS, Floris; VAN DEN BUSSCHE, Jan & WEERWAG, Timmy (2019) On the Expressive Power of Query Languages for Matrices. In: ACM TRANSACTIONS ON DATABASE SYSTEMS, 44 (4) (Art N° 15).

DOI: 10.1145/3331445
Handle: http://hdl.handle.net/1942/30378

On the Expressive Power of Query Languages for Matrices

ROBERT BRIJDER, Hasselt University, Belgium
FLORIS GEERTS, University of Antwerp, Belgium
JAN VAN DEN BUSSCHE, Hasselt University, Belgium
TIMMY WEERWAG, Hasselt University, Belgium

We investigate the expressive power of MATLANG, a formal language for matrix manipulation based on common matrix operations and linear algebra. The language can be extended with the operation inv for inverting a matrix. In MATLANG + inv we can compute the transitive closure of directed graphs, whereas we show that this is not possible without inversion. Indeed we show that the basic language can be simulated in the relational algebra with arithmetic operations, grouping, and summation. We also consider an operation eigen for diagonalizing a matrix. It is defined such that for each eigenvalue a set of mutually orthogonal eigenvectors is returned that span the eigenspace of that eigenvalue. We show that inv can be expressed in MATLANG + eigen. We put forward the open question whether there are boolean queries about matrices, or generic queries about graphs, expressible in MATLANG + eigen but not in MATLANG + inv. Finally, the evaluation problem for MATLANG + eigen is shown to be complete for the complexity class $\exists\mathbb{R}$.

CCS Concepts: • **Information systems → Query languages**; • **Theory of computation → Database query languages (principles)**; • **Computing methodologies →** *Linear algebra algorithms.*

Additional Key Words and Phrases: matrix query languages, relational algebra with aggregates, query evaluation problem, graph queries

ACM Reference Format:
Robert Brijder, Floris Geerts, Jan Van den Bussche, and Timmy Weerwag. 2020. On the Expressive Power of Query Languages for Matrices. 1, 1 (January 2020), 30 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In view of the importance of large-scale statistical and machine learning (ML) algorithms in the overall data analytics workflow, database systems are in the process of being redesigned and extended. The aim is to allow for a seamless integration of ML algorithms and mathematical and statistical frameworks, such as R, SAS, and MATLAB, with existing data manipulation and data querying functionality [7, 12, 15, 31, 33, 39, 44, 48, 49, 52, 60, 69]. In particular, data scientists often use *matrices* to represent their data, as opposed to using the relational data model, and create custom data analytics algorithms using *linear algebra*, instead of writing SQL queries. Here, linear algebra algorithms are expressed in a declarative manner by composing basic linear algebra constructs. Examples of such constructs are: matrix multiplication, matrix transposition, element-wise operations on the entries of matrices, solving nonsingular systems of linear equations (matrix inversion), diagonalization (eigenvalues and eigenvectors), singular value decomposition, just to name a few. The main challenges from a database system's perspective are to ensure scalability.

Authors' addresses: Robert Brijder, Hasselt University, Hasselt, Belgium; Floris Geerts, University of Antwerp, Antwerp, Belgium; Jan Van den Bussche, Hasselt University, Hasselt, Belgium; Timmy Weerwag, Hasselt University, Hasselt, Belgium.

XXXX-XXXX/2020/1-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

We identify two general approaches in this direction: (i) to provide physical data independence and (ii) to provide optimizations. The former is to relieve users from the manual handling of data distribution, communication, fault tolerance, among other things. The second is to compile linear algebra algorithms into efficient programs hereby mimicking cost-based query optimization used to evaluate SQL queries. We refer to [62] for an overview of the different systems addressing these challenges.

In this context, the following natural questions arise: Which linear algebra constructs need to be supported to perform specific data analytical tasks? Does the additional support for certain linear algebra operations increase the overall functionality? When are two linear algebra algorithms equivalent (perform the same task)? Such questions have been extensively studied for classical query languages (fragments and extensions of SQL) in database theory and finite model theory [1, 38]. Indeed, the questions raised all relate to the *expressive power* of query languages. In this paper we enrol in the investigation of the expressive power of *matrix query languages*.

As a starting point we focus on matrices and matrix query languages alone, leaving the study of the expressive power of languages that operate on *both* relational data and matrices for future work. Even this "matrix only" setting turns out to be quite interesting and challenging on its own.

To set the stage, we need to formally define what we mean by a matrix query language. There has been work in finite model theory and logic to understand the capability of certain logics to express linear algebra operations [18–20, 26, 29, 32]. In particular, the extent to which fixpoint logics with counting and their extension with so-called rank operators can express linear algebra has been considered. The motivation for that line of work is mainly to find a logical characterization of polynomial-time computability and less so in understanding the expressive power of specific linear algebra operations.

In this paper, we take the opposite approach in which we define a basic matrix query language, referred to as MATLANG, which is built up from *basic* linear algebra operations, supported by linear algebra systems such as R and MATLAB, and then closing these operations under *composition*. Throughout the paper we consider matrices with entries in the complex field $\mathbf{C}$, unless specified otherwise. Let us have a sneak preview of MATLANG.

Example 1.1 (Google matrix). Let A be the adjacency matrix of a directed graph (modeling the Web graph) on n nodes numbered $1, \ldots, n$. Let $0 < d < 1$ be a fixed "damping factor". Let k_i denote the outdegree of node i. For simplicity, we assume k_i to be nonzero for every i. Then the Google matrix [8, 11] of A is the $n \times n$ matrix G defined by

$$G_{i,j} = d\frac{A_{ij}}{k_i} + \frac{1-d}{n}.$$

To perform the calculation of G from A we can formulate the following MATLANG expression:

$$\mathsf{apply}[+]\Big(d\odot\mathsf{apply}[/]\Big(X, X\cdot\mathbf{1}(X)\cdot(\mathbf{1}(X))^*\Big), (1-d)\odot\Big(\mathsf{apply}[1/x]\big((\mathbf{1}(X))^*\cdot\mathbf{1}(X)\big)\Big)\odot\Big(\mathbf{1}(X)\cdot(\mathbf{1}(X))^*\Big)\Big).$$

Let us unfold this expression to understand its meaning. The basic operations in MATLANG used in this expression are: (i) a *matrix variable*, denoted by X, which is to be instantiated with the input matrix A; (ii) *matrix multiplication*, denoted by "$\cdot$"; (iii) *matrix transposition*, denoted by "$*$"; (iv) the *one-vector*, denoted by $\mathbf{1}(\cdot)$, returning the column vector with each entry equal to "1" and with dimension equal to the number of rows of the input matrix; and (v) *pointwise function applications*, denoted by $\mathsf{apply}[f](\cdot)$, whose semantics will be explained below. In the expression we also find the operation "$\odot$". This is a shorthand notation for *scalar multiplication*. As we will see later in the paper, it can be expressed in terms of the basic operations in MATLANG.

Given this, the sub-expression $\mathbf{1}(X) \cdot (\mathbf{1}(X))^*$ will evaluate, when X is assigned to A, to the $n \times n$ matrix J in which every entry equals to one. Similarly, $(\mathbf{1}(X))^* \cdot \mathbf{1}(X)$, when X is assigned to A, returns the dimension n of A. Furthermore, the result of $\mathrm{apply}[1/x]\big((\mathbf{1}(A))^* \cdot \mathbf{1}(A)\big)$ is obtained by applying the function $x \mapsto 1/x$ to every (non-zero) element in its input, in this case only to the value n, resulting in $1/n$. The second term in the Google matrix G is thus obtained by multiplying J, as previously computed, by $1/n$ and $1 - d$ using scalar multiplication $\odot$.

We next consider the first term of G. The sub-expression $\mathrm{apply}[/]\Big(X, X \cdot \mathbf{1}(X) \cdot (\mathbf{1}(X))^*\Big)$ evaluates, when A is assigned to X, to the $n \times n$ matrix B which holds A_{ij}/k_i in entry (i, j). Indeed, $A \cdot \mathbf{1}(A) \cdot (\mathbf{1}(A))^* = A \cdot J$ consists of the $n \times n$-matrix K in which the ith row consists solely of the number k_i. The pointwise function application has now two arguments, X and $X \cdot \mathbf{1}(X) \cdot (\mathbf{1}(X))^*$. For every entry in these two inputs (A_{ij} and $K_{ij} = k_i$) it applies the function $(x, y) \mapsto x/y$. This results in the matrix B. Finally, a scalar multiplication by d provides the first term in G. It remains to sum up both terms to obtain G. This is done by a final pointwise function application mapping each of its two input entries to the sum of those entries, using the function $(x, y) \mapsto x + y$. ◻

In the previous example we actually used almost all basic operations (matrix variables, matrix multiplication, transpose, one-vector, function applications) in MATLANG. Missing here is the *diagonalization operation* (diag(·)) turning a column vector into a diagonal matrix. All six basic linear algebra operations supported in MATLANG stem from “atomic” operations supported in popular linear algebra packages. While many other operations are supported by these packages, we feel that they are somewhat less atomic. We present more examples later on, showing that MATLANG is indeed capable of expressing common matrix manipulations. In fact, we propose MATLANG as *an analog for matrices of the relational algebra for relations*. With MATLANG as the starting point, what can we say about its expressive power?

To answer this question, we relate MATLANG to the relational algebra with aggregates [37, 43], using a standard representation of matrices as relations. The only aggregate function that is needed is summation. In fact, it turns out that MATLANG is already subsumed by aggregate logic with only *three* nonnumerical variables. Conversely, MATLANG can express all queries from graph databases (binary relational structures) to binary relations that can be expressed in first-order logic with three variables. In contrast, the four-variable query asking if the graph contains a *four-clique*, is not expressible. We note that the connection with three-variable logics has recently been strengthened [23]. Indeed, it has been shown that two undirected graphs are indistinguishable by means of sentences in MATLANG if and only if they are indistinguishable by means of sentences in the three-variable fragment of first-order logic with counting. A MATLANG sentence here refers to an expression that always returns single (complex) numbers. We observe that as a direct consequence from the locality of relational algebra with aggregates [43], it follows that the *transitive closure* of graph is also not expressible in MATLANG given its adjacency matrix.

We thus see that, for example, when data analysts want to check for four-cliques in a graph, more advanced linear algebra operations than those in MATLANG need to be considered when building scalable linear algebra systems. Similarly, extracting information related to the connectivity of graphs requires extending MATLANG. We consider two such extensions in the paper:

- MATLANG + inv: The extension of MATLANG with an operation (inv) for *inverting* a matrix. We show that MATLANG + inv is strictly more expressive than MATLANG. Indeed, the transitive closure of binary relations becomes expressible. The possibility of reducing transitive closure to matrix inversion has been pointed out by several researchers [16, 17, 41, 57]. We show that the restricted setting of MATLANG suffices for this reduction to work.
- MATLANG + eigen: The extension of MATLANG with an operation (eigen) which returns *eigenvectors* and *eigenvalues*. There are various ways to define this operation formally. Since

> no unique set of eigenvectors exists, the eigen operation is intrinsically *non-deterministic*. We show that the resulting language MATLANG + eigen can express inversion and this by using a deterministic MATLANG + eigen expression (i.e., despite it using eigen, it always deterministically returns the inverse of a matrix, if it exists). The argument is well known from linear algebra, but our result shows that starting from the eigenvalues and eigenvectors, MATLANG is expressive enough to construct the inverse. It once more attests that we have defined an adequate matrix language for performing common matrix manipulations.

It is natural to conjecture that MATLANG+eigen is actually strictly more powerful than MATLANG+inv in expressing, say, boolean queries about matrices. Proving this is an interesting open problem.

We conclude the introduction by going back to our earlier question regarding the equivalence of linear algebra algorithms. Here, one would like to know, at the very least, whether the *equivalence* of linear algebra expressions is decidable. We answer this question affirmatively for expressions in our most expressive matrix query language MATLANG + eigen. Related to this is the question whether the *evaluation* of expressions in MATLANG + eigen is effectively computable. This may seem like an odd question, since linear algebra computations are done in practice. These evaluation algorithms, however, often use techniques from numerical mathematics [25], resulting in approximations of the precise result. We are interested in the exact result.

We show that the input-output relation of an expression e in MATLANG + eigen, applied to input matrices of given dimensions, is definable in the *existential theory of the real numbers*, by a formula of size polynomial in the size of e and the given dimensions. Here, we encode complex numbers in input matrices by pairs of real numbers. The existential theory of the reals is decidable; actually, the full first-order theory of the reals is decidable [3, 5]. More specifically, the class of problems that can be reduced in polynomial time to the existential theory of the reals forms a complexity class on its own, known as $\exists\mathbf{R}$ [58, 59]. To situate $\exists\mathbf{R}$ among classical complexity classes: It is known to contain NP (this follows easily from the definition of $\exists\mathbf{R}$) and is contained in PSPACE [14]. We thus place natural decision versions of the evaluation problem for MATLANG + eigen in the complexity class $\exists\mathbf{R}$ (combined complexity). We obviously restrict ourselves in this setting to pointwise function applications that are definable in the existential theory of the real numbers.

We show, moreover, that there exists a fixed expression (data complexity) in MATLANG + eigen for which the evaluation problem is $\exists\mathbf{R}$-complete, even when restricted to input matrices with integer entries. We remark that the $\exists\mathbf{R}$-hardness proof heavily relies on the non-deterministic character of the eigen operation. The precise complexity of the evaluation problem for deterministic MATLANG + eigen expressions is left open.

Organization of the paper. We discuss related work in Section 2 and introduce the syntax, semantics and type-checking system for MATLANG in Section 3. The expressive power of MATLANG is considered in Section 4, followed by an investigation of the extensions MATLANG + inv, in Section 5, and of MATLANG + eigen, in Section 6. The evaluation problem for expressions in MATLANG + eigen is treated in Section 7. We compare the efficiency of computing the transitive closure of graphs by means of specialized algorithms with the evaluation of the corresponding MATLANG + inv expression in Section 8. Finally, in Section 9 we conclude the paper.

2 RELATED WORK

Programming languages to manipulate matrices trace back to the APL language [34]. Providing database support for matrices and multidimensional arrays has been a long-standing research topic [55], originally geared towards applications in scientific data management.

In [39], Lara is proposed as a domain-specific programming language written in Scala that provides both linear algebra (LA) and relational algebra (RA) constructs. This is done by introducing three core types corresponding to bags, matrices, and vectors with various operations for each type. This approach is taken one step further in [33] where it is shown that the RA operations and a number of LA operations can be defined in terms of three core operations called Ext, Union, and Join. The resulting language (although different from Lara of [39]) is also called Lara. Using these three core operations, the RA operations and some LA operations can be combined in a single language that can be implemented efficiently. Similarly to what we show in this paper for the language MATLANG, it seems that the expressive power of the language formed by Ext, Union, and Join is subsumed by the relational algebra with aggregates.

Another relevant related work is the FAQ framework [2], which focuses on the project-join fragment of the algebra for K-relations [28] (relations where the tuples are annotated with elements from some commutative semiring K). The connection between MATLANG and the algebra for K-relations is more deeply investigated in a forthcoming paper [10]. Yet another related formalism is that of logics with rank operators [18, 19, 26, 29, 32, 54]. These operators solve 0, 1-matrices over finite fields, and increase the expressive power of established logics over abstract structures. In contrast, in this paper we are interested in queries on arbitrary matrices.

Modest changes to SQL in order to perform LA operations in a scalable way within relational databases are proposed in [45]. In this way, various linear algebra operations are implemented in an efficient way using the relational algebra. The exact scope of the linear algebra operations that can be implemented in this way remains to be formally understood. More generally, various systems are being developed in which relational and linear algebra functionalities are combined [7, 12, 15, 31, 33, 39, 44, 48, 49, 52, 60, 69].

In this vein, we investigate in this paper the expressive power of common linear algebra operations, and we relate MATLANG to the relational algebra. While the previous work is focused on showing that relational algebra (appropriately extended) can serve as a platform for supporting large scale linear algebra operations, the focus of our work here is complementary. Indeed, we want to understand the precise expressive power of common linear algebra operations, as adequately formalized in the language MATLANG and its extensions. In particular, we compare the expressive power of matrix queries to that of relational queries.

A conference version of this paper was presented at ICDT 2018 [9]. In this journal version, we provide detailed proofs of all results and report on some preliminary experiments in which we investigate the efficiency of computing the transitive closure of graph using linear algebra operators.

3 MATLANG

We start by defining the language MATLANG in Section 3.1, provide its semantics in Section 3.2, and conclude by describing a type-checking system for MATLANG expressions in Section 3.3.

3.1 Syntax of MATLANG expressions

We assume a sufficient supply of *matrix variables*, which serve to indicate the inputs to expressions in MATLANG. The syntax of MATLANG expressions is defined by the grammar:

$$\begin{array}{rll} e ::= & M & \text{(matrix variable)} \\ | & \mathsf{let}\ M = e_1\ \mathsf{in}\ e_2 & \text{(local binding)} \\ | & e^* & \text{(conjugate transpose)} \\ | & \mathbf{1}(e) & \text{(one-vector)} \\ | & \mathrm{diag}(e) & \text{(diagonalization of a vector)} \\ | & e_1 \cdot e_2 & \text{(matrix multiplication)} \end{array}$$

$$\begin{bmatrix} 0 & 1+i \\ 2 & 3-i \\ 4+4i & 5 \end{bmatrix}^* = \begin{bmatrix} 0 & 2 & 4-4i \\ 1-i & 3+i & 5 \end{bmatrix} \qquad \mathbf{1}\left(\begin{bmatrix} 2 & \sqrt{3} & 4 \\ 4 & 5 & 6 \end{bmatrix}\right) = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & -3 \\ 2 & 7 \\ \frac{2}{3} & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 3 & -1 & 3-i \\ 5 & -2 & 1 & 0 \end{bmatrix} = \begin{bmatrix} -15 & -6 & -3 & 0 \\ 37 & -8 & 5 & 6-2i \\ 5\frac{2}{3} & 0 & \frac{1}{3} & 2-\frac{2}{3}i \end{bmatrix} \qquad \mathrm{diag}\left(\begin{bmatrix} 6 \\ 7 \end{bmatrix}\right) = \begin{bmatrix} 6 & 0 \\ 0 & 7 \end{bmatrix}$$

$$\mathrm{apply}[\dot{-}]\left(\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}\right) = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Fig. 1. Basic matrix operations of MATLANG.

$$\mid \quad \mathrm{apply}[f](e_1, \dots, e_n) \qquad\qquad \text{(pointwise application, } f \in \Omega\text{)}$$

In the last rule, f is the name of a function $f : \mathbf{C}^n \to \mathbf{C}$, where $\mathbf{C}$ denotes the complex numbers. Formally, the syntax of MATLANG is parameterized by a repertoire Ω of such functions, but for simplicity we will not reflect this in the notation. We will see various examples of MATLANG expressions below.

Remark. As can be seen in the grammar, variables can also be introduced in let-constructs inside expressions as a way to give names to intermediate results. This makes it easier to write expressions. When considering MATLANG and its extension MATLANG + inv (to be defined in Section 5), the let-construct is not an essential operation and can be easily eliminated from expressions. We will see later, however, that it plays an important role when considering MATLANG + eigen (see Section 6).

3.2 Semantics of MATLANG expressions

In defining the semantics of the language, we begin by defining the basic matrix operations. Following practical matrix sublanguages such as those of R or MATLAB, we will work throughout with matrices over the complex numbers. However, a real-number version of the language could be defined as well. The semantics of the different operations is:

Transpose: If A is a matrix then A^* is its conjugate transpose. So, if A is an $m \times n$ matrix then A^* is an $n \times m$ matrix and the entry $A^*_{i,j}$ is the complex conjugate of the entry $A_{j,i}$.

One-vector: If A is an $m \times n$ matrix then $\mathbf{1}(A)$ is the $m \times 1$ column vector consisting of all ones.

Diag: If v is an $m \times 1$ column vector then $\mathrm{diag}(v)$ is the $m \times m$ diagonal square matrix with v on the diagonal and zero everywhere else.

Matrix multiplication: If A is an $m \times n$ matrix and B is an $n \times p$ matrix then the well known matrix multiplication AB is defined to be the $m \times p$ matrix where $(AB)_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}$. In MATLANG we explicitly denote this as $A \cdot B$.

Pointwise application: If $A^{(1)}, \dots, A^{(n)}$ are matrices of the same dimensions $m \times p$, then $\mathrm{apply}[f](A^{(1)}, \dots, A^{(n)})$ is the $m \times p$ matrix C where $C_{i,j} = f(A^{(1)}_{i,j}, \dots, A^{(n)}_{i,j})$.

Example 3.1. The operations are illustrated in Figure 1. In the pointwise application example, we use the function $\dot{-}$ defined by $x \dot{-} y = x - y$ if x and y are both real numbers and $x \geq y$, and $x \dot{-} y = 0$ otherwise. □

The formal semantics of MATLANG expressions is defined in a straightforward manner, as shown in Figure 2. Expressions will be evaluated over instances where an *instance* I is a function, defined

$$\frac{M \in \mathrm{var}(I)}{M(I) = I(M)} \qquad \frac{e_1(I) = A \qquad e_2(I[M := A]) = B}{(\text{let } M = e_1 \text{ in } e_2)(I) = B} \qquad \frac{e(I) = A}{e^*(I) = A^*} \qquad \frac{e(I) = A}{\mathbf{1}(e)(I) = \mathbf{1}(A)}$$

$$\frac{e(I) = A \qquad A \text{ is a column vector}}{\mathrm{diag}(e)(I) = \mathrm{diag}(A)}$$

$$\frac{e_1(I) = A \qquad e_2(I) = B \qquad \text{number of columns of } A \text{ equals the number of rows of } B}{e_1 \cdot e_2(I) = AB}$$

$$\frac{\forall k = 1, \ldots, n : (e_k(I) = A_k) \qquad \text{all } A_k \text{ have the same dimensions}}{\mathrm{apply}[f](e_1, \ldots, e_n)(I) = \mathrm{apply}[f](A_1, \ldots, A_n)}$$

Fig. 2. Big-step operational semantics of MATLANG. The notation $I[M := A]$ denotes the instance that is equal to I, except that M is mapped to the matrix A.

on a nonempty finite set $\mathrm{var}(I)$ of matrix variables, that assigns a matrix to each element of $\mathrm{var}(I)$. Figure 2 provides the rules that allow to derive that an expression e, on an instance I, *successfully evaluates* to a matrix A. We denote this success by $e(I) = A$. The reason why an evaluation may not succeed can be found in the rules that have a condition attached to them. The rule for variables fails when an instance simply does not provide a value for some input variable. The rules for diag, apply, and matrix multiplication have conditions on the dimensions of matrices, that need to be satisfied for the operations to be well-defined.

Example 3.2 (Scalars). As a first example we show how to express scalars (elements in $\mathbf{C}$). Obviously, in practice, scalars would be part of the language. In this paper, however, we are interested in expressiveness, so we start from a minimal language (MATLANG) and then see what is already expressible in this language. To express a scalar $c \in \mathbf{C}$, consider the constant function $c : \mathbf{C} \to \mathbf{C} : z \mapsto c$ and the MATLANG expression defined as

$$\text{let } N = \mathbf{1}(M)^* \text{ in } \mathrm{apply}[c]\big(\mathbf{1}(N)\big).$$

We overload notation a bit and also denote this expression by c. Regardless of the matrix assigned to M, the expression c evaluates to the 1×1 matrix whose single entry equals the scalar c. We remark that the expression c is actually equivalent to $\mathrm{apply}[c]\big(\mathbf{1}(\mathbf{1}(M)^*)\big)$ in which we eliminated the let-construct by plugging in the definition of $N = \mathbf{1}(M)^*$ into $\mathrm{apply}[c]\big(\mathbf{1}(N)\big)$. Let-constructs can always be eliminated from MATLANG expressions in this way. □

Example 3.3 (Scalar multiplication). We can also express scalar multiplication of a matrix by a scalar, i.e., the operation which multiplies every entry of a matrix by the same scalar. Indeed, let c be a scalar and consider the MATLANG expression

$$\text{let } O = \mathbf{1}(M) \cdot c(M) \cdot (\mathbf{1}(M^*))^* \text{ in } \mathrm{apply}[\times](O, M),$$

where c is the scalar expression from the previous example. If M is assigned an $m \times n$ matrix A, then $c(A)$ returns the 1×1 matrix $[c]$ and in variable O we compute the $m \times n$ matrix where every entry equals c. Then pointwise multiplication $\times$ which returns xy on input (x, y) is used to do the scalar multiplication of A by c. This example generalizes in a straightforward manner to

$$\mathrm{apply}[\times]\big(\mathbf{1}(e_2) \cdot e_1 \cdot (\mathbf{1}(e_2^*))^*, e_2\big),$$

where e_1 and e_2 are MATLANG expressions such that $e_1(I)$ is a 1×1-matrix for any instance I. It should be clear that this expression evaluates to the scalar multiplication of $e_2(I)$ by $e_1(I)$ for any I. We use $e_1 \odot e_2$ as a shorthand notation for this expression. For example, $c \odot e_2$ represents the scalar multiplication of e_2 by the scalar c. □

Example 3.4 (Google matrix). We have already seen a MATLANG expression for computing the Google matrix in Example 1.1. The previous example shows that the scalar multiplication $\odot$ with $1/n$ and constants $1/n$, d and $1-d$ used in that expression is indeed expressible in MATLANG. □

Example 3.5 (Diag on matrices). In MATLANG we only defined the operation diag on column vectors. Linear algebra packages also allow the application of diag on square matrices. More specifically, $\mathsf{diag}(A)$ for an $n \times n$ matrix A is defined as the column vector holding the diagonal entries of A in its entries. We can easily express this in MATLANG, as follows:

$$\Big(\mathsf{apply}[\times]\big(M, \mathsf{diag}(\mathbf{1}(M))\big)\Big) \cdot \mathbf{1}(M).$$

Indeed, in this expression we first perform pointwise multiplication of the input matrix with the identity matrix to extract the entries on the diagonal, followed by the multiplication with the one vector to return the desired column vector. □

Example 3.6 (Minimum of a vector). A less obvious example is the following. Let $v = (v_1, \dots, v_n)^*$ be a column vector of real numbers; we would like to extract the minimum from v. This can be done as follows:

$$\begin{aligned}
&\mathsf{let}\ V = v \cdot \mathbf{1}(v)^*\ \mathsf{in}\\
&\quad \mathsf{let}\ C = \big(\mathsf{apply}[\leq](V, V^*)\big) \cdot \mathbf{1}(v)\ \mathsf{in}\\
&\qquad \mathsf{let}\ N = \mathbf{1}(v)^* \cdot \mathbf{1}(v)\ \mathsf{in}\\
&\qquad\quad \mathsf{let}\ S = \mathsf{apply}[=](C, \mathbf{1}(v) \cdot N)\ \mathsf{in}\\
&\qquad\qquad \mathsf{let}\ M = \mathsf{apply}[1/x](S^* \cdot \mathbf{1}(v))\ \mathsf{in}\ M \cdot v^* \cdot S
\end{aligned}$$

The pointwise functions applied are $\leq$, which returns 1 on (x, y) if $x \leq y$ and 0 otherwise; $=$, defined analogously; and the reciprocal function. The expression works as follows. In variable V we compute a square matrix holding n copies of v. Then in variable C we compute the $n \times 1$ column vector where C_i counts the number of v_j such that $v_i \leq v_j$. If $C_i = n$ then v_i equals the minimum. Variable N computes the scalar n and column vector S is a selector where $S_i = 1$ if v_i equals the minimum, and $S_i = 0$ otherwise. Since the minimum may appear multiple times in v, we compute in M the inverse of its multiplicity. Finally we sum the different occurrences of the minimum in v and divide by the multiplicity. □

The naive evaluation of the MATLANG expression in Example 3.6 yields a quadratic time algorithm, whereas the minimum can clearly be computed in linear time. An analogous situation occurs in SQL, where an explicit `MIN` function is present to avoid this problem. It is an interesting problem to formally prove that the minimum of a set of ordered elements is not expressible in the relational algebra with order comparisons, without generating an intermediate result of quadratic size.

3.3 Types and schemas

We now introduce a notion of schema, which assigns types to matrix names, so that expressions can be type-checked against schemas. We already remarked the need for this. Indeed, due to conditions on the dimensions of matrices, MATLANG expressions are not well-defined on all instances. For example, if I is an instance where $I(M)$ is a 3×4 matrix and $I(N)$ is a 2×4 matrix, then the expression $M \cdot N$ is not defined on I. The expression $M \cdot N^*$, however, is well-defined on I.

$$\frac{M \in \mathrm{var}(\mathcal{S})}{\mathcal{S} \vdash M : \mathcal{S}(M)} \qquad \frac{\mathcal{S} \vdash e_1 : \tau_1 \qquad \mathcal{S}[M := \tau_1] \vdash e_2 : \tau_2}{\mathcal{S} \vdash \mathsf{let}\ M = e_1\ \mathsf{in}\ e_2 : \tau_2} \qquad \frac{\mathcal{S} \vdash e : s_1 \times s_2}{\mathcal{S} \vdash e^* : s_2 \times s_1} \qquad \frac{\mathcal{S} \vdash e : s_1 \times s_2}{\mathcal{S} \vdash \mathbf{1}(e) : s_1 \times 1}$$

$$\frac{\mathcal{S} \vdash e : s \times 1}{\mathcal{S} \vdash \mathsf{diag}(e) : s \times s} \qquad \frac{\mathcal{S} \vdash e_1 : s_1 \times s_2 \qquad \mathcal{S} \vdash e_2 : s_2 \times s_3}{\mathcal{S} \vdash e_1 \cdot e_2 : s_1 \times s_3}$$

$$\frac{n > 0 \qquad f : \mathbb{C}^n \to \mathbb{C} \qquad \forall k = 1, \dots, n : (\mathcal{S} \vdash e_k : \tau)}{\mathcal{S} \vdash \mathsf{apply}[f](e_1, \dots, e_n) : \tau}$$

Fig. 3. Type-checking MATLANG. The notation $\mathcal{S}[M := \tau]$ denotes the schema that is equal to $\mathcal{S}$, except that M is mapped to the type τ.

Our types need to be able to guarantee equalities between numbers of rows or numbers of columns, so that apply and matrix multiplication can be type-checked. Our types also need to be able to recognize vectors, so that diag can be type-checked.

Formally, we assume a sufficient supply of *size symbols*, which we will denote by the letters α, β, γ. A size symbol represents the number of rows or columns of a matrix. Together with an explicit 1, we can indicate arbitrary matrices as $\alpha \times \beta$, square matrices as $\alpha \times \alpha$, column vectors as $\alpha \times 1$, row vectors as $1 \times \alpha$, and scalars as 1×1. Formally, a *size term* is either a size symbol or an explicit 1. A *type* is then an expression of the form $s_1 \times s_2$ where s_1 and s_2 are size terms. Finally, a *schema* $\mathcal{S}$ is a function, defined on a nonempty finite set $\mathrm{var}(\mathcal{S})$ of matrix variables, that assigns a type to each element of $\mathrm{var}(\mathcal{S})$.

The type-checking rules for expressions are shown in Figure 3. The figure provides the rules that allow to infer an output type τ for an expression e over a schema $\mathcal{S}$. To indicate that a type can be *successfully inferred*, we use the notation $\mathcal{S} \vdash e : \tau$. When we cannot infer a type, we say e is not well-typed over $\mathcal{S}$. For example, when $\mathcal{S}(M) = \alpha \times \beta$ and $\mathcal{S}(N) = \gamma \times \beta$, then the expression $M \cdot N$ is not well-typed over $\mathcal{S}$. The expression $M \cdot N^*$, however, is well-typed with output type $\alpha \times \gamma$.

To establish the soundness of the type system, we need a notion of conformance of an instance to a schema.

Formally, a *size assignment* σ is a function from size symbols to positive natural numbers. We extend σ to any size term by setting $\sigma(1) = 1$. Now, let $\mathcal{S}$ be a schema and I an instance with $\mathrm{var}(I) = \mathrm{var}(\mathcal{S})$. We say that I is an *instance* of $\mathcal{S}$ if there is a size assignment σ such that for all $M \in \mathrm{var}(\mathcal{S})$, if $\mathcal{S}(M) = s_1 \times s_2$, then $I(M)$ is a $\sigma(s_1) \times \sigma(s_2)$ matrix. In that case we also say that I *conforms* to $\mathcal{S}$ by the size assignment σ.

We now obtain the following obvious but desirable property.

Proposition 3.7 (Safety). *If $\mathcal{S} \vdash e : s_1 \times s_2$, then for every instance I conforming to $\mathcal{S}$, by size assignment σ, the matrix $e(I)$ is well-defined and has dimensions $\sigma(s_1) \times \sigma(s_2)$.* □

It is clear from the semantics and also from the type-checking rules that MATLANG operations can only produce matrices with dimensions coming from the input matrices. Consequently, certain operations supported by linear algebra packages such as the direct sum, the Kronecker product, or tensor product fall outside the scope of our current formalism.

4 EXPRESSIVE POWER OF MATLANG

In this section we relate MATLANG to standard relational query languages. In particular, we show that MATLANG can be simulated in the relational algebra with aggregates (Section 4.2) and the relational calculus with aggregates in which only three base variables are needed (Section 4.3). This provides an easy way to implement MATLANG on top of a relational database, although specific optimizations will still be required to make this scalable [45]. Our main interest in this paper, however, is to use these translations to show the limitations of MATLANG. In particular, we use the locality of these relational languages to show that the transitive closure of an adjacency matrix cannot be expressed in MATLANG and similarly, we use the simulation of MATLANG in the relational calculus with aggregates to show that the existence of a four-clique cannot be detected in MATLANG (Section 4.4).

4.1 Relational representation of matrices

We start by fixing our representation of matrices as relations. It is natural to represent an $m \times n$ matrix A by a ternary relation

$$Rel_2(A) := \{(i, j, A_{i,j}) \mid i \in \{1, \dots, m\},\ j \in \{1, \dots, n\}\}.$$

In the special case where A is an $m \times 1$ matrix (column vector), A can also be represented by a binary relation $Rel_1(A) := \{(i, A_{i,1}) \mid i \in \{1, \dots, m\}\}$. Similarly, a $1 \times n$ matrix (row vector) A can be represented by $Rel_1(A) := \{(j, A_{1,j}) \mid j \in \{1, \dots, n\}\}$. Finally, a 1×1 matrix (scalar) A can be represented by the unary singleton relation $Rel_0(A) := \{(A_{1,1})\}$. We remark that the relation representation alone does not distinguish between row and column vectors. When carrying out the translation of MATLANG into the relational algebra with aggregates below, we always know, however, whether we are dealing with a row or column vector based on the types of the MATLANG expressions involved. We then manipulate the relations $Rel_1(A)$ accordingly.

Note that in MATLANG, we perform calculations on matrix entries, but not on row or column indices. This fits well to the relational model with aggregates as formalized by Libkin [43]. In this model, the columns of relations are typed as “base”, indicated by **b**, or “numerical”, indicated by **n**. In the relational representations of matrices presented above, the last column is of type **n** and the other columns (if any) are of type **b**. In particular, in our setting, numerical columns hold complex numbers. We now rephrase our relational encoding more formally in this setting.

More formally, we assume a supply of *relation variables*, which, for convenience, we can take to be the same as the matrix variables. A *relation type* is a tuple of **b**’s and **n**’s. A *relational schema* $\mathcal{S}$ is a function, defined on a nonempty finite set $\text{var}(\mathcal{S})$ of relation variables, that assigns a relation type to each element of $\text{var}(\mathcal{S})$.

To define relational instances, we assume a countably infinite universe **dom** of abstract atomic data elements. It is convenient to assume that the natural numbers are contained in **dom**. We stress that this assumption is not essential but simplifies the presentation. Alternatively, we would have to work with explicit embeddings from the natural numbers into **dom**.

Let τ be a relation type. A *tuple of type* τ is a tuple $(t(1), \dots, t(n))$ of the same arity as τ, such that $t(i) \in \mathbf{dom}$ when $\tau(i) = \mathbf{b}$, and $t(i)$ is a complex number when $\tau(i) = \mathbf{n}$. A *relation of type* τ is a finite set of tuples of type τ. An *instance* of a relational schema $\mathcal{S}$ is a function I defined on $\text{var}(\mathcal{S})$ so that $I(R)$ is a relation of type $\mathcal{S}(R)$ for every $R \in \text{var}(\mathcal{S})$.

The matrix data model can now be formally connected to the relational data model, as follows. Let $\tau = s_1 \times s_2$ be a matrix type. Let us call τ a *general type* if s_1 and s_2 are both size symbols; a *vector type* if s_1 is a size symbol and s_2 is 1, or vice versa; and the *scalar type* if τ is 1×1. To every matrix type τ we associate a relation type

$$Rel(\tau) := \begin{cases} (\mathbf{b}, \mathbf{b}, \mathbf{n}) & \text{if } \tau \text{ is general;} \\ (\mathbf{b}, \mathbf{n}) & \text{if } \tau \text{ is a vector type;} \\ (\mathbf{n}) & \text{if } \tau \text{ is scalar.} \end{cases}$$

Then to every matrix schema $\mathcal{S}$ we associate the relational schema $\mathit{Rel}(\mathcal{S})$ where $\mathit{Rel}(\mathcal{S})(M) = \mathit{Rel}(\mathcal{S}(M))$ for every $M \in \mathrm{var}(\mathcal{S})$. For each instance I of $\mathcal{S}$, we define the instance $\mathit{Rel}(I)$ over $\mathit{Rel}(\mathcal{S})$ by

$$\mathit{Rel}(I)(M) = \begin{cases} \mathit{Rel}_2(I(M)) & \text{if } \mathcal{S}(M) \text{ is a general type;} \\ \mathit{Rel}_1(I(M)) & \text{if } \mathcal{S}(M) \text{ is a vector type;} \\ \mathit{Rel}_0(I(M)) & \text{if } \mathcal{S}(M) \text{ is the scalar type.} \end{cases}$$

Remark. The different treatment of matrices, vectors and scalars will allow us to use a “clean” version of the relational algebra where we do not need constants for base columns. We come back to this issue after the translation of MATLANG into the relation algebra with aggregates in the next subsection.

4.2 From MATLANG to relational algebra with summation

Given the representation of matrices by relations, we now show that MATLANG can be simulated in the relational algebra with aggregates. Actually, the only aggregate operation we need is summation. The relational algebra with summation extends the well-known relational algebra for relational databases and is defined as follows. For a full formal definition, see [43]. For our purposes it suffices to highlight the following about the relational algebra with summation:

- Expressions are built up from relation names using the classical operations union, set difference, cartesian product (×), selection (σ), and projection (π), plus two new operations: *function application* and *summation*.
- For selection, we only use equality and nonequality comparisons on base columns. No selection on numerical columns will be needed in our setting.
- For any function $f : \mathbb{C}^n \rightarrow \mathbb{C}$, the operation $\mathrm{apply}[f; i_1, \ldots, i_n]$ can be applied to any relation r having columns $i_1, \ldots, i_n$, which must be numerical. The result is the relation $\{(t, f(t(i_1), \ldots, t(i_n))) \mid t \in r\}$, appending a numerical column to r. We allow $n = 0$, in which case f is a constant.
- The operation $\mathrm{sum}[i; i_1, \ldots, i_n]$ can be applied to any relation r having columns $i, i_1, \ldots, i_n$, where column i must be numerical. In our setting we only need the operation in cases where columns $i_1, \ldots, i_n$ are base columns. The result of the operation is the relation

  $$\Big\{ \big(t(i_1), \ldots, t(i_n), \sum_{t' \in \mathrm{group}[i_1, \ldots, i_n](r,t)} t'(i) \big) \,\Big|\, t \in r \Big\},$$

  where

  $$\mathrm{group}[i_1, \ldots, i_n](r,t) = \big\{ t' \in r \,\big|\, t'(i_1) = t(i_1) \wedge \cdots \wedge t'(i_n) = t(i_n) \big\}.$$

  Again, n can be zero, in which case the result is a singleton. Note that in the definition of sum above we are using set semantics.

Given that relations are typed, one can define well-typedness for expressions in the relation algebra with summation, and define the output type. We omit this definition here, as it follows a well-known methodology [64] and is analogous to what we have already done for MATLANG in Section 3.3. The simulation of MATLANG into the relational algebra with summation can now be formally stated:

Theorem 4.1. *Let $\mathcal{S}$ be a matrix schema, and let e be a* MATLANG *expression that is well-typed over $\mathcal{S}$ with output type τ. Let $\ell = 2, 1,$ or 0, depending on whether τ is general, a vector type, or scalar, respectively.*

(1) *There exists an expression* $Rel(e)$ *in the relational algebra with summation, that is well-typed over* $Rel(\mathcal{S})$ *with output type* $Rel(\tau)$*, such that for every instance* I *of* $\mathcal{S}$*, we have* $Rel_\ell(e(I)) = Rel(e)(Rel(I))$*.*
(2) *The expression* $Rel(e)$ *uses neither set difference, nor selection conditions on numerical columns.*
(3) *The only functions used in* $Rel(e)$ *are those used in pointwise applications in* e*; complex conjugation; multiplication of two numbers; and the constant functions* 0 *and* 1*.*

Proof. We assign to each MATLANG expression e that is well-typed over $\mathcal{S}$, an expression $Rel(e)$ in the relational algebra with summation by induction on the structure of e. Since the let operation is syntactic sugar for MATLANG expressions, we do not consider this operation in this proof. Consider expressions e and e' in MATLANG and let $\tau = s_1 \times s_2$ be the output type of e'.

- If $e = M$ is a matrix variable of $\mathcal{S}$, then $Rel(e) := M$.
- If $e = (e')^*$, then

$$Rel(e) := \begin{cases} \pi_{1,2,4}\Big(\mathrm{apply}[\overline{z};3]\big(\pi_{2,1,3}(Rel(e'))\big)\Big) & \text{if } \tau \text{ is a general type;} \\ \pi_{1,3}\Big(\mathrm{apply}[\overline{z};2]\big(Rel(e')\big)\Big) & \text{if } \tau \text{ is a vector type;} \\ \pi_{2}\Big(\mathrm{apply}[\overline{z};1]\big(Rel(e')\big)\Big) & \text{if } \tau \text{ is the scalar type,} \end{cases}$$

  where $\overline{z}$ denotes the complex conjugate function mapping a complex number z to its complex conjugate $\overline{z}$.
- If $e = \mathbf{1}(e')$, then

$$Rel(e) := \begin{cases} \pi_{1,4}\Big(\mathrm{apply}[1;3]\big(Rel(e')\big)\Big) & \text{if } \tau \text{ is a general type;} \\ \pi_{1,3}\Big(\mathrm{apply}[1;2]\big(Rel(e')\big)\Big) & \text{if } s_1 \neq 1 = s_2; \\ \pi_{3}\Big(\mathrm{apply}[1;2]\big(Rel(e')\big)\Big) & \text{if } s_1 = 1 \neq s_2; \\ \pi_{2}\Big(\mathrm{apply}[1;1]\big(Rel(e')\big)\Big) & \text{if } \tau \text{ is the scalar type,} \end{cases}$$

  where 1 in the first argument of apply stands for the constant function $1 : \mathbf{C} \to \mathbf{C} : z \mapsto 1$. We observe the different treatment of $Rel(e')$ depending on whether e' is an $s_1 \times 1$ column vector or a $1 \times s_2$ row vector.
- If $e = \mathrm{diag}(e')$, then we define $Rel(e)$ as

$$\sigma_{1=2}\big(\pi_1(Rel(e')) \times Rel(e')\big) \cup \mathrm{apply}[0;\,]\Big(\sigma_{1\neq 2}\big(\pi_1(Rel(e')) \times \pi_1(Rel(e'))\big)\Big)$$

  if $s_1 \neq 1 = s_2$ and as $Rel(e')$ if τ is the scalar type. The 0 in the first argument of apply stands for the constant function $0 : \mathbf{C} \to \mathbf{C} : z \mapsto 0$.
- If $e = e_1 \cdot e_2$ where e_1 is of type $s_1 \times s_3$ and e_2 is of type $s_3 \times s_2$, then $Rel(e)$ is defined as

$$\begin{cases} \mathrm{sum}[7;1,5]\Big(\mathrm{apply}[\times;3,6]\big(\sigma_{2=4}(Rel(e_1) \times Rel(e_2))\big)\Big) & \text{if } s_1 \neq 1 \neq s_2 \text{ and } s_3 \neq 1; \\ \mathrm{sum}[6;1]\Big(\mathrm{apply}[\times;3,5]\big(\sigma_{2=4}(Rel(e_1) \times Rel(e_2))\big)\Big) & \text{if } s_1 \neq 1 = s_2 \text{ and } s_3 \neq 1; \\ \mathrm{sum}[6;4]\Big(\mathrm{apply}[\times;2,5]\big(\sigma_{1=3}(Rel(e_1) \times Rel(e_2))\big)\Big) & \text{if } s_1 = 1 \neq s_2 \text{ and } s_3 \neq 1; \\ \mathrm{sum}[5;\,]\Big(\mathrm{apply}[\times;2,4]\big(\sigma_{1=3}(Rel(e_1) \times Rel(e_2))\big)\Big) & \text{if } s_1 = 1 = s_2 \text{ and } s_3 \neq 1; \\ \pi_{1,3,5}\big(\mathrm{apply}[\times;2,4](Rel(e_1) \times Rel(e_2))\big) & \text{if } s_1 \neq 1 \neq s_2 \text{ and } s_3 = 1; \\ \pi_{1,4}\big(\mathrm{apply}[\times;2,3](Rel(e_1) \times Rel(e_2))\big) & \text{if } s_1 \neq 1 = s_2 \text{ and } s_3 = 1; \\ \pi_{2,4}\big(\mathrm{apply}[\times;1,3](Rel(e_1) \times Rel(e_2))\big) & \text{if } s_1 = 1 \neq s_2 \text{ and } s_3 = 1; \\ \pi_{3}\big(\mathrm{apply}[\times;1,2](Rel(e_1) \times Rel(e_2))\big) & \text{if } s_1 = 1 = s_2 \text{ and } s_3 = 1. \end{cases}$$

- Finally, if $e = \mathsf{apply}[f](e_1, \ldots, e_n)$, then $Rel(e)$ is defined as

  $$\begin{cases} \pi_{1,2,3n+1}\Big(\mathsf{apply}[f;3,6,\ldots,3n]\Big(\sigma_{p_1}\big(Rel(e_1)\times\cdots\times Rel(e_n)\big)\Big)\Big) & \text{if } \tau \text{ is a general type;}\\ \pi_{1,2n+1}\Big(\mathsf{apply}[f;2,4,\ldots,2n]\Big(\sigma_{p_2}\big(Rel(e_1)\times\cdots\times Rel(e_n)\big)\Big)\Big) & \text{if } \tau \text{ is a vector type;}\\ \pi_{n+1}\Big(\mathsf{apply}[f;1,2,\ldots,n]\big(Rel(e_1)\times\cdots\times Rel(e_n)\big)\Big) & \text{if } \tau \text{ is the scalar type,} \end{cases}$$

  where p_1 is the predicate $(1 = 4 = \cdots = (3n-5) = (3n-2)) \wedge (2 = 5 = \cdots = (3n-4) = (3n-1))$ and p_2 is the predicate $1 = 3 = \cdots = (2n-3) = (2n-1)$.

Notice that the only functions used in apply in $Rel(e)$ aside from those used in apply in e are complex conjugation ($\bar{z}$), multiplication of two numbers ($\times$), and the constant functions 0 and 1. Also notice that $Rel(e)$ uses neither set difference, nor selection conditions on numerical columns.

By induction on the structure of e one straightforwardly observes that (1) $Rel(e)$ is well-typed over $Rel(\mathcal{S})$ with output type $Rel(\tau)$ and (2) for every instance I of $\mathcal{S}$, we have $Rel_\ell(e(I)) = (Rel(e))(Rel(I))$, where ℓ is 2 if τ is of a general type, 1 if τ is of a vector type, and 0 if τ is of the scalar type. ☐

Remark. As mentioned earlier, the different treatment of general types, vector types, and scalar types allows us to use a "clean" version of the relational algebra, where we do not need constants for base columns. In contrast, if we had used the relational encoding Rel_2 also for vector types, for example by assuming that the second base attribute is the fixed constant 1, then expressing the $\mathbf{1}$ operation would require the constant 1 in the second base column:

$$Rel(\mathbf{1}(M)) = \pi_{1,\text{`}1',4}\big(\mathsf{apply}[1;3](M)\big),$$

with M a matrix variable of general type, cf. the definition of $Rel(\mathbf{1}(M))$ in the proof of Theorem 4.1 above. So, here we would need a generalized projection π that can insert a base column with constant '1'. (This constant 1 in a base column should not be confused with the value 1 in the numerical column.)

4.3 From MATLANG to relational calculus with summation

We can sharpen Theorem 4.1 by working not in the relational algebra, but in the *relational calculus* with aggregates. In this logic, we have base variables and numerical variables. Base variables can be bound to base columns of relations, and compared for equality. Numerical variables can be bound to numerical columns, and can be equated to function applications and aggregates. We will not recall the syntax formally (see [43] for a full definition). As an example expression in the relation calculus with aggregates we show how matrix multiplication is expressed. Matrix multiplication $M \cdot N$ with M of type $\alpha \times \beta$ and N of type $\beta \times \gamma$ can be expressed by the formula

$$\varphi(i,j,z) \equiv z = \mathsf{sum}\, k,x,y.(M(i,k,x) \wedge N(k,j,y), x \times y).$$

Here, i, j and k are base variables and x, y and z are numerical variables. The semantics of this expression is as follows. First, for given i, j, all triples (k,x,y) that satisfy $M(i,k,x) \wedge N(k,j,y)$ are collected. Then, the function $x \times y$ is applied to all these triples resulting in a multi-set consisting of the products xy. Finally, summation is applied on this multi-set and the result is assigned to variable z. We note that of the base variables, only i and j are free. In the subformula $M(i,k,x)$ only i and k are free, and in $N(k,j,y)$ only k and j are free.

The advantage of the relational calculus is that variables, especially base variables, can be *repeated* and *reused*. As we show below, this implies that when simulating MATLANG expression in the relational calculus with aggregates we only need formulas with at most *three* base variables. This will give us additional insights into the expressive power of MATLANG in Section 4.4.

To illustrate the reuse of variables, consider again our example expression $\varphi(i,j,z)$ corresponding to matrix multiplication. We observe that, if M or N had been a subexpression involving matrix multiplication in turn, we could have reused one of the three variables. For example, $(M \cdot N) \cdot N'$, where N' is of type $\gamma \times \delta$, can be expressed by the formula

$$\varphi'(i,j,z) \equiv z = \mathrm{sum}\, k,x,y.(M(i,k,x) \wedge \big(y = \mathrm{sum}\, i,x_1,x_2.(N(k,i,x_1) \wedge N'(i,j,x_2), x_1 \times x_2)\big), x \times y).$$

We will see that the other operations of MATLANG need only two base variables. We now state the simulation result more precisely:

Proposition 4.2. *Let $\mathcal{S}$, e, τ and ℓ as in Theorem 4.1. For every* MATLANG *expression e there is a formula φ_e over Rel($\mathcal{S}$) in the relational calculus with summation, such that*

1. *If τ is general, $\varphi_e(i,j,z)$ has two free base variables i and j and one free numerical variable z; if τ is a vector type, we have $\varphi_e(i,z)$; and if τ is scalar, we have $\varphi_e(z)$.*
2. *For every instance I, the relation defined by φ_e on Rel(I) equals $Rel_\ell(e(I))$.*
3. *The formula φ_e uses only three distinct base variables. The functions used in pointwise applications in φ_e are as in the statement of Theorem 4.1. Furthermore, φ_e neither uses equality conditions between numerical variables nor equality conditions on base variables involving constants.*

Proof. The proof is analogous to the proof of Theorem 4.1 and is deferred to the appendix. The only additional observation is that we only need three base variables, as explained earlier. ☐

4.4 Expressing graph queries

So far we have looked at expressing matrix queries in terms of relational queries. It is also natural to express relational queries as matrix queries. This works best for binary relations, or graphs, which we can represent by their adjacency matrices.

Formally, we define a *graph schema* to be a relational schema where every relation variable is assigned the type $(\mathbf{b},\mathbf{b})$ of arity two. We define a *graph instance* as an instance I of a graph schema, where the active domain of I equals $\{1,\dots,n\}$ for some positive natural number n. The assumption that the active domain always equals an initial segment of the natural numbers is convenient for forming the bridge to matrices. This assumption, however, is not essential for our results to hold. Indeed, the logics we consider do not have any built-in predicates on base variables, besides equality. Hence, they view the active domain elements as abstract data values.

To every graph schema $\mathcal{S}$ we associate a matrix schema $Mat(\mathcal{S})$, where $Mat(\mathcal{S})(R) = \alpha \times \alpha$ for every $R \in \mathrm{var}(\mathcal{S})$, for a fixed size symbol α. So, all matrices are square matrices of the same dimension. Let I be a graph instance of $\mathcal{S}$, with active domain $\{1,\dots,n\}$. We will denote the $n \times n$ adjacency matrix of a binary relation r over $\{1,\dots,n\}$ by $Adj_I(r)$. Now any such instance I is represented by the matrix instance $Mat(I)$ over $Mat(\mathcal{S})$, where $Mat(I)(R) = Adj_I(I(R))$ for every $R \in \mathrm{var}(\mathcal{S})$.

A *graph query* over a graph schema $\mathcal{S}$ is a function that maps each graph instance I of $\mathcal{S}$ to a binary relation on the active domain of I. We say that a MATLANG expression e *expresses* the graph query q if e is well-typed over $Mat(\mathcal{S})$ with output type $\alpha \times \alpha$, and for every graph instance I of $\mathcal{S}$, we have $Adj_I(q(I)) = e(Mat(I))$.

We can now give a partial converse to Theorem 4.1. We assume active-domain semantics for first-order logic [1]. Please note that the following result deals only with pure first-order logic, without aggregates or numerical columns.

Theorem 4.3. *Every graph query expressible in* FO^3 *(first-order logic with equality, using at most three distinct variables) is expressible in* MATLANG. *The only functions needed in pointwise applications are boolean functions on $\{0,1\}$, and testing if a number is positive.*

Proof. It is known [46, 61] that FO^3 graph queries can be expressed in the algebra of binary relations with the operations *all*, identity, union, set difference, converse, and relational composition. These operations are well known, except perhaps for *all*, which, on a graph instance I, evaluates to the cartesian product of the active domain of I with itself. Identity evaluates to the identity relation on the active domain of I. Each of these operations is easy to express in MATLANG. For *all* we use $\mathbf{1}(R) \cdot \mathbf{1}(R)^*$, where for R we can take any relation variable from the schema. Identity is expressed as $\mathrm{diag}(\mathbf{1}(R))$. Union $r \cup s$ is expressed as $\mathrm{apply}[x \vee y](r, s)$, and set difference $r - s$ as $\mathrm{apply}[x \wedge \neg y](r, s)$. Converse is transpose. Relational composition $r \circ s$ is expressed as $\mathrm{apply}[x > 0](r \cdot s)$, where $x > 0$ is 1 if x is positive and 0 otherwise. □

We can complement the above theorem by showing that the quintessential first-order query requiring four variables is not expressible.

Proposition 4.4. *The graph query over a single binary relation R that maps I to $I(R)$ if $I(R)$ contains a four-clique, and to the empty relation otherwise, is not expressible in* MATLANG.

To prove Proposition 4.4 we first state the following lemma, which refines Proposition 4.2 in the setting of graph queries.

Lemma 4.5. *If a graph query q is expressible in* MATLANG, *then q is expressible by a formula $\psi_q(i, j)$ in the relational calculus with summation, where i and j are base variables, and ψ_q uses at most three distinct base variables.*

Proof. Let e be a MATLANG expression that expresses q. Let $\varphi_e(i, j, z)$ be the formula given by Proposition 4.2. This formula does not express the graph query q since it has a free numerical variable and contains relation variables (of type $(\mathbf{b}, \mathbf{b}, \mathbf{n})$) corresponding to the matrix variables in e. We need to transform $\varphi_e(i, j, z)$ into an expression over relation variables (of type $(\mathbf{b}, \mathbf{b})$) in the graph schema and ensure that there are only two free base variables. This can be easily done, as follows. First, let $\varphi'_e(i, j, z)$ be the formula obtained from $\varphi_e(i, j, z)$ by replacing each atomic formula of the form $R(i', j', x)$, where i' and j' are base variables and x is a numerical variable, by $(x = 1 \wedge R(i', j')) \vee (x = 0 \wedge \neg R(i', j'))$. Here, we are simply expressing the adjacency matrix stored in $R(i', j', x)$ by means of the binary relation $R(i', j')$. Now $\psi_q(i, j)$ can be obtained as $\exists z\, (z = 1 \wedge \varphi'(i, j, z))$. Indeed, it suffices to only list those positions in the result adjacency matrix that are non-zero. The fact that ψ_q only uses three base variables is simply because φ_e only uses three base variables and in the transformation from φ_e to ψ_q we did not introduce additional base variables. □

We now show that MATLANG cannot verify the existence of four-cliques.

Proof of Proposition 4.4. Let e be a MATLANG expression expressing some graph query q. Let ψ_q be the formula given by Lemma 4.5. Although ψ_q takes a binary relation of type $(\mathbf{b}, \mathbf{b})$ as input and also returns a binary relation of type $(\mathbf{b}, \mathbf{b})$, (non-free) numerical variables may be present in ψ_q. To show that the existence of four-cliques cannot be expressed we want to rely on a result for logics in which only base variables are allowed. The challenge is to eliminate the numerical variables from ψ_q.

This can be done as follows. First, we eliminate all pointwise function applications, arithmetic and summation from $\psi_q(i, j)$ following a standard method. Indeed, it is known [30, 43] that every formula in the relational calculus with aggregates can be equivalently expressed by a formula $\psi_q^\circ(i, j)$ in infinitary logic with counting. This logic, referred to as $\mathcal{L}_C$ in [43], works on typed relations (types $\mathbf{b}$ and $\mathbf{n}$) and extends first-order logic with infinitary disjunctions and conjunctions, and counting quantifiers $\exists^{\geq m}$, for $m \geq 1$, on base variables. We refer to [30, 43] for the detailed translation. We observe that the base variables in $\psi_q^\circ(i,j)$ are those in the original formula $\psi_q(i,j)$ and thus $\psi_q^\circ(i,j)$ only uses at most three base variables. Furthermore, we note that in $\psi_q^\circ(i,j)$ all numerical variables are quantified. Consider such a numerical variable z and let $\exists z\, \varphi(\bar{x}, z, \bar{z}')$ be the sub-formula in $\psi_q^\circ(i,j)$ in which z occurs. In this formula, $\bar{x}$ are base variables and z and $\bar{z}'$ are numerical variables. Then, to eliminate the variable z it suffices to add one infinitary disjunction and replace $\exists z\, \varphi(\bar{x}, z, \bar{z}')$ by $\bigvee_{c \in \mathbf{C}} \varphi(\bar{x}, c, \bar{z}')$. In other words, we replace z by all possible complex numbers. By doing this for every numerical variable in $\psi_q^\circ(i,j)$ we end up with a formula $\varphi_q(i,j)$ in which no numerical variables are present. Proposition 4.2 further states that $\psi_q(i,j)$, and thus also $\varphi_q^\circ(i,j)$ and $\varphi(i,j)$, does not involve equality conditions between base variables and constants. So, $\varphi_q(i,j)$ only contains "pure" equalities between variables. This is a consequence of our encoding of matrices into relations (recall our earlier remark on how we avoided the need for the constant '1' in base columns).

Hence, $\varphi_q(i,j)$ is a formula in infinitary counting logic with three distinct variables over a graph schema. This logic is denoted by $C^3_{\infty\omega}$ in [53] and the four-clique query is not expressible in $C^3_{\infty\omega}$. Indeed, to see this, consider the four-clique graph G, to which we apply the Cai-Fürer-Immerman construction [13, 53], yielding graphs G^0 and G^1 which are indistinguishable in $C^3_{\infty\omega}$.[1] This construction is such that G^0 contains a "four-clique formed by paths of length three": four nodes such that there is a path of length three between any two of them. The graph G^1, however, does not contain four such nodes.

Now suppose, for the sake of contradiction, that there would be a sentence φ in $C^3_{\infty\omega}$ expressing the existence of a four-clique. We can replace each atomic formula $R(x,y)$ by $\exists z(R(x,z) \land \exists x(R(z,x) \land R(x,y)))$. The resulting $C^3_{\infty\omega}$ sentence looks for a four-clique formed by paths of length three, and would distinguish G^0 from G^1, which yields our contradiction.

Similarly, suppose that we can express the four-clique graph query q as in the statement of the proposition by means of a MATLANG expression e. We then consider the $C^3_{\infty\omega}$ sentence $\exists i,j\, \varphi_q(i,j)$ which returns true on a graph if and only if the graph contains a four-clique, which again leads to a contradiction. ☐

We conclude by showing that MATLANG cannot express the transitive-closure graph query which maps a graph to its transitive closure. Indeed, by Theorem 4.1, any graph query expressible in MATLANG is expressible in the relational algebra with aggregates. It is known [30, 43] that such queries are local. We recall the definition of locality. For a graph G, vertices a and b, and a nonnegative integer r, denote by $N_r^G(a,b)$ the subgraph of G induced by the vertices that are at most distance r from either a or b, where by distance we mean the shortest path length in the undirected graph induced by G. A graph query q over a schema with one relation variable R is said to be *local* if there is a nonnegative integer r such that for every graph instance I and for all vertices a, b, c, d, the existence of a graph isomorphism h from $N_r^{I(R)}(a,b)$ to $N_r^{I(R)}(c,d)$ with $h(a) = c$ and $h(b) = d$ implies that $(a,b) \in q(I)$ if and only if $(c,d) \in q(I)$. The transitive-closure query, however, is known not to be local [43]. We thus conclude:

PROPOSITION 4.6. *The graph query over a single binary relation R that maps I to the transitive-closure of $I(R)$ is not expressible in* MATLANG. ☐

5 MATRIX INVERSION

We now consider the extension of MATLANG with matrix inversion. More precisely, we extend MATLANG as follows. Let $\mathcal{S}$ be a schema and e be an expression that is well-typed over $\mathcal{S}$, with output type of the form $\alpha \times \alpha$. Then the expression e^{-1} is also well-typed over $\mathcal{S}$, with the same

---

[1] Specifically, G^0 and G^1 are the graphs $\mathfrak{A}$ and $\mathfrak{A}'$ defined by Otto [53, Example 2.7 and Lemma 2.8] for the case $m = 3$.

output type $\alpha \times \alpha$. The semantics is defined as follows. For an instance I, if $e(I)$ is an invertible matrix, then $e^{-1}(I)$ is defined to be the inverse of $e(I)$; otherwise, it is defined to be the zero square matrix of the same dimensions as $e(I)$. The extension of MATLANG with inversion is denoted by MATLANG + inv.

Example 5.1 (PageRank). Recall Example 1.1 where we computed the Google matrix of A. In the process we already showed how to compute the $n \times n$ matrix B defined by $B_{i,j} = A_{i,j}/k_i$, and the scalar n. We use e_B and e_n to denote the corresponding MATLANG expressions. Let I be the $n \times n$ identity matrix, and let $\mathbf{1}$ denote the $n \times 1$ column vector consisting of all ones. The PageRank vector v of A can be computed as follows [21]:

$$v = \frac{1-d}{n}(I - dB)^{-1}\mathbf{1}.$$

This calculation is readily expressed in MATLANG + inv as

$$(1-d) \odot (\mathsf{apply}[1/x](e_n)) \odot \big(\mathsf{apply}[-](\mathsf{diag}(\mathbf{1}(M)), d \odot e_B)\big)^{-1} \cdot \mathbf{1}(M). \qquad \square$$

Example 5.2 (Transitive closure). We next show that the reflexive-transitive closure of a binary relation is expressible in MATLANG + inv. Let A be the adjacency matrix of a binary relation r on $\{1, \dots, n\}$. Let I be the $n \times n$ identity matrix, expressible as $\mathsf{diag}(\mathbf{1}(A))$. Let e_n be the expression computing the scalar n. The matrix $B = \frac{1}{n+1}A$ has 1-norm strictly less than 1, so $S = \sum_{k=0}^{\infty} B^k$ converges, and is equal to $(I - B)^{-1}$ [25, Lemma 2.3.3]. Now (i, j) belongs to the reflexive-transitive closure of r if and only if $S_{i,j}$ is nonzero. Thus, we can compute the reflexive-transitive closure of r by evaluating

$$\mathsf{apply}[\neq 0]\Big(\Big(\mathsf{apply}[-]\big(\mathsf{diag}(\mathbf{1}(M)), \mathsf{apply}[1/(x+1)](e_n) \odot M\big)\Big)^{-1}\Big),$$

by assigning matrix variable M to A. Here, $\neq 0$ is the function which returns 1 if the value is nonzero and 0 otherwise. We can express the transitive closure by multiplying the above expression by M. □

Given our earlier observation that the transitive-closure query cannot be expressed in MATLANG (Proposition 4.6) and the MATLANG + inv expression given in the previous example which does express this query, we may conclude:

THEOREM 5.3. MATLANG + inv *is strictly more powerful than* MATLANG *in expressing graph queries.*

Once we have the transitive closure, we can do many other things such as checking bipartiteness of undirected graphs, checking connectivity, and checking cyclicity. MATLANG is expressive enough to reduce these queries to the transitive-closure query, as shown in the following example for bipartiteness. The same approach via FO^3 can be used for connectedness or cyclicity.

Example 5.4 (Bipartiteness). To check bipartiteness of an undirected graph, given as a symmetric binary relation R without self-loops, we first compute the transitive closure T of the composition of R with itself. Then the FO^3 condition $\neg\exists x \exists y (R(x, y) \wedge T(y, x))$ expresses that R is bipartite (no odd cycles). The result now follows from Theorem 4.3. □

Example 5.5 (Number of connected components). Using transitive closure we can also easily compute the number of connected components of a binary relation R on $\{1, \dots, n\}$, given as an adjacency matrix. We start from the union of R and its converse. This union, denoted by S, is expressible by Theorem 4.3. We then compute the reflexive-transitive closure C of S. Now the number of connected components of R equals $\sum_{i=1}^{n} 1/k_i$, where k_i is the degree of node i in C. This sum is simply expressible as $\mathbf{1}(C)^* \cdot \mathsf{apply}[1/x](C \cdot \mathbf{1}(C))$. □

Example 5.6 (Regular path queries). MATLANG + inv can express regular path queries on graph databases [68]. For different edge labels, say a and b, we use different matrices, say A and B, respectively, to store the adjacency matrices of the a-edges and b-edges. Regular path queries are, syntactically, regular expressions over the edge labels. Now, concatenation and union are expressed in MATLANG as already described in the proof of Theorem 4.3. Kleene star is expressed as described in Example 5.2. □

We do not know whether the four-clique graph query can be expressed in MATLANG + inv.

6 EIGENVECTORS

Another workhorse in data analysis is diagonalizing a matrix, i.e., finding a basis of eigenvectors. We next consider the extension of MATLANG with an operation eigen.

Formally, we define the operation eigen as follows. Let A be an $n \times n$ matrix. Recall that A is called diagonalizable if there exists a basis of $\mathbf{C}^n$ consisting of eigenvectors of A. In that case, there also exists such a basis where eigenvectors corresponding to the same eigenvalue are orthogonal. Accordingly, we define eigen(A) to return an $n \times n$ matrix, the columns of which form a basis of $\mathbf{C}^n$ consisting of eigenvectors of A, where eigenvectors corresponding to a same eigenvalue are orthogonal. If A is not diagonalizable, we define eigen(A) to be the $n \times n$ zero matrix.

Note that eigen is nondeterministic; in principle there are infinitely many possible results. This models the situation in practice where numerical packages such as R or MATLAB return approximations to the eigenvalues and a set of corresponding eigenvectors. Eigenvectors, however, are not unique. In fact, there are infinitely many eigenvectors.

Hence, some care must be taken in extending MATLANG with the eigen operation. Syntactically, as for inversion, whenever e is a well-typed expression with a square output type, we now also allow the expression eigen(e), with the same output type. Semantically, however, the rules of Figure 2 must be adapted so that they do not infer statements of the form $e(I) = B$, but rather of the form $B \in e(I)$, i.e., B is a possible result of $e(I)$. The let-construct now becomes crucial; it allows us to assign a possible result of eigen to a new variable, and work with that intermediate result consistently.

In this and the next section, we assume notions from linear algebra. An excellent introduction to the subject has been given by Axler [4].

Remark (Eigenvalues). We can easily recover the eigenvalues from the eigenvectors, using inversion. Indeed, if A is diagonalizable and $B \in \text{eigen}(A)$, then $\Lambda = B^{-1}AB$ is a diagonal matrix with all eigenvalues of A on the diagonal, so that the ith eigenvector in B corresponds to the eigenvalue in the ith column of Λ. This is the well-known eigendecomposition. However, the same can also be accomplished without using inversion. Indeed, suppose $B = (v_1, \dots, v_n)$, and let λ_i be the eigenvalue to which v_i corresponds. Then $AB = (\lambda_1 v_1, \dots, \lambda_n v_n)$. Each eigenvector is nonzero, so we can divide away the entries from B in AB (setting division by zero to zero). We thus obtain a matrix where the ith column consists of zeros or λ_i, with at least one occurrence of λ_i. By counting multiplicities, dividing them out, and finally summing, we obtain $\lambda_1, \dots, \lambda_n$ in a column vector. We can apply a final diag to get it back into diagonal form. The MATLANG expression for doing all this uses similar tricks as those shown in Examples 1.1 and 3.6. □

The above remark suggests a shorthand in MATLANG + eigen where we return both B (eigenvectors) and Λ (eigenvalues) together:

$$\text{let } (B, \Lambda) = \text{eigen}(A) \text{ in } \dots$$

This models how the eigen operation works in the languages R and MATLAB. We agree that Λ, like B, is the zero matrix if A is not diagonalizable.

Example 6.1 (Rank of a matrix). Since the rank of a diagonalizable matrix equals the number of nonzero entries in its diagonal form, we can express the rank of a diagonalizable matrix A as follows:

$$\text{let } (B,\Lambda) = \text{eigen}(A) \text{ in } \mathbf{1}(A)^* \cdot \text{apply}[\neq 0](\Lambda) \cdot \mathbf{1}(A). \qquad \square$$

Example 6.2 (Graph partitioning). A popular graph clustering method consists of partitioning the vertex set V of a graph $G = (V, E)$ into two parts V_1 and $V_2 = V \setminus V_1$ such that the number of edges between vertices in these two parts is minimized, and, in addition, the number of vertices in V_1 and V_2 are the same [42]. This optimization problem can be phrased in terms of the Laplacian $L = D - A$ of the adjacency matrix A of G. Here, D, called the degree matrix of A, is the diagonal matrix where each diagonal entry is equal to the degree of the corresponding vertex. More specifically, it suffices to solve $f_{\text{opt}} = \arg\min_f f^* \cdot L \cdot f$ such that $f^* \cdot \mathbf{1} = 0$ and $f_v \in \{-1, 1\}$ for $v \in V$ [42]. Due to the intractability of the corresponding decision problem [66], in practice, the relaxed optimization problem $f_{\text{opt}} = \arg\min_f f^* \cdot L \cdot f$ such that $f^* \cdot \mathbf{1} = 0$ and $f^* \cdot f = n$, where n is the number of vertices in G, is solved instead. Furthermore, a partitioning of V is obtained from f_{opt} by defining $V_1 = \{v \in V \mid f_v \geq 0\}$ and $V_2 = \{v \in V \mid f_v < 0\}$. We consider connected graphs G and assume, for convenience, that the second-smallest eigenvalue λ_2 (i.e., the smallest non-zero eigenvalue) of their laplacian L has multiplicity one so that all the eigenvectors of λ_2 are scalar multiples of each other [2]. Such an eigenvector is call a *Fiedler* vector and is known to be a solution of the relaxed optimization problem. We now show that Fiedler vectors can be obtained in MATLANG + eigen. Indeed, the Laplacian L can be derived from the adjacency matrix A as

$$\text{let } D = \text{diag}(A \cdot \mathbf{1}(A)) \text{ in } \text{apply}[-](D, A).$$

Now let $(B, \Lambda) \in \text{eigen}(L)$. In an analogous way to Example 3.6, we can compute a matrix E, obtained from Λ by replacing the occurrences of the second-smallest eigenvalue λ_2 by 1 and all other entries by 0. Then an eigenvector f corresponding to this eigenvalue can be isolated from B (and the other eigenvectors zeroed out) by multiplying $B \cdot E$. We then normalize f such that $f^* \cdot f = n$. We remark that f is not unique. Nevertheless we want to return a representation of the induced partition into V_1 and V_2 which is independent of the eigenvector f returned. To do so, we first set non-negative entries in f to 1 and negative entries to -1 by means of a function application $\pm 1(x) = 1$ if $x \geq 0$ and $\pm 1(x) = -1$ otherwise. Next we create a $|V| \times |V|$ matrix P such that $P_{ij} = 1$ if vertices i and j belong to the same partition and $P_{ij} = 0$ otherwise. We can do this by evaluating $\text{apply}[>0]\big(\text{apply}[\pm 1](f) \cdot (\text{apply}[\pm 1](f))^*\big)$, where > 0 maps every positive entry to 1 and all non-positive entries to 0. $\square$

It turns out that MATLANG + inv is subsumed by MATLANG + eigen.

Theorem 6.3. *Matrix inversion is expressible in* MATLANG + eigen.

Proof. We describe a fixed procedure for determining A^{-1}, for any square matrix A. Let $S = A^*A$. Then A is invertible if and only if S is. Let us assume first that S is indeed invertible.

Since S is self-adjoint, $\mathbf{C}^n$ has an orthogonal basis consisting of eigenvectors of S. Eigenvectors of a self-adjoint operator that correspond to distinct eigenvalues are always orthogonal. Hence, eigen(S) always returns an orthogonal basis of $\mathbf{C}^n$ consisting of eigenvectors of S. Let $(B, \Lambda) \in \text{eigen}(S)$ (using the shorthand introduced before Example 6.1). We can normalize the columns of B in MATLANG as

$$\text{apply}[x/\sqrt{y}](B, \mathbf{1}(B) \cdot (B^* \cdot B \cdot \mathbf{1}(B))^*).$$

---

[2] If λ_2 has multiplicity $m > 1$, we have m independent eigenvectors for this eigenvalue. Since in MATLANG we cannot select a single one of these eigenvectors, the construction given in this example needs to be modified. More precisely, all m eigenvectors are extracted and combined into a single eigenvector. This can be done, for example, by summing up all m eigenvectors.

(This expression works because the columns in B are mutually orthogonal.) So, we may now assume that B contains an orthonormal basis consisting of eigenvectors of S. In particular, $B^{-1} = B^*$, and $S = B\Lambda B^*$.

Since we have assumed S to be invertible, none of the eigenvalues is zero. We can invert Λ simply by replacing each entry on the diagonal by its reciprocal. Thus, Λ^{-1} can be computed from Λ by pointwise application of the reciprocal function.

Now A^{-1} can be computed by the expression $C = B\Lambda^{-1}B^*A^*$. To see that C indeed equals A^{-1}, we calculate $CA = B\Lambda^{-1}B^*A^*A = B\Lambda^{-1}B^*S = B\Lambda^{-1}B^*B\Lambda B^*$ which simplifies to the identity matrix.

When S is not invertible, we should return the zero matrix. In MATLANG we can compute the matrix Z that is zero if one of the eigenvalues is zero, and the identity matrix otherwise. We then multiply the final expression with Z. A final detail is to make the computation well-defined in all cases. Note that the functions $(x, y) \mapsto x/\sqrt{y}$ and $x \mapsto 1/x$, used in pointwise applications, are not total functions. If S is invertible, then, in the pointwise application of $x/\sqrt{y}$, the argument y is always a positive real number, and, in the pointwise application of $1/x$, the argument x is always nonzero. If S is not invertible, then $x/\sqrt{y}$ and $1/x$ can be extended to total functions in an arbitrary manner. $\square$

We do not know whether the four-clique graph query can be expressed in MATLANG + eigen. Another interesting open problem is the following: *Are there graph queries expressible deterministically in* MATLANG + eigen*, but not in* MATLANG + inv*?* This is an interesting question for further research. The answer may depend on the functions that can be used in pointwise applications.

Remark (Determinacy). The stipulation *deterministically* in the above open question is important. Ideally, we use the nondeterministic eigen operation only as an intermediate construct. It is an aid to achieve a powerful computation, but the final expression should have only a single possible output on every input. The expression of Example 6.1 is deterministic in this sense, as is the expression for inversion underlying the proof of Theorem 6.3.

7 THE EVALUATION PROBLEM

We next consider the evaluation problem of expressions in our most expressive language MATLANG+ eigen. Naively, the evaluation problem asks, given an input instance I and an expression e, to compute the result $e(I)$. There are some issues with this naive formulation, however. Indeed, in our theory we have been working with arbitrary complex numbers. How do we even represent the input? Notably, the eigen operation on a matrix with only rational entries may produce irrational entries. In fact, the eigenvalues of an adjacency matrix (even of a tree) need not even be definable in radicals [24]. Practical systems, of course, apply numerical methods to compute rational approximations. But it is still theoretically interesting to consider the exact evaluation problem. For a treatise on computations of eigenvectors, inverses, and other matrix notions, we refer to [25].

Our approach is to represent the output symbolically, following the idea of constraint query languages [35, 40]. Specifically, we can define the input-output relation of an expression, for given dimensions of the input matrices, by an *existential first-order logic formula over the reals.* Such formulas are built from real variables, integer constants, addition, multiplication, equality, inequality ($<$), disjunction, conjunction, and existential quantification.

Any $m \times n$ matrix A can be represented by a tuple of $2mn$ real numbers. Indeed, let $a_{i,j} = \Re A_{i,j}$ (the real part of a complex number), and let $b_{i,j} = \Im A_{i,j}$ (the imaginary part). Then A can be represented by the tuple $(a_{1,1}, b_{1,1}, a_{1,2}, b_{1,2}, \dots, a_{m,n}, b_{m,n})$. The next result introduces the variables $x_{M,i,j,\Re}$, $x_{M,i,j,\Im}$, $y_{i,j,\Re}$, and $y_{i,j,\Im}$, where the x-variables describe an arbitrary input matrix $I(M)$ and the y-variables describe an arbitrary possible output matrix $e(I)$.

In the following, an *input-sized expression* consists of a schema $\mathcal{S}$, an expression e in MATLANG + eigen that is well-typed over $\mathcal{S}$ with output type $t_1 \times t_2$, and a size assignment σ defined on the size symbols occurring in $\mathcal{S}$. For complexity considerations, we assume the sizes given in σ are coded in unary. Whether this assumption can be avoided remains open.

THEOREM 7.1. *There exists a polynomial-time computable translation that maps any input-sized expression e to an existential first-order formula ψ_e over the vocabulary of the reals, expanded with symbols for the functions used in pointwise applications in e, such that*

(1) *Formula ψ_e has the following free variables:*
  - *For every $M \in \mathrm{var}(\mathcal{S})$, let $\mathcal{S}(M) = s_1 \times s_2$. Then ψ_e has the free variables $x_{M,i,j,\Re}$ and $x_{M,i,j,\Im}$, for $i = 1, \dots, \sigma(s_1)$ and $j = 1, \dots, \sigma(s_2)$.*
  - *In addition, ψ_e has the free variables $y_{e,i,j,\Re}$ and $y_{e,i,j,\Im}$, for $i = 1, \dots, \sigma(t_1)$ and $j = 1, \dots, \sigma(t_2)$.*

  The set of these free variables is denoted by $\mathrm{FV}(\mathcal{S}, e, \sigma)$.

(2) *Any assignment ρ of real numbers to these variables specifies, through the x-variables, an instance I conforming to $\mathcal{S}$ by σ, and through the y-variables, a $\sigma(t_1) \times \sigma(t_2)$ matrix B.*

(3) *Formula ψ_e is true over the reals under such an assignment ρ, if and only if $B \in e(I)$.*

PROOF. We prove this result by induction on the structure of e. Let I be an instance conforming to $\mathcal{S}$ by σ. For notational transparency we work in this proof exclusively with complex numbers. It is then understood that formulas like "$y_{e,i,j} = x_{M,i,j}$" are short for $(y_{e,i,j,\Re} = x_{M,i,j,\Re}) \wedge (y_{e,i,j,\Im} = x_{M,i,j,\Im})$.

- Let $e = M$ for some matrix variable $M \in \mathrm{var}(\mathcal{S})$. We have $e(I) = I(M)$ and so the formula $\psi_e := \bigwedge_{i,j}(y_{e,i,j} = x_{M,i,j})$ satisfies the required property. Here, i ranges over $\{1, \dots, \sigma(t_1)\}$ and j ranges over $\{1, \dots, \sigma(t_2)\}$.
- Let $e = \mathsf{let}\ M = e_1\ \mathsf{in}\ e_2$. Then the formula $\psi_e := \exists_{i,j}\ y_{e_1,i,j}, y_{e_2,i,j}(\psi_{e_1} \wedge \psi_{e_2} \wedge \bigwedge_{i,j}(y_{e_1,i,j} = x_{M,i,j}) \wedge \bigwedge_{i,j}(y_{e,i,j} = y_{e_2,i,j}))$ satisfies the required property.
- Let $e = (e_1)^*$. Then the formula $\psi_e := \exists_{i,j}\ y_{e_1,i,j}(\psi_{e_1} \wedge \bigwedge_{i,j}(y_{e,i,j} = y^*_{e_1,j,i}))$ satisfies the required property. Here, $y_{e,i,j} = y^*_{e_1,j,i}$ is short for $(y_{e,i,j,\Re} = y_{e_1,j,i,\Re}) \wedge (y_{e,i,j,\Im} = -y_{e_1,j,i,\Im})$.
- Let $e = \mathbf{1}(e_1)$. Then the formula $\psi_e := \bigwedge_i(y_{e,i,1} = 1) \wedge \bigwedge_{i,j} x_{M,i,j}$ satisfies the required property.
- Let $e = \mathrm{diag}(e_1)$. Then

$$\psi_e := \Big(\bigwedge_{\substack{i,j \\ i \neq j}}(y_{e,i,j} = 0)\Big) \wedge \exists_i\ y_{e_1,i,1}\Big(\psi_{e_1} \wedge \bigwedge_i (y_{e,i,i} = y_{e_1,i,1})\Big)$$

  satisfies the required property.
- Let $e = e_1 \cdot e_2$. Then the formula $\psi_e := \exists_{i,k}\ y_{e_1,i,k} \exists_{k,j}\ y_{e_2,k,j}\big(\psi_{e_1} \wedge \psi_{e_2} \wedge \bigwedge_{i,j}(y_{e,i,j} = \sum_k y_{e_1,i,k} \cdot y_{e_2,k,j})\big)$ satisfies the required property.
- Let $e = \mathrm{apply}[f](e_1, \dots, e_n)$. Then the formula $\psi_e := \exists_{i,j,k}\ y_{e_k,i,j}\big((\bigwedge_k \psi_{e_k}) \wedge \big(\bigwedge_{i,j}(y_{e,i,j} = f(y_{e_1,i,j}, \dots, y_{e_n,i,j}))\big)\big)$ satisfies the required property (here, f is merely a symbol).
- Let $e = \mathrm{eigen}(e_1)$. Denote by $[\bar{y}_{e_1}]$ the symbolic matrix corresponding to e_1 and denote by $[\bar{y}_e]$ the symbolic matrix corresponding to e.
  - To express that $[\bar{y}_e]$ is a basis, we write that there exists a matrix $[\bar{z}]$ such that $[\bar{y}_e] \cdot [\bar{z}]$ is the identity matrix. This condition is expressed by the following formula $\psi_{\mathrm{basis},e} := \exists_{j,k}\ z_{j,k}\big((\bigwedge_{\substack{i,k \\ i \neq k}}(\sum_j y_{e,i,j} \cdot z_{j,k} = 0)) \wedge (\bigwedge_i(\sum_j y_{e,i,j} \cdot z_{j,i} = 1))\big)$.

- To express, for each column vector v of $[\bar{y}_e]$, that v is an eigenvector of $[\bar{y}_{e_1}]$, we write that there exists λ such that $[\bar{y}_{e_1}] \cdot v = \lambda[\bar{y}_{e_1}]$. Explicitly, this condition is expressed by the formula $\psi_{\text{eigenv},e} := \bigwedge_j (\exists \lambda (\bigwedge_i (\sum_k y_{e_1,i,k} \cdot y_{e,k,j} = \lambda \cdot y_{e_1,i,j})))$.
- More challenging is to express is that distinct eigenvectors v and w that correspond to the same eigenvalue are orthogonal. We cannot write $\exists\lambda([\bar{y}_{e_1}] \cdot v = \lambda v \wedge [\bar{y}_{e_1}] \cdot w = \lambda w) \rightarrow v^* \cdot w = 0$, as this is not an existential formula due to the use of logical implication. Instead, we avoid an explicit quantifier over the eigenvalue λ by recovering it from the eigenvectors. This is done in a similar way as in how we retrieved the eigenvalues from the eigenvectors in the previous section. More precisely, given that v and w are eigenvectors we have that $([\bar{y}_{e_1}] \cdot v)_i = \lambda \cdot v_i$ and $([\bar{y}_{e_1}] \cdot w)_i = \mu \cdot w_i$ for eigenvalues λ and μ, respectively. The vectors v and w will be eigenvectors of the same eigenvalue if whenever $v_i \neq 0 \neq w_i$, $([\bar{y}_{e_1}] \cdot v)_i/v_i = ([\bar{y}_{e_1}] \cdot w)_i/w_i$. Furthermore, we remark that when $v_i \neq 0 \neq w_i$ never holds, then v and w are necessarily orthogonal. We thus use this condition in the premise of the implication and write

$$\psi_{\text{ortho},e} := \bigwedge_{\substack{v,w \text{ columns in } [y_e],\\ v \neq w}} \Bigg(\Big(\bigwedge_i \big(v_i \neq 0 \neq w_i \rightarrow ([\bar{y}_{e_1}] \cdot v)_i/v_i = ([\bar{y}_{e_1}] \cdot w)_i/w_i\big)\Big) \rightarrow v^* \cdot w = 0\Bigg).$$

- A final detail is that we should also be able to express that $[\bar{y}_{e_1}]$ is not diagonalizable, for in that case we need to define $[\bar{y}_e]$ to be the zero matrix. Nondiagonalizability is equivalent to the existence of a Jordan form with at least one 1 on the superdiagonal. We can express this as follows. We postulate the existence of an invertible matrix $[\bar{z}]$ such that the product $[\bar{z}] \cdot [\bar{y}_{e_1}] \cdot [\bar{z}]^{-1}$ has all entries zero, except those on the diagonal and the superdiagonal. The entries on the superdiagonal can only be 0 or 1, with at least one 1. Moreover, if an entry i, j on the superdiagonal is nonzero, the entries i, i and j, j must be equal. Denote by $\psi_{\text{notdiagable},e_1}$ the formula that expresses that $[\bar{y}_{e_1}]$ is not diagonalizable.

Putting all of the above pieces together, we obtain the following formula

$$\psi_e := \exists_{i,j}\, y_{e_1,i,j} \big(\psi_{e_1} \wedge \big((\psi_{\text{basis},e} \wedge \psi_{\text{eigenv},e} \wedge \psi_{\text{ortho},e}) \vee (\psi_{\text{nondiagable},e_1} \wedge \psi_{\text{null},e})\big)\big),$$

where $\psi_{\text{null},e} := \bigwedge_{i,j} y_{e,i,j} = 0$ to create the zero matrix in case of non-diagonalizability.

It should be clear from the translation that ψ_e can be computed in polynomial time and indeed satisfies the conditions (1), (2) and (3) as stated in the theorem. □

The existential theory of the reals is decidable; actually, the full first-order theory of the reals is decidable [3, 5]. But, specifically the class of problems that can be reduced in polynomial time to the existential theory of the reals forms a complexity class on its own, known as ∃R [58, 59]. This class lies between NP and PSPACE. The above theorem implies that the *intensional evaluation problem for* MATLANG + eigen belongs to this complexity class. We define this problem as follows. The idea is that an arbitrary specification, expressed as an existential formula χ over the reals, can be imposed on the input-output relation of an input-sized expression.

Definition 7.2. The *intensional evaluation problem* is a decision problem that takes as input:

- an input-sized expression $(\mathcal{S}, e, \sigma)$, where all functions used in pointwise applications are explicitly defined using existential formulas over the reals;[3]
- an existential formula χ with free variables in $\mathrm{FV}(\mathcal{S}, e, \sigma)$ (see Theorem 7.1 for the definition of $\mathrm{FV}(\mathcal{S}, e, \sigma)$).

---

[3] These are the functions whose graph is a semi-algebraic set [6].

The problem asks if there exists an instance I conforming to $\mathcal{S}$ by σ and a matrix $B \in e(I)$ such that (I, B) satisfies χ. □

For example, χ may completely specify the matrices in I by giving the values of the entries as rational numbers, and may express that the output matrix has at least one nonzero entry.

An input $(\mathcal{S}, e, \sigma, \chi)$ is a yes-instance to the intensional evaluation problem precisely when the existential sentence $\exists \mathrm{FV}(\mathcal{S}, e, \sigma)(\psi_e \wedge \chi)$ is true in the reals, where ψ_e is the formula obtained by Theorem 7.1. Hence we can conclude:

Corollary 7.3. *The intensional evaluation problem for* MATLANG + eigen *belongs to* $\exists\mathbf{R}$. □

Since the full first-order theory of the reals is decidable, our theorem implies many other decidability results. We give just two examples.

Corollary 7.4. *The equivalence problem for input-sized expressions is decidable. This problem takes as input two input-sized expressions $(\mathcal{S}, e_1, \sigma)$ and $(\mathcal{S}, e_2, \sigma)$ (with the same $\mathcal{S}$ and σ) and asks if for all instances I conforming to $\mathcal{S}$ by σ, we have $B \in e_1(I) \iff B \in e_2(I)$.* □

Note that the equivalence problem for MATLANG expressions on arbitrary instances (size not fixed) is undecidable by Theorem 4.3, since equivalence of FO^3 formulas over binary relational vocabularies is undecidable [27].

Corollary 7.5. *The determinacy problem for input-sized expressions is decidable. This problem takes as input an input-sized expression $(\mathcal{S}, e, \sigma)$ and asks if for every instance I conforming to $\mathcal{S}$ by σ, there exists at most one $B \in e(I)$.* □

Corollary 7.3 gives an $\exists\mathbf{R}$ upper bound on the combined complexity of query evaluation [65]. Our final result is a matching lower bound, already for data complexity alone.

Theorem 7.6. *There exists a fixed schema $\mathcal{S}$ and a fixed expression e in* MATLANG + eigen, *well-typed over $\mathcal{S}$, such that the following problem is hard for $\exists\mathbf{R}$: Given an integer instance I over $\mathcal{S}$, decide whether the zero matrix is a possible result of $e(I)$. The pointwise applications in e use only simple functions definable by quantifier-free formulas over the reals (representing complex numbers as pairs of reals).*

Proof. The feasibility problem [59] takes as input an equation $p = 0$, with p a multivariate polynomial with integer coefficients, and asks whether the equation has a solution over the reals. We may assume that p is given in "standard form", as a sum of terms of the form $a \cdot \mu$ where a is an integer and μ is a monomial [47]. The feasibility problem is known to be complete for $\exists\mathbf{R}$. We will design a schema $\mathcal{S}$ and an expression e so that the feasibility problem reduces in polynomial time to our problem.

We use a construction by Valiant [63] in which a polynomial p is converted into a directed, edge-weighted graph G. The fundamental property of Valiant's construction is that the determinant of the adjacency matrix A of G equals p. Let p be a polynomial in normal form $\sum_{\mu \in M} a_\mu \cdot \mu$ for some set M of monomials. The *length* $|\mu|$ of a monomial μ is the number of multiplications used in the monomial. Similarly, $|a_\mu \cdot \mu| = 1 + |\mu|$ and the length $|p|$ of p is given by $\sum_{\mu \in M}(1 + |\mu|) + |M| - 1$, where we also account for the number of additions. The *size* $\|p\|$ of p is $|p| \cdot \log_2(m)$ where m is an upper bound on the maximum number of variables and the largest integer coefficient in p. In general, Valliant's construction results in a graph of at most $|p| + 2$ vertices. Furthermore, the edge weights in G are coefficients or variables from p, or the value 1. Similarly, the entries in A are zero or edge weights from G. The computation of the graph G and its adjacency matrix A require polynomial time in $\|p\|$.

$$A = \begin{bmatrix} 0 & 1 & 0 & 3 & 5 & 0 & 0 & 0 \\ 0 & 1 & x & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & y & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & y & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & y & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & z \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad Coef = \begin{bmatrix} 0 & 1 & 0 & 3 & 5 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad Enc = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

$$Vars = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad v = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \quad V = \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ v_1 & v_1 & v_1 & v_1 & v_1 & v_1 & v_1 & v_1 \\ v_2 & v_2 & v_2 & v_2 & v_2 & v_2 & v_2 & v_2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ v_2 & v_2 & v_2 & v_2 & v_2 & v_2 & v_2 & v_2 \\ v_2 & v_2 & v_2 & v_2 & v_2 & v_2 & v_2 & v_2 \\ v_3 & v_3 & v_3 & v_3 & v_3 & v_3 & v_3 & v_3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{Enc \cdot v \cdot \mathbf{1}(Coef)^*}$$

$$\underbrace{\begin{bmatrix} 0 & 1 & 0 & 3 & 5 & 0 & 0 & 0 \\ 0 & 1 & v_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & v_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & v_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & v_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & v_3 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{A^{(v)}} = \underbrace{\begin{bmatrix} 0 & 1 & 0 & 3 & 5 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{Coef} + \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & v_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & v_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & v_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & v_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & v_3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{\mathrm{apply}[g](Vars, V)}$$

Fig. 4. Construction of matrix $A^{(v)}$ using matrices *Coef*, *Vars*, *Enc*. The pointwise function $g : \mathbf{C}^2 \to \mathbf{C}$ is defined as $g(x, y)$ is y if $x = 1$, and zero otherwise. The matrix $A^{(v)}$ is such that $\det(A^{(v)}) = 0$ if and only if $p(v_1, v_2, v_3) = 0$ for polynomial $p(x, y, z) = 3 + 1xy + 5y^2z$. This follows from the fact that for the symbolic matrix A, $\det(A) = p(x, y, z)$.

The construction has a specific property: when p is given in standard form, with an explicit coefficient before each monomial (even if it is merely the value 1), each row of A contains at most one variable. This property is important for the expression e, specified below, to work.

Example 7.7 (Valiant's construction). Consider the polynomial $p(x, y, z) = 3 + 1xy + 5y^2z$ given in standard form in which each monomial has an associated coefficient. Following the construction by Valiant [63], the symbolic matrix A shown in Figure 4 is such that $\det(A) = p(x, y, z)$. ☐

Assume G has nodes $1, \ldots, n$, and let the variables in p be $x_1, \ldots, x_k$. We represent A by three integer matrices *Coef*, *Vars*, and *Enc*. Matrix *Coef* is the $n \times n$ matrix obtained from A by omitting the variable entries (these are set to zero). On the other hand, *Vars*, also $n \times n$, is obtained from A by keeping only the variable entries, but setting them to 1. All other entries are set to zero. Finally, *Enc* encodes which variables are represented by the one-entries in *Vars*. Specifically, *Enc* is the $n \times k$

matrix where $Enc_{i,j} = 1$ if the ith row of A contains variable x_j, and zero otherwise. In Figure 4 we depict these matrices for our example polynomial $p(x, y, z) = 3 + 1xy + 5y^2z$.

We thus reduce an input $p = 0$ of the feasibility problem to the instance I consisting of the matrices $Coef$, $Vars$, Enc. Additionally, for technical reasons, I also has the $k \times 1$ column vector F, which has value 1 in its first entry and is zero everywhere else. Formally, this instance is over the fixed schema $\mathcal{S}$ consisting of the matrix variables M_{Coef}, M_{Vars}, M_{Enc}, and M_F, where the first two variables have type $\alpha \times \alpha$; the third variable has type $\alpha \times \beta$; and M_F has type $\beta \times 1$. To reduce clutter, however, in what follows we will write these variables simply as $Coef$, $Vars$, Enc, and F.

We must now give an expression e that has the zero matrix as possible result of $e(I)$ if and only if $p = 0$ has a solution over the reals. For any $k \times 1$ vector v of real numbers, let $A^{(v)}$ denote the matrix A where we have substituted the entries of v for the variables $x_1, \ldots, x_k$. By Valiant's construction, the expression e should return the zero matrix as a possible result, if and only if there exists a v such that $A^{(v)}$ has determinant zero, i.e., is not invertible.

The desired expression e works as follows. We apply eigen to the $k \times k$ zero matrix, which we compute as O in the expression given below. By selecting the first column of the result, we can nondeterministically obtain all possible nonzero $k \times 1$ column vectors. Taking only the real part ($\mathfrak{R}$) of the entries, we obtain all possible real column vectors v. Then the matrix $A^{(v)}$ is assembled (in matrix variable AA) using the matrices $Coef$, $Vars$, and Enc. Finally, we apply inv to AA so that the zero matrix is returned if and only if AA has determinant zero.

In conclusion, expression e reads as follows:

$$\begin{aligned}
&\mathsf{let}\ O = \mathsf{apply}[0](F \cdot F^*)\ \mathsf{in}\\
&\mathsf{let}\ B = \mathsf{eigen}(O)\ \mathsf{in}\\
&\mathsf{let}\ v = \mathsf{apply}[\mathfrak{R}](B \cdot F)\ \mathsf{in}\\
&\mathsf{let}\ V = Enc \cdot v \cdot \mathbf{1}(Coef)^*\ \mathsf{in}\\
&\mathsf{let}\ AA = \mathsf{apply}[+](Coef, \mathsf{apply}[g](Vars, V))\ \mathsf{in}\\
&\mathsf{inv}(AA)
\end{aligned}$$

Here, in the last expression, $g(x, y)$ is y if $x = 1$, and zero otherwise. In Figure 4 we illustrate the resulting matrices for V and AA (i.e., $A^{(v)}$) for our example polynomial. □

Remark (Complexity of deterministic expressions). Our proof of Theorem 7.6 relies on the nondeterminism of the eigen operation. In particular, we use the eigen operation to non-deterministically select an $n \times 1$-vector from all possible complex $n \times 1$ vectors. The hardness therefore holds for *any* extension of MATLANG with an operation choice(·) which non-deterministically chooses a complex vector, whose dimensions could, for example, be determined by the dimension of the input column vector of this operation. For example, in the expression e at the end of the proof of Theorem 7.6, we could eliminate the use of the eigen operation by simply replacing the first two lines by $B = \mathsf{choice}(A)$.

Remark. Coming back to our remark on determinacy at the end of the previous section, it is an interesting question for further research to understand not only the expressive power but also the complexity of the evaluation problem for *deterministic* MATLANG + eigen expressions.

8 EXPERIMENTS ON COMPUTING THE TRANSITIVE CLOSURE

We have seen that various natural matrix manipulations are expressible in our matrix query languages. Each such expression in turn directly corresponds to a possible implementation in terms of the primitives of MATLANG, MATLANG + inv or MATLANG + eigen. However, this

Algorithm	Progr. lang.	2^8 nodes	2^9 nodes	2^{10} nodes	2^{11} nodes
Tarjan	SageMath	30.2 ms	122 ms	516 ms	2.16 s
Matrix inversion (Ex. 5.2)	R	17 ms	132 ms	691 ms	4.91 s
	SageMath	280 ms	1.66 s	3.24 s	15.7 s
Furman	R	91 ms	346 ms	2.58 s	20.9 s
	SageMath	370 ms	2.15 s	12.0 s	70.6 s
Floyd-Warshall	R	4.14 s	38.6 s	383 s	> 1 h
	SageMath	30.4 s	476 s	> 1 h	> 1 h

Table 1. Running times (best of three runs) of transitive closure algorithms on random dense graphs implemented in R or SageMath. Hardware setup: Lenovo ThinkCentre E71 with Intel Pentium CPU G630 at 2.70 GHz.

implementation may not be optimal for practical purposes. In this section we report on a preliminary experimental investigation assessing the efficiency of the MATLANG + inv expression given in Example 5.2 which computes the transitive closure of a graph given its adjacency matrix A.

We have implemented the algorithm corresponding to this expression in a straightforward way in both R and SageMath (which is an open source competitor of MATLAB), and we have compared this algorithm to three other algorithms: (1) Furman's algorithm [22] which first computes $A := A + A^2$ a number of times logarithmic in the number of vertices and then sets all nonzero entries to 1; (2) Floyd-Warshall's algorithm; and (3) an algorithm [67] based on Tarjan's algorithm that computes the strongly connected components of a graph. It is known that algorithms based on Tarjan's algorithm perform best (especially for sparse graphs) [50, 51], and, indeed, our modest computer experiments on random dense graphs with up to 2^{11} nodes show that our tested implementation based on Tarjan's algorithm is significantly faster than the other algorithms, cf. Table 1. Our implementation corresponding to the MATLANG + inv expression turns out to be faster than the algorithms based on Furman's algorithm and Floyd-Warshall's algorithm. The inversion-based algorithm performs especially well in R, since R invokes the LAPACK library for fast computation of matrix inversion, which is the dominating step of the algorithm. Moreover, the expression from Example 5.2 corresponds to a matrix level (as opposed to matrix-entry level) program that is very easy to write in R and SageMath.

9 CONCLUSION

There is a commendable trend in contemporary database research to leverage, and considerably extend, techniques from database query processing and optimization, to support large-scale linear algebra computations. In principle, data scientists could then work directly in SQL or related languages. Still, some users will prefer to continue using the matrix languages they are more familiar with. Supporting these languages is also important so that existing code need not be rewritten. As already discussed in Section 2, the optimization and efficient processing of matrix query expressions is a rich area for further research.

In this paper we have proposed a framework for viewing matrix manipulation from the point of view of expressive power of database query languages. Moreover, our results formally confirm that the basic set of matrix operations offered by systems in practice, formalized here in the language MATLANG + inv + eigen, really is adequate for expressing a range of linear algebra techniques and procedures.

In the paper we have already mentioned some intriguing questions for further research. Deep inexpressibility results have been developed for logics with rank operators [54]. Although these results are mainly concerned with finite fields, they might still provide valuable insight in our open questions. Also, we have not covered all standard constructs from linear algebra. For instance, it may be worthwhile to extend our framework with the operation of putting matrices in upper triangular form, with the Gram-Schmidt procedure (which is now partly hidden in the eigen operation), and with the singular value decomposition.

Furthermore, as suggested by an anonymous referee, it may be fruitful to make connections to circuit complexity classes. Thus, MATLANG may be compared to the complexity class TC^0, and MATLANG + inv to the complexity class DET. Note, however, that these complexity classes assume the bit model of computation, whereas our presentation of MATLANG has been over arbitrary complex numbers.

Finally, we note that various authors have proposed to go beyond matrices, introducing data models and algebra for tensors or multidimensional arrays [36, 55, 56]. When moving to more and more powerful and complicated languages, however, it becomes less clear at what point we should simply move all the way to full SQL, or extensions of SQL with recursion.

ACKNOWLEDGMENTS

We thank Bart Kuijpers for telling us about the complexity class $\exists\mathbf{R}$. We also thank Lauri Hella and Wied Pakusa for helpful discussions, and Christoph Berkholz and Anuj Dawar for their help with the proof of Proposition 4.4. Finally, we thank the anonymous referees for their insightful comments, which we have used to improve the paper. R.B. is a postdoctoral fellow of the Research Foundation – Flanders (FWO).

REFERENCES

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

[2] M. Abo Khamis, H.Q. Ngo, and A. Rudra. 2016. FAQ: Questions Asked Frequently. In *Proceedings 35th ACM Symposium on Principles of Database Systems*, T. Milo and W.-C. Tan (Eds.). ACM, 13–28.

[3] D.S. Arnon. 1988. Geometric reasoning with logic and algebra. *Artificial Intelligence* 37 (1988), 37–60.

[4] S. Axler. 2015. *Linear Algebra Done Right* (third ed.). Springer.

[5] S. Basu, R. Pollack, and M.-F. Roy. 2008. *Algorithms in Real Algebraic Geometry* (second ed.). Springer.

[6] J. Bochnak, M. Coste, and M.-F. Roy. 1998. *Real Algebraic Geometry*. Springer-Verlag.

[7] M. Boehm, M.W. Dusenberry, D. Eriksson, A.V. Evfimievski, F.M. Manshadi, N. Pansare, B. Reinwald, F.R. Reiss, P. Sen, A.C. Surve, and S. Tatikonda. 2016. SystemML: Declarative machine learning on Spark. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1425–1436.

[8] A. Bonato. 2008. *A Course on the Web Graph*. Graduate Studies in Mathematics, Vol. 89. American Mathematical Society.

[9] R. Brijder, F. Geerts, J. Van den Bussche, and T. Weerwag. 2018. On the Expressive Power of Query Languages for Matrices. In *Proceedings 21st International Conference on Database Theory (LIPIcs)*, Vol. 98. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 10:1–10:17.

[10] R. Brijder, M. Gyssens, and J. Van den Bussche. 2019. On matrices and K-relations. arXiv:1904.03934.

[11] S. Brin and L. Page. 1998. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems* 30 (1998), 107–117.

[12] P.G. Brown. 2010. Overview of sciDB: Large Scale Array Storage, Processing and Analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, 963–968.

[13] J.-Y. Cai, M. Fürer, and N. Immerman. 1992. An optimal lower bound on the number of variables for graph identification. *Combinatorica* 12, 4 (1992), 389–410.

[14] J. Canny. 1988. Some Algebraic and Geometric Computations in PSPACE. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing (STOC '88)*. ACM, 460–467.

[15] L. Chen, A. Kumar, J. Naughton, and J.M. Patel. 2017. Towards linear algebra over normalized data. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1214–1225.

[16] S. Datta, R. Kulkarni, A. Mukherjee, T. Schwentick, and T. Zeume. 2015. Reachability is in DynFO. In *Proceedings 42nd International Colloquium on Automata, Languages and Programming, Part II (Lecture Notes in Computer Science)*, M.M. Halldórsson, K. Iwama, N. Kobayashi, and B. Speckmann (Eds.), Vol. 9135. Springer, 159–170.

[17] S. Datta, R. Kulkarni, A. Mukherjee, T. Schwentick, and T. Zeume. 2018. Reachability Is in DynFO. *J. ACM* 65, 5 (2018), 33:1–33:24.

[18] A. Dawar. 2008. On the descriptive complexity of linear algebra. In *Logic, Language, Information and Computation, Proceedings 15th WoLLIC (Lecture Notes in Computer Science)*, W. Hodges and R. de Queiroz (Eds.), Vol. 5110. Springer, 17–25.

[19] A. Dawar, M. Grohe, B. Holm, and B. Laubner. 2009. Logics with rank operators. In *Proceedings 24th Annual IEEE Symposium on Logic in Computer Science*. 113–122.

[20] A. Dawar and B. Holm. 2017. Pebble games with algebraic rules. *Fundamenta Informaticae* 150, 3-4 (2017), 281–316.

[21] G.M. Del Corso, A. Gulli, and F. Romani. 2005. Fast PageRank computation via a sparse linear system. *Internet Mathematics* 2, 3 (2005), 251–273.

[22] M.E. Furman. 1970. Application of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph. *Soviet Mathematics Doklady* 11, 5 (1970), 1252.

[23] F. Geerts. 2019. On the Expressive Power of Linear Algebra on Graphs. In *Proceedings 22nd International Conference on Database Theory (LIPIcs)*, Vol. 127. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 7:1–7:19.

[24] C.D. Godsil. 1982. Some graphs with characteristic polynomials which are not solvable by radicals. *Journal of Graph Theory* 6 (1982), 211–214.

[25] G.H. Golub and C.F. Van Loan. 2013. *Matrix Computations* (fourth ed.). The Johns Hopkins University Press.

[26] E. Grädel and W. Pakusa. 2015. Rank Logic is Dead, Long Live Rank Logic!. In *24th EACSL Annual Conference on Computer Science Logic (CSL)*. 390–404.

[27] E. Grädel, E. Rosen, and M. Otto. 1999. Undecidability results on two-variable logics. *Archive of Mathematical Logic* 38 (1999), 313–354.

[28] T.J. Green, G. Karvounarakis, and V. Tannen. 2007. Provenance semirings. In *Proceedings 26th ACM Symposium on Principles of Database Systems*. 31–40.

[29] M. Grohe and W. Pakusa. 2017. Descriptive complexity of linear equation systems and applications to propositional proof complexity. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–12.

[30] L. Hella, L. Libkin, J. Nurmonen, and L. Wong. 2001. Logics with aggregate operators. *J. ACM* 48, 4 (2001), 880–907.

[31] J.M. Hellerstein, C. Ré, F. Schoppmann, D.Z. Wang, E. Fratkin, A. Gorajek, K.S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. 2012. The MADlib Analytics Library: Or MAD Skills, the SQL. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1700–1711.

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

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

[34] K.E. Iverson. 1962. *A Programming Language*. John Wiley & Sons, Inc.

[35] P.C. Kanellakis, G.M. Kuper, and P.Z. Revesz. 1995. Constraint query languages. *J. Comput. System Sci.* 51, 1 (Aug. 1995), 26–52.

[36] M. Kim. 2014. *TensorDB and Tensor-Relational Model for Efficient Tensor-Relational Operations*. Ph.D. Dissertation. Arizona State University.

[37] A. Klug. 1982. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM* 29, 3 (1982), 699–717.

[38] Ph.G. Kolaitis. 2007. On the expressive power of logics on finite models. In *Finite Model Theory and Its Applications*. Springer, Chapter 2, 27–123.

[39] A. Kunft, A. Alexandrov, A. Katsifodimos, and V. Markl. 2016. Bridging the Gap: Towards Optimization Across Linear and Relational Algebra. In *Proceedings 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*. 1:1–1:4.

[40] G. Kuper, L. Libkin, and J. Paredaens (Eds.). 2000. *Constraint Databases*. Springer.

[41] B. Laubner. 2010. *The Structure of Graphs and New Logics for the Characterization of Polynomial Time*. Ph.D. Dissertation. Humboldt-Universität zu Berlin.

[42] J. Leskovec, A. Rajaraman, and J.D. Ullman. 2014. *Mining of Massive Datasets* (second ed.). Cambridge University Press.

[43] L. Libkin. 2003. Expressive power of SQL. *Theoretical Computer Science* 296 (2003), 379–404.

[44] S. Luo, Z.J. Gao, M. Gubanov, L.L. Perez, and C. Jermaine. 2018. Scalable Linear Algebra on a Relational Database System. *SIGMOD Record* 47, 1 (2018), 24–31.

[45] S. Luo, Z.J. Gao, M.N. Gubanov, L. Leopoldo Perez, and C.M. Jermaine. 2017. Scalable Linear Algebra on a Relational Database System. In *Proceedings 33rd International Conference on Data Engineering*. IEEE Computer Society, 523–534.

[46] M. Marx and Y. Venema. 1997. *Multi-Dimensional Modal Logic*. Springer.

[47] J. Matoušek. 2014. Intersection graphs of segments and $\exists\mathbb{R}$. arXiv:1406.2636.

[48] Microsoft SQL Server R Services. 2019.

[49] H.Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. 2017. In-database factorized learning. In *Proceedings 11th Alberto Mendelzon International Workshop on Foundations of Data Management (CEUR Workshop Proceedings)*, J.L. Reutter and D. Srivastava (Eds.), Vol. 1912.

[50] E. Nuutila. 1994. An Efficient Transitive Closure Algorithm for Cyclic Digraphs. *Inform. Process. Lett.* 52, 4 (1994), 207–213.

[51] E. Nuutila. 1995. *Efficient Transitive Closure Computation in Large Digraphs.* Ph.D. Dissertation. Helsinki University of Technology.

[52] Oracle R Enterprise. 2019.

[53] M. Otto. 1997. *Bounded Variable Logics and Counting: A Study in Finite Models.* Lecture Notes in Logic, Vol. 9. Springer.

[54] W. Pakusa. 2015. *Linear Equation Systems and the Search for a Logical Characterisation of Polynomial Time.* Ph.D. Dissertation. RWTH Aachen.

[55] F. Rusu and Y. Cheng. 2013. A survey on array storage, query languages, and systems. arXiv:1302.0103.

[56] T. Sato. 2017. Embedding Tarskian semantics in vector spaces. arXiv:1703.03193.

[57] T. Sato. 2017. A linear algebra approach to datalog evaluation. *Theory and Practice of Logic Programming* 17, 3 (2017), 244–265.

[58] M. Schaefer. 2009. Complexity of some geometric and topological problems. In *Graph Drawing (Lecture Notes in Computer Science)*, D. Eppstein and E.R. Gansner (Eds.), Vol. 5849. Springer, 334–344.

[59] M. Schaefer and D. Štefankovič. 2017. Fixed points, Nash equilibria, and the existential theory of the reals. *Theory of Computing Systems* 60, 2 (2017), 172–193.

[60] M. Schleich, D. Olteanu, and R. Ciucanu. 2016. Learning linear regression models over factorized joins. In *Proceedings 2016 International Conference on Management of Data.* ACM, 3–18.

[61] A. Tarski and S. Givant. 1987. *A Formalization of Set Theory Without Variables.* AMS Colloquium Publications, Vol. 41. American Mathematical Society.

[62] A. Thomas and A. Kumar. 2018. A Comparative Evaluation of Systems for Scalable Linear Algebra-based Analytics. *Proceedings of the VLDB Endowment* 11, 13 (2018), 2168–2182.

[63] L.G. Valiant. 1979. Completeness classes in algebra. In *Proceedings 11th ACM Symposium on Theory of Computing.* 249–261.

[64] J. Van den Bussche, D. Van Gucht, and S. Vansummeren. 2007. A crash course in database queries. In *Proceedings 26th ACM Symposium on Principles of Database Systems.* ACM Press, 143–154.

[65] M. Vardi. 1982. The Complexity of Relational Query Languages. In *Proceedings 14th ACM Symposium on the Theory of Computing.* 137–146.

[66] D. Wagner and F. Wagner. 1993. Between Min Cut and Graph Bisection. In *Proceedings 18th International Symposium on Mathematical Foundations of Computer Science*, A. M. Borzyszkowski and S. Sokołowski (Eds.). Springer Berlin Heidelberg, 744–750.

[67] B. Westerbaan. 2016. Python implementation of Tarjan's algorithm. https://pypi.org/project/tarjan/

[68] P. Wood. 2012. Query Languages for Graph Databases. *SIGMOD Record* 41, 1 (March 2012), 50–60.

[69] Y. Zhang, W. Zhang, and J. Yang. 2010. I/O-efficient statistical computing with RIOT. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010).* 1157–1160.

A PROOF OF PROPOSITION 4.2

Let us assign, to each MATLANG expression e that is well-typed over $\mathcal{S}$, an expression φ_e in the relational calculus with summation as follows. As before, since the let operation is syntactic sugar for MATLANG expressions, we do not consider this operation in this proof.

- If $e = M$ is a matrix variable of $\mathcal{S}$, then $\varphi_e(i, j, x) := \mathit{Rel}_2(M)(i, j, x)$ if M is of general type, $\varphi_e(i, x) := \mathit{Rel}_1(M)(i, x)$ if M is of vector type, and $\varphi_e(x) := \mathit{Rel}_0(M)(x)$ if M is of scalar type.

Let e' be a MATLANG and let $\tau = s_1 \times s_2$ be the output type of e'.

- If $e = (e')^*$, then $\varphi_e(i, j, x) := \exists x' \, (\varphi_{e'}(j, i, x') \wedge x = \overline{x'})$ if τ is a general type, $\varphi_e(i, x) := \exists x' \, (\varphi_{e'}(i, x') \wedge x = \overline{x'})$ if τ is a vector type, and $\varphi_e(x) := \exists x' \, (\varphi_{e'}(x') \wedge x = \overline{x'})$ if τ is the scalar type. Here, $\overline{x}$ denotes the complex conjugate operation.
- If $e = \mathbf{1}(e')$, then $\varphi_e(i, x) := \exists j, x' \, (\varphi_{e'}(i, j, x') \wedge x = 1(x'))$ if τ is a general type, $\varphi_e(i, x) := \exists x' \, (\varphi_{e'}(i, x') \wedge x = 1(x'))$ is a vector type and $s_1 \neq 1 = s_2$, $\varphi_e(x) := \exists i, x' \, (\varphi_{e'}(i, x') \wedge x = 1(x'))$ is a vector type and $s_1 = 1 \neq s_2$, and $\varphi_e(x) := \exists x' \, (\varphi_{e'}(x') \wedge x = 1(x'))$ if τ is the scalar type. As before, 1 in the expression φ_e is the constant 1 function.

- If $e = \mathrm{diag}(e')$, then $\varphi_e(i,j,x) := (\varphi_{e'}(i,x) \wedge j = i) \vee (\exists x', x''\; \varphi_{e'}(i,x') \wedge \varphi_{e'}(j,x'') \wedge i \neq j \wedge x = 0(x'))$ if $s_1 \neq 1 = s_2$ and $\varphi_e(x) := \varphi_{e'}(x)$ if τ is the scalar type.
- If $e = e_1 \cdot e_2$ where e_1 is of type $s_1 \times s_3$ and e_2 is of type $s_3 \times s_2$, then

$$\begin{cases} \varphi_e(i,j,z) := z = \mathsf{sum}\, k,x,y.(\varphi_{e_1}(i,k,x) \wedge \varphi_{e_2}(k,j,y), x \times y) & \text{if } s_1 \neq 1 \neq s_2 \text{ and } s_3 \neq 1; \\ \varphi_e(i,z) := z = \mathsf{sum}\, k,x,y.(\varphi_{e_1}(i,k,x) \wedge \varphi_{e_2}(k,y), x \times y) & \text{if } s_1 \neq 1 = s_2 \text{ and } s_3 \neq 1; \\ \varphi_e(i,z) := z = \mathsf{sum}\, k,x,y.(\varphi_{e_1}(k,x) \wedge \varphi_{e_2}(k,i,y), x \times y) & \text{if } s_1 = 1 \neq s_2 \text{ and } s_3 \neq 1; \\ \varphi_e(z) := z = \mathsf{sum}\, k,x,y.(\varphi_{e_1}(k,x) \wedge \varphi_{e_2}(k,y), x \times y) & \text{if } s_1 = 1 = s_2 \text{ and } s_3 \neq 1; \\ \varphi_e(i,j,z) := \varphi_{e_1}(i,x) \wedge \varphi_{e_2}(j,y) \wedge z = x \times y & \text{if } s_1 \neq 1 \neq s_2 \text{ and } s_3 = 1; \\ \varphi_e(i,z) := \varphi_{e_1}(i,x) \wedge \varphi_{e_2}(y) \wedge z = x \times y & \text{if } s_1 \neq 1 = s_2 \text{ and } s_3 = 1; \\ \varphi_e(i,z) := \varphi_{e_1}(x) \wedge \varphi_{e_2}(i,y) \wedge z = x \times y & \text{if } s_1 = 1 \neq s_2 \text{ and } s_3 = 1; \\ \varphi_e(z) := \varphi_{e_1}(x) \wedge \varphi_{e_2}(y) \wedge z = x \times y & \text{if } s_1 = 1 = s_2 \text{ and } s_3 = 1. \end{cases}$$

- If $e = \mathrm{apply}[f](e_1, \dots, e_n)$, then

$$\begin{gathered} \varphi_e(i,j,x) := \exists x_1, \dots, x_n\, (\varphi_{e_1}(i,j,x_1) \wedge \cdots \wedge \varphi_{e_n}(i,j,x_n) \wedge x = f(x_1, \dots, x_n)), \\ \varphi_e(i,x) := \exists x_1, \dots, x_n\, (\varphi_{e_1}(i,x_1) \wedge \cdots \wedge \varphi_{e_n}(i,x_n) \wedge x = f(x_1, \dots, x_n)), \text{ and} \\ \varphi_e(x) := \exists x_1, \dots, x_n\, (\varphi_{e_1}(x_1) \wedge \cdots \wedge \varphi_{e_n}(x_n) \wedge x = f(x_1, \dots, x_n)) \end{gathered}$$

  depending on whether τ is of general, vector or scalar type, respectively.

Notice that the only functions in φ_e aside from those used in apply in e are complex conjugation ($\bar{z}$), multiplication of two numbers ($\times$), and the constant functions 0 and 1. Also notice that φ_e uses neither negation, nor equality conditions on numerical variables, nor equality conditions on variables involving a constant.

By induction on the structure of e one straightforwardly observes that φ_e satisfies the conditions (1) and (2) in the statement of the theorem. Furthermore, it is clear for all operations except for matrix multiplication that when $\varphi_{e'}$ (or the φ_{e_i}'s in the case of apply) uses at most 3 base variables than so does φ_e. When it comes to matrix multiplication, assume that $\varphi_{e_1}(i,k,x)$ uses base variables i, j', k and $\varphi_{e_2}(k,j,y)$ uses base variables i', j, k. Since j' is not free in $\varphi_{e_1}(i,k,x)$, we can rename j' to j. Similarly, we can rename i' to i in $\varphi_{e_2}(k,j,y)$. In this way, $\varphi_e(i,j,z) := z = \mathsf{sum}\, k,x,y.(\varphi_{e_1}(i,k,x) \wedge \varphi_{e_2}(k,j,y), x \times y)$ uses at most 3 base variables as well (the cases where not all types are general is similar). ☐