Matrices As Arrows!
A Biproduct Approach to Typed Linear Algebra

Hugo Daniel Macedo[1] and José Nuno Oliveira[2]

[1] Minho University, Portugal
hmacedo@di.uminho.pt
[2] Minho University, Portugal
jno@di.uminho.pt

Abstract. Motivated by the need to formalize generation of fast running code for linear algebra applications, we show how an index-free, calculational approach to matrix algebra can be developed by regarding matrices as morphisms of a category with biproducts. This shifts the traditional view of matrices as indexed structures to a type-level perspective analogous to that of the pointfree algebra of programming. The derivation of fusion, cancellation and abide laws from the biproduct equations makes it easy to calculate algorithms implementing matrix multiplication, the kernel operation of matrix algebra, ranging from its divide-and-conquer version to the conventional, iterative one.
From errant attempts to learn how particular products and coproducts emerge from biproducts, we not only rediscovered block-wise matrix combinators but also found a way of addressing other operations calculationally such as e.g. Gaussian elimination. A strategy for addressing vectorization along the same lines is also given.

1 Introduction

Automatic generation of fast running code for linear algebra applications calls for matrix multiplication as kernel operator, whereby matrices are viewed and transformed in an index-free way [1]. Interestingly, the successful language SPL [2] used in generating automatic parallel code has been created envisaging the same principles as advocated by the purist computer scientist: index-free abstraction and composition (multiplication) as a kernel way of connecting objects of interest (matrices, programs, etc).

There are several domain specific languages (DSLs) bearing such purpose in mind [2, 3, 1]. However, they arise as programming dialects with poor type checking. This may lead to programming errors, hindering effective use of such languages and calling for a "type structure" in linear algebra systems similar to that underlying modern functional programming languages such as Haskell, for instance [4].

It so happens that, in the same way function composition is the kernel operation of functional programming, leading to the *algebra of programming* [5], so does matrix multiplication once matrices are viewed and transformed in an

index-free way. So, rather than interpreting the product AB of matrices A and B as an algorithm for computing a new matrix C out of A and B, and trying to build and explain matrix algebra systems out of such an algorithm, one wishes to abstract from *how* the operation is carried out. Instead, the emphasis is put on its type structure, regarded as the pipeline $A \cdot B$ (to be read as "A after B"), as if A and B were functions

$$C = A \cdot B \tag{1}$$

or binary relations — the actual building block of the algebra of programming [5]. In this discipline, relations are viewed as (typed) composable arrows (morphisms) which can be combined in a number of ways, namely by joining or intersecting relations of the same type, reversing them (thus swapping their source and target types), and so on.

If relations, which are Boolean matrices, can be regarded as morphisms of a suitable mathematical framework, why not regard arbitrary matrices in the same way? This matches with the categorical characterization of matrices, which can be traced back to Mac Lane [6], whereby matrices are regarded as arrows in a category whose objects are natural numbers (matrix dimensions):

$$A = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix}_{m \times n} \qquad m \xleftarrow{A} n \tag{2}$$

Such a category $\mathbf{Mat}_K$ of matrices over a field K merges categorical products and coproducts into a single construction termed *biproduct* [6]. Careful analysis of the biproduct axioms as a system of equations provides one with a rich *palette* of constructs for building matrices from smaller ones. In [7], for instance, we outlined an approach to matrix blocked operation stemming from one particular solution to such equations, which in fact offers explicit operators for building block-wise matrices (row and column-wise) as defined by [8].

In the current paper we elaborate on [7] and show in detail how block-driven divide-and-conquer algorithms for linear algebra arise from biproduct laws emerging from the underlying categorial basis. In summary, this paper gives the details of a constructive approach to matrix algebra operations leading to elegant, index-free proofs of the corresponding algorithms. As happens with state-of-the-art algebra of programming, the whole framework is fully typed, enabling parametric type checking of matrix combinators.

2 The Category of Matrices $\mathbf{Mat}_K$

Matrices are mathematical objects that can be traced back to ancient times, documented as early as 200 BC [9]. The word "matrix" was introduced in the western culture much later, in the 1840's, by the mathematician James Sylvester (1814-1897) when both matrix theory and linear algebra emerged.

The traditional way of viewing matrices as rectangular tables (2) of elements or entries (the "container view") which in turn are other mathematical objects

such as e.g. complex numbers (in general: inhabitants of the field K which underlies $\mathbf{Mat}_K$), encompasses as special cases one column and one line matrices, referred to as column (resp. row) *vectors*, that is, matrices of shapes

$$v = \begin{bmatrix} v_1 \\ \vdots \\ v_m \end{bmatrix} \qquad \text{and} \qquad w = \begin{bmatrix} w_1 \ \ldots \ w_n \end{bmatrix}$$

What is a matrix? The standard answer to this question is to regard matrix A (2) as a computation unit, or transformation, which commits itself to producing a (column) vector of size m provided it is supplied with a (column) vector of size n. How is such output produced? Let us abstract from this at this stage and look at diagram

$$m \xleftarrow{A} n \xleftarrow{v} 1 \qquad m \xleftarrow{w} 1$$

arising from depicting the situation in arrow notation. This suggests a pictorial representation of the product of matrix $A_{m\times n}$ and matrix $B_{n\times q}$, yielding a new matrix $C = (AB)_{m\times q}$ with dimensions $m \times q$, as follows,

$$m \xleftarrow{A} n \xleftarrow{B} q \qquad m \xleftarrow{C=A\cdot B} q \tag{3}$$

which automatically "type-checks" the construction: the "target" of $n \xleftarrow{B} q$ simply matches the "source" of $m \xleftarrow{A} n$ yielding a matrix whose type $m \longleftarrow q$ is the composition of the given types.

Having defined matrices as composable arrows in a category, we need to define its identities [6]: for every object n, there must be an arrow of type $n \longleftarrow n$ which is the unit of composition. This is nothing but the identity matrix of size n, which we will denote by $n \xleftarrow{id_n} n$. For every matrix $m \xleftarrow{A} n$, one has

$$id_m \cdot A \;=\; A \;=\; A \cdot id_n \qquad \begin{array}{ccc} n & \xleftarrow{id_n} & n \\ {\scriptstyle A}\downarrow & \swarrow_{A} & \downarrow{\scriptstyle A} \\ m & \xleftarrow[id_m]{} & m \end{array} \tag{4}$$

(Subscripts m and n can be omitted wherever the underlying diagrams are assumed.)

Transposed matrices. One of the kernel operations of linear algebra is *transposition*, whereby a given matrix changes shape by turning its rows into columns and vice-versa. Type-wise, this means converting an arrow $n \xleftarrow{A} m$ into an

arrow $m \xleftarrow{A^T} n$, that is, source and target types (dimensions) switch over. By analogy with relation algebra, where a similar operation is termed *converse* and denoted A°, we will use this notation instead of A^T (which misleadingly suggests a kind of exponential) and will say "A converse" wherever reading A°. Index-wise, we have, for A as in (2):

$$A^\circ = \begin{bmatrix} a_{11} & \dots & a_{m1} \\ \vdots & \ddots & \vdots \\ a_{1n} & \dots & a_{mn} \end{bmatrix} \qquad\qquad n \xleftarrow{A^\circ} m$$

Instead of telling how transposition is carried out index-wise, again we prefer to stress on (index-free) properties of this operation such as, among others, idempotence and contravariance:

$$(A^\circ)^\circ = A \tag{5}$$

$$(A \cdot B)^\circ = B^\circ \cdot A^\circ \tag{6}$$

Bilinearity. Given two matrices of the same type $m \xleftarrow{M,N} n$ (i.e., in the same homset of $\mathbf{Mat}_K$) it makes sense to add them up index-wise, leading to matrix $M + N$ where symbol + promotes the underlying element-level additive operator to matrix-level. In fact, matrices form an *Abelian category*: each homset in the category forms an additive Abelian (ie. commutative) group with respect to which composition is bilinear:

$$M \cdot (N + L) = M \cdot N + M \cdot L \tag{7}$$

$$(N + L) \cdot K = N \cdot K + L \cdot K \tag{8}$$

Polynomial expressions (such as in the properties above) denoting matrices built up in an index-free way from addition and composition play a major role in matrix algebra. This can be appreciated in the explanation of the very important concept of a *biproduct* which follows.

Biproducts. In an Abelian category, a *biproduct* diagram for the objects m, n is a diagram of shape

$$m \underset{i_1}{\overset{\pi_1}{\leftrightarrows}} r \underset{i_2}{\overset{\pi_2}{\leftrightarrows}} n$$

whose arrows π_1, π_2, i_1, i_2 satisfy the identities which follow:

$$\pi_1 \cdot i_1 = id_m \tag{9}$$

$$\pi_2 \cdot i_2 = id_n \tag{10}$$

$$i_1 \cdot \pi_1 + i_2 \cdot \pi_2 = id_r \tag{11}$$

Morphisms π_i and i_i are termed *projections* and *injections*, respectively. From the underlying arithmetics one easily derives the following orthogonality properties

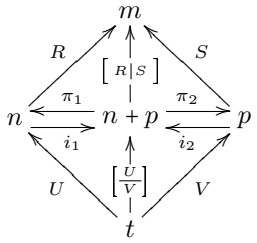
(see e.g. [6]):

$$\pi_1 \cdot i_2 = 0 \quad , \quad \pi_2 \cdot i_1 = 0 \tag{12}$$

One wonders: how do biproducts relate to products and co-products in the category? The answer in Mac Lane's [6] words is as follows:

> *Theorem 2: Two objects a and b in Abelian category A have a product in A iff they have a biproduct in A. Specifically, given a biproduct diagram, the object r with the projections π_1 and π_2 is a product of m and n, while, dually, r with i_1 and i_2 is a coproduct. In particular, two objects m and n have a product in A if and only if they have a coproduct in A.*

The diagram and definitions below depict how products and coproducts arise from biproducts (the product diagram is in the lower half; the upper half is the coproduct one):



$$\left[\, R \middle| S \,\right] = R \cdot \pi_1 + S \cdot \pi_2 \tag{13}$$

$$\left[\frac{U}{V}\right] = i_1 \cdot U + i_2 \cdot V \tag{14}$$

By analogy with the algebra of programming [5], expressions $\left[\, R \middle| S \,\right]$ and $\left[\frac{U}{V}\right]$ will be read "R junc S" and "U split V", respectively. What is the intuition behind these combinators, which come out of the blue in texts such as e.g. [8]? Expressed in terms of definitions (13) and (14), axiom (11) rewrites to both

$$\left[\, i_1 \middle| i_2 \,\right] = id \tag{15}$$

$$\left[\frac{\pi_1}{\pi_2}\right] = id \tag{16}$$

somehow suggesting that the two injections and the two projections "decompose" the identity matrix. On the other hand, each of (15,16) has the shape of a *reflection* corollary [5] of some universal property. Below we derive such a property for $\left[\, R \middle| S \,\right]$,

$$X = \left[\, R \middle| S \,\right] \Leftrightarrow \begin{cases} X \cdot i_1 = R \\ X \cdot i_2 = S \end{cases} \tag{17}$$

from the underlying biproduct equations, by circular implication:

$$\begin{array}{ll} & X = \left[\, R \middle| S \,\right] \\ \Leftrightarrow & \quad \{ \text{ identity (4) ; (13) } \} \end{array}$$

$$
\begin{array}{ll}
& X \cdot id = R \cdot \pi_1 + S \cdot \pi_2 \\
\Leftrightarrow & \quad \{ \ (11) \ \} \\
& X \cdot (i_1 \cdot \pi_1 + i_2 \cdot \pi_2) = R \cdot \pi_1 + S \cdot \pi_2 \\
\Leftrightarrow & \quad \{ \ \text{bilinearity (7)} \ \} \\
& X \cdot i_1 \cdot \pi_1 + X \cdot i_2 \cdot \pi_2 = R \cdot \pi_1 + S \cdot \pi_2 \\
\Rightarrow & \quad \{ \ \text{Leibniz (twice)} \ \} \\
& \begin{cases} (X \cdot i_1 \cdot \pi_1 + X \cdot i_2 \cdot \pi_2) \cdot i_1 = (R \cdot \pi_1 + S \cdot \pi_2) \cdot i_1 \\ (X \cdot i_1 \cdot \pi_1 + X \cdot i_2 \cdot \pi_2) \cdot i_2 = (R \cdot \pi_1 + S \cdot \pi_2) \cdot i_2 \end{cases} \\
\Leftrightarrow & \quad \{ \ \text{bilinearity (8) ; biproduct (9,10) ; orthogonality (12)} \ \} \\
& \begin{cases} X \cdot i_1 + X \cdot i_2 \cdot 0 = R + S \cdot 0 \\ X \cdot i_1 \cdot 0 + X \cdot i_2 = R \cdot 0 + S \end{cases} \\
\Leftrightarrow & \quad \{ \ \text{trivia} \ \} \\
& \begin{cases} X \cdot i_1 = R \\ X \cdot i_2 = S \end{cases} \\
\Rightarrow & \quad \{ \ \text{Leibniz (twice)} \ \} \\
& \begin{cases} X \cdot i_1 \cdot \pi_1 = R \cdot \pi_1 \\ X \cdot i_2 \cdot \pi_2 = S \cdot \pi_2 \end{cases} \\
\Rightarrow & \quad \{ \ \text{Leibniz} \ \} \\
& X \cdot i_1 \cdot \pi_1 + X \cdot i_2 \cdot \pi_2 = R \cdot \pi_1 + S \cdot \pi_2 \\
\Leftrightarrow & \quad \{ \ \text{as shown above} \ \} \\
& X = \left[R | S \right]
\end{array}
$$

The derivation of the universal property of $\left[\frac{U}{V}\right]$,

$$X = \left[\frac{U}{V}\right] \Leftrightarrow \begin{cases} \pi_1 \cdot X = U \\ \pi_2 \cdot X = V \end{cases} \tag{18}$$

is (dually) analogous.

Parallel with relation algebra. Similar to matrix algebra, relation algebra [10, 5] can also be explained in terms of biproducts once morphism addition (11) is interpreted as relational union, object union is disjoint union, i_1 and i_2 as the corresponding injections and π_1,π_2 their converses, respectively. Relational product should not, however, be confused with the *fork* construct [11] in fork

(relation) algebra, which involves pairing. (For this to become a product one has to restrict to functions.)

In the next section we show that the converse relationship (duality) between projections and injections is not a privilege of relation algebra: the most intuitive biproduct solution in the category of matrices also offers such a duality.

3 Chasing biproducts

Let us now address the intuition behind products and coproducts of matrices. This has mainly to do with the interpretation of projections π_1, π_2 and injections i_1,i_2 arising as solutions of biproduct equations (9,10,11). Concerning this, Mac Lane [6] laconically writes:

> *"In other words, the [biproduct] equations contain the familiar calculus of matrices."*

In what way? The answer to this question proved more interesting than it seems at first, because of the multiple solutions arising from a non-linear system of three equations (9,10,11) with four variables. In trying to exploit this freedom we became aware that each solution offers a particular way of putting matrices together via the corresponding "junc" and "split" combinators.

Our inspection of solutions started by reducing the "size" of the objects involved and experimenting with the smaller biproduct depicted below:

$$1 \underset{i_1}{\overset{\pi_1}{\leftrightarrows}} 1 + 1 \underset{i_2}{\overset{\pi_2}{\leftrightarrows}} 1$$

The "puzzle" in this case is more manageable,

$$\begin{cases} \pi_1 \cdot i_1 = [1] \\ \pi_2 \cdot i_2 = [1] \\ i_1 \cdot \pi_1 + i_2 \cdot \pi_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \end{cases}$$

yet the set of solutions is not small. We used the Mathematica software [12] to solve this system by inputting the projections and injections as suitably typed matrices leading to a larger, non-linear system:

$$\begin{cases} \begin{bmatrix} \pi_{11} & \pi_{12} \end{bmatrix} \cdot \begin{bmatrix} i_{11} \\ i_{12} \end{bmatrix} = [1] \\ \begin{bmatrix} \pi_{21} & \pi_{22} \end{bmatrix} \cdot \begin{bmatrix} i_{21} \\ i_{22} \end{bmatrix} = [1] \\ \begin{bmatrix} i_{11} \\ i_{12} \end{bmatrix} \cdot \begin{bmatrix} \pi_{11} & \pi_{12} \end{bmatrix} + \begin{bmatrix} i_{21} \\ i_{22} \end{bmatrix} \cdot \begin{bmatrix} \pi_{21} & \pi_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \end{cases}$$

This was solved using the standard Solve command obtaining the output presented in Figure 1, which offers several solutions. Among these we first picked the

sol = Solve[{pi1.i1 == I1, pi2.i2 == I1, i1.pi1 + i2.pi2 == I2}]
Solve::svars : Equations may not give solutions for all "solve" variables. ⟩⟩
$\left\{\left\{i_{21} \to \frac{1}{\pi_{21}}, i_{22} \to -\frac{\pi_{11} i_{12}}{\pi_{21}}, \pi_{12} \to \frac{1}{i_{12}}, i_{11} \to 0, \pi_{22} \to 0\right\},\right.$
$\left.\left\{i_{21} \to -\frac{\pi_{12} i_{11}}{\pi_{22}}, i_{22} \to \frac{\pi_{22}+\pi_{12}\pi_{21} i_{11}}{(\pi_{22})^2}, \pi_{11} \to \frac{\pi_{22}+\pi_{12}\pi_{21} i_{11}}{\pi_{22} i_{11}}, i_{12} \to -\frac{\pi_{21} i_{11}}{\pi_{22}}\right\}\right\}$

Fig. 1. Fragment of Mathematica script

one which purports the most intuitive reading of the *junc* and *split* combinators — that of simply gluing matrices vertically and horizontally (respectively) with no further computation of matrix entries:

$$\pi_1 = \begin{bmatrix} 1 & 0 \end{bmatrix} \qquad \pi_2 = \begin{bmatrix} 0 & 1 \end{bmatrix}$$
$$i_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad i_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Interpreted in this way, $\left[\frac{R}{S}\right]$ (14) and $\left[\, R | S \,\right]$ (13) are the block gluing matrix operators which we can find in [8]. Our choice of notation — R above S in the case of (14) and R besides S in the case of (13) reflects this semantics.

The obvious generalization of this solution to higher dimensions of the problem leads to the following matrices with identities of size m and n in the appropriate place, so as to properly typecheck:

$$\pi_1 = m \xleftarrow{\left[\, id_m \mid 0 \,\right]} m+n \quad , \quad \pi_2 = n \xleftarrow{\left[\, 0 \mid id_n \,\right]} m+n$$
$$i_1 = m+n \xleftarrow{\left[\frac{id_m}{0}\right]} m \quad , \quad i_2 = m+n \xleftarrow{\left[\frac{0}{id_m}\right]} n$$

By inspection, one immediately infers the same duality found in relation algebra,

$$\pi_1^\circ = i_1 \ , \ \pi_2^\circ = i_2 \tag{19}$$

whereby *junc* (13) and *split* (14) become self dual:

$$\begin{array}{l} \left[\, R | S \,\right]^\circ \\ = \qquad \{ \ (13) \ ; \ (6) \ \} \\ \pi_1^\circ \cdot R^\circ + \pi_2^\circ \cdot S^\circ \end{array}$$

$$= \quad \{ \ (19) \ ; (14) \ \}$$

$$\left[\frac{R^\circ}{S^\circ}\right]$$

This particular solution to the biproduct equations captures what in the literature is meant by *blocked* matrix algebra, a generalization of the standard element-wise operations to sub-matrices, or blocks, leading to *divide-and-conquer* versions of the corresponding algorithms. The next section shows the exercise of deriving such laws, thanks to the algebra which emerges from the universal properties of the block-gluing matrix combinators *junc* (17) and *split* (18). We combine the standard terminology with that borrowed from the algebra of programming [5] to stress the synergy between blocked matrix algebra and relational algebra.

4 Blocked Linear Algebra — calculationally!

Further to reflection laws (15,16), the derivation of the following equalities from universal properties (17,18) is a standard exercise in (high) school algebra, where capital letters A, B, etc. denote suitably typed matrices (the types, ie. dimensions, involved in each equality can be inferred by drawing the corresponding diagram):

- Two "fusion"-laws:

$$C \cdot \left[\, A \middle| B \,\right] = \left[\, C \cdot A \middle| C \cdot B \,\right] \tag{20}$$

$$\left[\frac{A}{B}\right] \cdot C = \left[\frac{A \cdot C}{B \cdot C}\right] \tag{21}$$

- Four "cancellation"-laws:

$$\left[\, A \middle| B \,\right] \cdot i_1 = A \ , \ \left[\, A \middle| B \,\right] \cdot i_2 = B \tag{22}$$

$$\pi_1 \cdot \left[\frac{A}{B}\right] = A \ , \ \pi_2 \cdot \left[\frac{A}{B}\right] = B \tag{23}$$

- Three "abide"-laws [3]: the *junc/split* exchange law

$$\left[\left[\frac{A}{C}\middle|\frac{B}{D}\right]\right] = \left[\left[\frac{A}{C}\right]\middle|\left[\frac{B}{D}\right]\right] = \left[\frac{A}{C}\middle|\frac{B}{D}\right] \tag{24}$$

[3] Neologism "abide" (= "above and beside") was introduced by Richard Bird [13] as a generic name for algebraic laws in which two binary operators written in infix form change place between "above" and "beside", e.g.

$$\frac{a}{b} \times \frac{c}{d} = \frac{a \times c}{b \times d}$$

which tells the equivalence between row-major and column-major construction of matrices (thus the four entry *block* notation on the right), and two *blocked addition* laws:

$$\left[\, A \middle| B \,\right] + \left[\, C \middle| D \,\right] = \left[\, A + C \middle| B + D \,\right] \tag{25}$$

$$\left[\frac{A}{B}\right] + \left[\frac{C}{D}\right] = \left[\frac{A+C}{B+D}\right] \tag{26}$$

The laws above are more than enough for us to derive standard linear algebra rules and algorithms in a calculational way. As an example of their application we provide a simple proof of the rule which underlies *divide-and-conquer* matrix multiplication:

$$\left[\, R \middle| S \,\right] \cdot \left[\frac{U}{V}\right] = R \cdot U + S \cdot V \tag{27}$$

We calculate:

$$\begin{array}{ll} & \left[\, R \middle| S \,\right] \cdot \left[\frac{U}{V}\right] \\ = & \quad \{\ (14)\ \} \\ & \left[\, R \middle| S \,\right] \cdot (i_1 \cdot U + i_2 \cdot V) \\ = & \quad \{\ \text{bilinearity (7)}\ \} \\ & \left[\, R \middle| S \,\right] \cdot i_1 \cdot U + \left[\, R \middle| S \,\right] \cdot i_2 \cdot V \\ = & \quad \{\ +\text{-cancellation (22)}\ \} \\ & R \cdot U + S \cdot V \end{array}$$

As another example, let us show how standard block-wise matrix-matrix multiplication (MMM),

$$\left[\frac{R \middle| S}{U \middle| V}\right] \cdot \left[\frac{A \middle| B}{C \middle| D}\right] = \left[\frac{RA + SC \middle| RB + SD}{UA + VC \middle| UB + VD}\right] \tag{28}$$

relies on *divide-and-conquer* (27):

$$\begin{array}{ll} & \left[\left[\frac{R}{U}\right] \middle\| \left[\frac{S}{V}\right]\right] \cdot \left[\left[\frac{A}{C}\right] \middle\| \left[\frac{B}{D}\right]\right] \\ = & \quad \{\ \textit{junc}\text{-fusion (20)}\ \} \\ & \left[\left[\left[\frac{R}{U}\right] \middle\| \left[\frac{S}{V}\right]\right] \cdot \left[\frac{A}{C}\right] \middle\| \left[\left[\frac{R}{U}\right] \middle\| \left[\frac{S}{V}\right]\right] \cdot \left[\frac{B}{D}\right]\right] \\ = & \quad \{\ \text{divide and conquer (27) twice}\ \} \\ & \left[\left[\frac{R}{U}\right] \cdot A + \left[\frac{S}{V}\right] \cdot C \;\middle|\; \left[\frac{R}{U}\right] \cdot B + \left[\frac{S}{V}\right] \cdot D\right] \end{array}$$

$$
\begin{array}{ll}
= & \{\ \text{split-fusion (20) four times}\ \} \\
& \left[\left[\frac{R\cdot A}{U\cdot A}\right]+\left[\frac{S\cdot C}{V\cdot C}\right]\ \middle|\ \left[\frac{R\cdot B}{U\cdot B}\right]+\left[\frac{S\cdot D}{V\cdot D}\right]\right] \\
= & \{\ \text{blocked addition (26) twice}\ \} \\
& \left[\left[\frac{R\cdot A+S\cdot C}{U\cdot A+V\cdot C}\right]\ \middle|\ \left[\frac{R\cdot B+S\cdot D}{U\cdot B+V\cdot D}\right]\right] \\
= & \{\ \text{the same in block notation (24)}\ \} \\
& \left[\frac{RA+SC\,|\,RB+SD}{UA+VC\,|\,UB+VD}\right]
\end{array}
$$

5 Calculating Triple Nested Loops

By putting together the universal factorization of matrices in terms of the *junc* and *split* combinators, one easily infers yet another such property handling four blocks at a time:

$$X=\left[\frac{A_{11}\,|\,A_{12}}{A_{21}\,|\,A_{22}}\right]\Leftrightarrow\begin{cases}\pi_1\cdot X\cdot i_1=A_{11}\\ \pi_1\cdot X\cdot i_2=A_{12}\\ \pi_2\cdot X\cdot i_1=A_{21}\\ \pi_2\cdot X\cdot i_2=A_{22}\end{cases}$$

Alternatively, one may generalize (13,14) to blocked notation

$$\left[\frac{A_{11}\,|\,A_{12}}{A_{21}\,|\,A_{22}}\right]=i_1\cdot A_{11}\cdot\pi_1+i_1\cdot A_{12}\cdot\pi_1+i_1\cdot A_{21}\cdot\pi_1+i_2\cdot A_{22}\cdot\pi_2$$

which rewrites to

$$\left[\frac{A_{11}\,|\,A_{12}}{A_{21}\,|\,A_{22}}\right]=\begin{bmatrix}A_{11}&0\\0&0\end{bmatrix}+\begin{bmatrix}0&A_{12}\\0&0\end{bmatrix}+\begin{bmatrix}0&0\\A_{21}&0\end{bmatrix}+\begin{bmatrix}0&0\\0&A_{22}\end{bmatrix}$$

once injections and projections are replaced by the biproduct solution found in Section 3.

Iterated Biproducts. It should be noted that biproducts generalize to finitely many arguments, leading to an n-ary generalization of the (binary) *junc* / *split* combinators. The following notation is adopted in generalizing (13,14):

$$A=\left[\,A_1\,|\ldots|\,A_p\,\right]=\bigoplus_{1\le j\le p}A\cdot i_j=\sum_{j=1}^{p}A\cdot i_j\cdot\pi_j \tag{29}$$

$$A=\left[\frac{A_1}{\vdots}{\atop A_m}\right]=\bigominus_{1\le j\le m}\pi_j\cdot A=\sum_{j=1}^{m}i_j\cdot\pi_j\cdot A \tag{30}$$

Note that all laws given so far generalize accordingly to n-ary *splits* and *juncs*. In particular, we have the following universal properties:

$$X = \bigcirc_{1\leq j\leq p} A_j \Leftrightarrow \bigwedge_{1\leq j\leq p} X \cdot i_j = A_j \tag{31}$$

$$X = \ominus_{1\leq j\leq m} A_j \Leftrightarrow \bigwedge_{1\leq j\leq m} \pi_j \cdot X = A_j \tag{32}$$

Further note that m, p can be chosen as large as possible, the limit taking place when blocks A_i become atomic. In this limit situation, a given matrix $m \xleftarrow{A} n$ is defined in terms of its elements A_{jk} as:

$$A = \left[\begin{array}{c|c|c} a_{11} & \cdots & a_{1n} \\ \hline \vdots & \ddots & \vdots \\ \hline a_{m1} & \cdots & a_{mn} \end{array}\right] = \sum_{\substack{1\leq j\leq m \\ 1\leq k\leq n}} i_j \cdot \pi_j \cdot A \cdot i_k \cdot \pi_k = \bigoplus_{\substack{1\leq j\leq m \\ 1\leq k\leq n}} \pi_j \cdot A \cdot i_k \tag{33}$$

where $\bigoplus_{\substack{1\leq j\leq m \\ 1\leq k\leq n}}$ abbreviates $\ominus_{1\leq j\leq m} \bigcirc_{1\leq k\leq n}$ — equivalent to $\bigcirc_{1\leq k\leq n} \ominus_{1\leq j\leq m}$ by the generalized exchange law (24).

Our final calculation shows how iterated biproducts "explain" the traditional for-loop implementation of MMM. Interestingly enough, such iterative implementation is shown to stem from generalized divide-and-conquer (27):

$$\begin{array}{ll} C = A \cdot B & \\ = & \{\ (33), (29) \text{ and } (30)\ \} \\ & (\ominus_{1\leq j\leq m} \pi_j \cdot A) \cdot (\bigcirc_{1\leq k\leq n} B \cdot i_k) \\ = & \{\ \text{generalized split-fusion } (21)\ \} \\ & \ominus_{1\leq j\leq m} (\pi_j \cdot A \cdot (\bigcirc_{1\leq k\leq n} B \cdot i_k)) \\ = & \{\ \text{generalized either-fusion } (20)\ \} \\ & \ominus_{1\leq j\leq m} (\bigcirc_{1\leq k\leq n} \pi_j \cdot A \cdot B \cdot i_k) \\ = & \{\ (29), (30) \text{ and generalized } (21) \text{ and } (20)\ \} \\ & \ominus_{1\leq j\leq m} (\bigcirc_{1\leq k\leq n} ((\bigcirc_{1\leq l\leq p} \pi_j \cdot A \cdot i_l) \cdot (\ominus_{1\leq l\leq p} \pi_l \cdot B \cdot i_k))) \\ = & \{\ \text{generalized divide-and-conquer } (27)\ \} \\ & \ominus_{1\leq j\leq m} (\bigcirc_{1\leq k\leq n} (\sum_{1\leq l\leq p} \pi_j \cdot A \cdot i_l \times \pi_l \cdot B \cdot i_k)) \end{array}$$

As we can see in the derivation path, the choices for the representation of A and B impact on the derivation of the intended algorithm. Different choices will alter the order of the triple loop obtained. Proceeding to the loop inference will

involve the expansion of C and the normalization of the formula into sum-wise notation:

$$\bigoplus_{\substack{1\le k\le m\\ 1\le j\le n}} \pi_j \cdot C \cdot i_k = \bigominus_{1\le j\le m} \left(\bigoplus_{1\le k\le n} \left(\sum_{1\le l\le p} \pi_j \cdot A \cdot i_l \times \pi_l \cdot B \cdot i_k \right) \right)$$

$$\Leftrightarrow \qquad \{ \ (33), (29) \text{ and } (30) \ \}$$

$$\bigominus_{1\le j\le m} \left(\bigoplus_{1\le k\le n} \pi_j \cdot C \cdot i_k \right) = \bigominus_{1\le j\le m} \left(\bigoplus_{1\le k\le n} \left(\sum_{1\le l\le p} \pi_j \cdot A \cdot i_l \times \pi_l \cdot B \cdot i_k \right) \right)$$

At this point we rely on the universality of the *junc* and *split* constructs (31,32) to obtain from above the post-condition of the algorithm:

$$\bigwedge_{1\le j\le m} \left(\bigwedge_{1\le k\le n} (\pi_j \cdot C \cdot i_k = \sum_{1\le l\le p} \pi_j \cdot A \cdot i_l \times \pi_l \cdot B \cdot i_k) \right)$$

This predicate expresses an outer traversal indexed by j, an inner traversal indexed by k and what the expected result in each element of output matrix C is. Thus we reach three nested for-loops of two different kinds: the two outer-loops (corresponding to indices j, k) provide for *navigation*, while the inner loop performs an *accumulation* (thus the need for initialization):

```
for j = 1 to m do
  for k = 1 to n do
    C[j][k] ← 0
    for l = 1 to p do
      C[j][k] ← C[j][k] + A[j][l] * B[l][k]
    end for
  end for
end for
```

Different matrix memory mapping schemes give rise to the interchange of the j, k and l in the loop above [14]. This is due to corresponding choices in the derivation granted by the generalized exchange law (24), among others.

Other variants of blocked MMM (28) such as e.g. Strassen's or Winograd's [15] rely mainly on the additive structure of Mat_K and thus don't pose new challenges. However, whether such algorithms can be better explained in more structured, biproduct-based derivations is a matter of future research.

6 Related Work

The formulation of categories of matrices can be traced back to [6] and [16], where the focus is either on exemplifying additive categories and on the relationship between linear transformations and matrices. No effort on exploiting biproducts calculationally is present, let alone algorithm derivation.

Bloom et al [8] make use of what we have identified as the standard biproduct (enabling blocked matrix algebra) to formalize column and row-wise matrix join

and fusion. Instead of a calculational approach to linear algebra algorithmics, the emphasis is on iteration theories which matricial theories are a particular case of. Furthermore this work makes use of other algebraic properties of $\mathbf{Mat}_K$ which we aim to encompass later.

Other categorial approaches to linear algebra include relative monads [17], whereby the category of finite-dimensional vector spaces arises as a kind of Kleisli category. Efforts by the mathematics of program construction community in the derivation of matrix algorithms include the study of two-dimensional pattern matching [18]. An account of work on calculational, index-free reasoning about regular and Kleene algebras of matrices can be found in [19].

7 Conclusions and Current Work

A comprehensive calculational approach to linear algebra algorithm specification, transformation and generation is still missing. However, the successes reported by the engineering field in the automatic library generation are a good cue to the feasibility of such an approach.

In this paper we have presented a formalization of matrices as categorial morphisms (arrows) in a way which relates categories of matrices to relation algebra and program calculation. Our case study — matrix multiplication — is dealt with in an elegant, calculational style whereby its divide-and-conquer and triple nested loop algorithmic implementations were derived.

The notion of a categorial biproduct is at the heart of the whole approach. Using the category of matrices and its biproducts the conversion from the declarative definition of a matrix to its indexed version is made possible thanks to the properties of projections and injections, as shown in the derivation of the triple for-loop.

We plan to carry on this work in several directions. The background of our project is the formalization of the SPL language [2] and, in this respect, work has only started. However in its beginning, our biproduct-centered approach is already telling us what to do next, as happens for instance with the biproduct nature of the *gather/scatter* matrices of the SPIRAL system [20].

Next in the plan we want to exploit other solutions to the biproduct equations, while checking which "chapters" of linear algebra [16] they are able to constructively explain. Think of Gaussian elimination, for instance, whose main steps involve row-switching, row-multiplication and row-addition, and suppose one defines the following transformation t catering for the last two, for a given α:

$$t : (\ n \xleftarrow{\quad} n\ ) \times (\ n+n \xleftarrow{\quad} m\ ) \to (\ n+n \xleftarrow{\quad} m\ )$$
$$t(\alpha, \left[\frac{A}{B}\right]) \ = \ \left[\frac{A}{\alpha A + B}\right]$$

(Thinking in terms of blocks A and B rather than rows is more general; in this setting, arrow $n \xleftarrow{\alpha} n$ means $n \xleftarrow{id} n$ with all 1s replaced by αs.) Let us

analyze the essence of t by using the blocked-matrix calculus in reverse order :

$$\begin{aligned} t(\alpha, \left[\frac{A}{B}\right]) &= \left[\frac{A}{\alpha A + B}\right] \\ &= \quad \{ \text{ (28) in reverse order } \} \\ &\quad \left[\frac{1|0}{\alpha|1}\right] \cdot \left[\frac{A}{B}\right] \\ &= \quad \{ \text{ divide-and-conquer (27) } \} \\ &\quad \left[\frac{1}{\alpha}\right] \cdot A + \left[\frac{0}{1}\right] \cdot B \end{aligned}$$

It can be shown that the last expression, which has the same shape as (14), is in fact the *split* combinator generated by another biproduct, parametric on α:

$$\begin{aligned} \pi'_1 &= \left[\, 1\ 0 \,\right], \pi'_2 = \left[\, -\alpha\ 1 \,\right] \\ i'_1 &= \begin{bmatrix} 1 \\ \alpha \end{bmatrix}, i'_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \end{aligned}$$

In summary, this biproduct, which extends the one studied earlier on (they coincide for $\alpha := 0$) provides a categorial interpretation of one of the steps of Gaussian elimination. We are currently investigating its role in a constructive proof of the corresponding, well-known algorithm, which we lay down recursively as follows, using block-notation (24):

$$\begin{aligned} &ge : (\ 1 + n \xleftarrow{\quad} 1 + m\ ) \to (\ 1 + n \xleftarrow{\quad} 1 + m\ ) \\ &ge \left[\frac{x\,|M}{N|Q}\right] = \left[\frac{x|\qquad M}{0|ge(Q - \frac{N}{x} \cdot M)}\right] \\ &ge\ x = x \end{aligned}$$

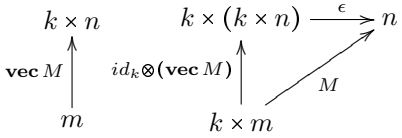
In particular, we want to provide a calculational alternative to the FLAME-styled derivation of the algorithm given in e.g. [3].

Last but not least, we want to address *vectorization* calculationally. The linearization of an arbitrary matrix into a vector is a data refinement step. This means finding suitable abstraction/representation relations [21] between the two formats and reasoning about them, including the refinement of all matrix operations into vector form.

The first part of the exercise proves easier than first expected: vectorization is akin to exponentiation, that is, *currying* [4] in functional languages. While currying "thins" the input of a given binary function by converting it into its unary (higher-order) counterpart, so does vectorization by thinning a given matrix $n \xleftarrow{M} km$ into $kn \xleftarrow{\mathbf{vec}\, M} m$, where k is the "thinning factor" [7]. (For $m = 1$, $\mathbf{vec}\, M$ is a column vector — the standard situation [22].) Once again,

our approach relies on capturing such a relationship by a universal property

$$X = \mathbf{vec}\, M \;\Leftrightarrow\; M = \epsilon \cdot (id \otimes X)$$



where $\otimes$ denotes Kronecker product [4], granting **vec** as a bijective transformation. So its converse **unvec** is also a bijection, whereby $\epsilon = \mathbf{unvec}\, id$. Put in other words, we are in presence of an adjunction between functor $FX = id_k \otimes X$ and itself. Taking advantage of this mathematical framework [23] in calculating the whole algebra of vectorization will keep the authors busy for a while [24].

Broadening scope, an aspect that needs investigation is how this "non-standard" treatment of matrices (data structures represented as arrows, as opposed to datatypes as objects) combines with theories of the rest of programming. For instance, its application to the emerging field of *linear algebra of programming* [25] and its combination with the monadic framework of [17] are topics for future research.

Acknowledgements. The authors would like to thank Markus Püschel (CMU) for driving their attention to the relationship between linear algebra and program transformation. Hugo Macedo further thanks the SPIRAL group for granting him an internship at CMU.

Thanks are also due to Michael Johnson and Robert Rosebrugh (Macquarie Univ.) for pointing the authors to the categories of matrices approach. Yoshiki Kinoshita (AIST, Japan) and Manuela Sobral (Coimbra Univ.) helped with further indications in the field.

This research was carried out in the context of the MONDRIAN Project funded by FCT contract PTDC/EIA-CCO/108302/2008. Hugo Macedo was partially supported by the *Fundação para a Ciência e a Tecnologia*, Portugal, under grant number SFRH/BD/33235/2007.

References

1. Franchetti, F., de Mesmay, F., McFarlin, D., Püschel, M.: Operator language: A program generation framework for fast kernels. In: IFIP Working Conference on Domain Specific Languages (DSL WC). (2009)
2. Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B.W., Xiong, J., Franchetti, F., Gačić, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code generation for DSP transforms. Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" **93**(2) (2005) 232–275

[4] Given $p \xleftarrow{A} m$ and $q \xleftarrow{B} n$, the Kronecker product $pq \xleftarrow{A \otimes B} mn$ is the matrix $A \otimes B = (a_{ij}B)$ [22].

3. de Geijn, R.A.V., Quintana-Ortí, E.S.: The Science of Programming Matrix Computations. `www.lulu.com` (2008)
4. Jones, S.P., Hughes, J., Augustsson, L., Barton, D., Boutel, B., Burton, W., Fasel, J., Hammond, K., Hinze, R., Hudak, P., Johnsson, T., Jones, M., Launchbury, J., Meijer, E., Peterson, J., Reid, A., Runciman, C., Wadler, P.: Report on the programming language Haskell 98 — a non-strict, purely functional language. Technical report (February 1999)
5. Bird, R., de Moor, O.: Algebra of Programming. Series in Computer Science. Prentice-Hall International (1997) C.A.R. Hoare, series editor.
6. MacLane, S.: Categories for the Working Mathematician (Graduate Texts in Mathematics). Springer (September 1998)
7. Macedo, H., Oliveira, J.: Matrices as arrows: a typed approach to linear algebra (2009) Extended abstract, CALCO-JNR Workshop, Sep. 6-10, 2009, Udine, Italy.
8. Bloom, S., Sabadini, N., Walters, R.: Matrices, machines and behaviors. Applied Categorical Structures **4**(4) (1996) 343–360
9. Allenby, R.B.J.T.: Linear Algebra. Elsevier (1995)
10. Tarski, A., Givant, S.: A Formalization of Set Theory without Variables. AMS (1987) AMS Col. Pub., volume 41, Providence, Rhode Island.
11. Frias, M.: Fork algebras in algebra, logic and computer science (2002) Logic and Computer Science. World Scientific Publishing Co.
12. Wolfram, S., et al.: Mathematica: a system for doing mathematics by computer. Addison-Wesley New York (1988)
13. Bird, R.: Lecture notes on constructive functional programming (1989) In M. Broy, editor, CMCS Int. Summer School directed by F.L. Bauer [et al.], Springer, 1989. NATO Adv. Science Institute (Series F: Comp. and System Sciences Vol. 55.
14. Goto, K., Geijn, R.A.v.d.: Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw. **34**(3) (2008) 1–25
15. D'Alberto, P., Nicolau, A.: Adaptive Strassen's matrix multiplication. In: ICS '07: Proc. of the 21st annual int. conf. on Supercomputing, NY, USA, ACM (2007)
16. MacLane, S., Birkhoff, G.: Algebra. AMS Chelsea (1999)
17. Altenkirch, T., Chapman, J., Uustalu, T.: Monads need not be endofunctors. In: Foundations of Software Science and Computational Structures. (2010) 297–311
18. Jeuring, J.: The derivation of hierarchies of algorithms on matrices. In Moller, B., ed.: Constructing Programs from Specifications, North-Holland (1991) 9–32
19. Backhouse, R.: Mathematics of Program Construction. Univ. of Nottingham (2004) Draft of book in preparation. 608 pages.
20. Voronenko, Y.: Library Generation for Linear Transforms. PhD thesis, Electrical and Computer Engineering, Carnegie Mellon University (2008)
21. Oliveira, J.N.: Transforming data by calculation. In: GTTSE'07. Volume 5235 of LNCS., Springer (2008) 134–195
22. Magnus, J., Neudecker, H.: The commutation matrix: Some properties and applications. The Annals of Statistics **7**(2) (1979) 381–394
23. Došen, K., Petrić, Z.: Self-adjunctions and matrices. Journal of Pure and Applied Algebra **184** (2003) 7–39
24. Macedo, H.D., Oliveira, J.N.: Exploring self-adjunctions in vectorization (2010) — in preparation.
25. Sernadas, A., Ramos, J., Mateus, P.: Linear algebra techniques for deciding the correctness of probabilistic programs with bounded resources. Technical report, SQIG-IT and TU Lisbon, 1049-001 Lisboa, Portugal (2008) — Short paper presented at LPAR 2008, Doha, Qatar. November 22-27.