# Type Your Matrices for Great Good (Functional Pearl)

A Haskell library of typed matrices and applications

Armando Santos
Department of Informatics
University of Minho
Braga, Portugal, Portugal
armando.j.santos@inesctec.pt

José N. Oliveira
Department of Informatics
University of Minho
Braga, Portugal, Portugal
jno@di.uminho.pt

## Abstract

We study a simple inductive data type for representing correct-by-construction matrices. Despite its simplicity, it can be used to implement matrix-manipulation algorithms efficiently and safely, performing in some cases faster than existing alternatives even though the algorithms are written in a direct and purely functional style. A rich collection of laws makes it possible to derive and optimise these algorithms using equational reasoning, avoiding the notorious off-by-one indexing errors when fiddling with matrix dimensions. We demonstrate the usefulness of the data type on several examples, and highlight connections to related topics in category theory.

***CCS Concepts*** • **Mathematics of computing**;

***Keywords*** Haskell, Linear Algebra of Programming, Matrices, Probabilistic Programming, Quantum Programming, Data Analysis

**ACM Reference Format:**
Armando Santos and José N. Oliveira. 2020. Type Your Matrices for Great Good (Functional Pearl): A Haskell library of typed matrices and applications. In *Proceedings of 13th ACM SIGPLAN International Haskell Symposium (Haskell'20).* ACM, New York, NY, USA, Article 1, 13 pages.

## 1 Introduction

Matrices are central to mathematics and computer science, from linear algebra and probability theory to machine learning and computer graphics. Matrices play an important role in problem specification as well as in finding efficient solutions, for which exists specialised hardware processing units [Sato et al. 2017; Volkov and Demmel 2008].

But what is a matrix really? A matrix is commonly viewed as an array of elements arranged in rows and columns. In textbooks, a matrix $A$ with $c$ columns and $r$ rows is typically pictured as a rectangular area whose elements $a_{ij}$ are numbers (or expressions denoting them):

$$A = \begin{bmatrix} a_{11} & \dots & a_{1c} \\ \vdots & \ddots & \vdots \\ a_{r1} & \dots & a_{rc} \end{bmatrix}$$

It is not surprising that software developers often translate this array-based matrix representation directly into code when implementing matrix algorithms. For dense matrices, this can give excellent results in terms of performance, but programming against such a low-level representation is challenging and ill-suited for purely functional programming languages.

On the other hand, matrix manipulation very often operates over *blocks* rather than over individual cell values, for instance over the three blocks of matrix $A = \left[\begin{array}{c|c} B & \frac{C}{D} \end{array}\right]$. In such cases, awkward, error-prone index calculations could remain implicit if smart matrix-block combinators were used.

This paper views matrices as inductive structures that can be constructed from simple primitives, as captured by the following data type[1]:

```
data Matrix e c r where
  One   :: e -> Matrix e () ()
  Join  :: Matrix e a r -> Matrix e b r
        -> Matrix e (Either a b) r
  Fork  :: Matrix e c a -> Matrix e c b
        -> Matrix e c (Either a b)
```

Here the type variable `e` stands for the type of the matrix *elements*, while `c` (*columns*) and `r` (*rows*) specify the dimensions. This data type will be discussed in more detail in §3; for now, we would like to emphasise that matrix dimensions appear only at the type level, i.e. it is impossible to make any dimension or indexing errors while constructing a value of type `Matrix e c r`.

This data type is matrix-block-oriented and particularly suitable to express block operations which, as will be seen soon, are very common and rely on a sound mathematical basis.

*Haskell'20, August 27-28, 2020, New Jersey, United States*
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00

[1]This paper uses Haskell but the presented ideas can be adapted to other functional programming languages.

### 1.1 Contributions

This paper presents a *type safe inductive matrix data structure* definition, exposing its biproduct architecture and equipped with a matrix programming library in Haskell, that stands out from the rest for having an inductive definition that enables writing statically typed matrix manipulation functions in a more elegant, calculational and efficient way.

The main goal of this paper is to demonstrate that by taking advantage of a strongly typed functional programming language, one can write and reason about elegant and composable linear algebra programs. Our specific contributions are:

- We develop a library for transforming and manipulating matrices and demonstrate its composability and flexibility.
- Compared to current libraries, ours is more compositional and polymorphic and does not have partial matrix manipulation functions (hence less chances for usage errors).
- Our implementation of matrices enables simple manipulation of submatrices, making it particularly suitable for formal verification and equation reasoning, using the mathematical framework defined by the linear algebra of programming [Oliveira 2012]. Furthermore, the data type constructors ensure that the matrices of this kind are sound, i.e. malformed matrices with incorrect dimensions of the sort, can not be constructed.
- Some practical examples that use the proposed matrix library in several application domains (e.g. probabilistic programming, quantum programming) are given and show how functional programming blends naturally with linear algebra.

### 1.2 Layout of the paper

The rest of the paper is structured as follows. A brief overview of the **Mat** category is given in §2.1 showing the rich algebra of matrices that emerges from the categorical notion of *biproduct*. The core representation of the proposed inductive matrix type unfolds in §3, followed by a concise explanation of how to write matrix manipulation functions around it (§4 to §7). Finally, some practical examples in the areas of probabilistic programming, quantum programming are given in §9, including evaluation. Related work, analysis of the proposed approach and directions for future research are discussed in sections §10 and §11, respectively.

## 2 Background

Category theory [Awodey 2010; MacLane 1971] is often referred to as a "theory of everything" because it is a framework where a lot of mathematical structures fit in. It has, in particular, a strong presence in functional programming [Hinze 2013; Milewski 2018]. Such an abstract approach is relevant because abstraction plays a major role in computing [Kramer 2007]. Category theory uses arrows as a generic notation able to cope with very distinct problem domains. These arrows, also called *morphisms*, are typed with source and target objects and need to satisfy certain properties in order to form a category.

A standard way to achieve typed functional programming is to use the category **Set** of sets to type functions. For instance, in addition to the function definition:

$$f\,x = x + 1$$

an arrow between sets is added in order to constraint the scope of the function application:

$$\begin{aligned} f &:: \mathbb{N} \longrightarrow \mathbb{N} \\ f\,x &= x + 1 \end{aligned}$$

This makes sure that $f$ can only be applied to arguments that are natural numbers. If by any chance this function is applied to a real number instead, this will be regarded as a wrong sentence and discarded by type checking.

The aphorism *functions are special cases of relations* means that a function $f :: A \longrightarrow B$ can also be typed in the wider category of binary relations, the **Rel** category. In general, a morphism $R :: A \longrightarrow B$ in **Rel** is a relation $R \subseteq A \times B$, for instance:

$$\begin{aligned} &R :: Object \longrightarrow Colour \\ &o\;R\;c = c \text{ is a colour of } o \end{aligned}$$

A function $f :: A \longrightarrow B$ is viewed as a relation wherever one writes the input/output relationship $b = f(a)$.

Both functions and relations have advanced type systems supported by the underlying categories. What about matrices? Matrices generalise relations into quantified ones, also termed *labelled* or *weighted* relations, leading into the category **Mat** of typed matrices.[2] A matrix $M$ with $c$ columns and $r$ rows can be regarded as a function $M(n, m)$ that tells the quantity occupying each cell $(n, m)$, for $1 \leq n \leq r, 1 \leq m \leq c$. From a categorical perspective, $M$ can be regarded as a morphism $c \xrightarrow{M} r$ indicating that matrix $M$ is of type $c \longrightarrow r$. In this setting, matrix-matrix multiplication can be expressed by arrow composition:

$$\begin{tikzcd} r & c \arrow[l, "M"'] & p \arrow[l, "N"'] \arrow[ll, bend left, "M\cdot N"] \end{tikzcd} \qquad (1)$$

It has been shown that other interesting combinators arise from so-called *biproducts* in the **Mat** category, which capture block-matrix operations in a natural way [Macedo and Oliveira 2013].

[2]See e.g. [Macedo and Oliveira 2013]. Strictly speaking, there is one such category $\mathbf{Mat}_k$ per cell-type $k$, but ignoring this detail will not harm this summary of the overall theory.

### 2.1 Structure of Mat

The structure of the **Mat** category is described below. Figure 1 provides a reference guide containing the main algebraic laws.

#### 2.1.1 Basic structure

For every dimension $d$ there is a matrix $d \longrightarrow d$ which is the unit of composition, i.e. the (square) *identity* matrix of size $d$:

$$\begin{tikzcd} m \arrow[d, "M"'] \arrow[dr, phantom] & m \arrow[l, "id_m"'] \arrow[d, "M"] \arrow[dl, "M"'] \\ n & n \arrow[l, "id_n"] \end{tikzcd} \qquad id_n \cdot M = M = M \cdot id_m \quad (2)$$

Under composition (1) matrices form a category whose objects are matrix dimensions and whose morphism $n \xrightarrow{M} m$, $k \xrightarrow{N} n$, etc are the matrices themselves [Macedo and Oliveira 2013; MacLane 1971]. *Vectors* are special cases of matrices in which one of the dimensions is 1, for instance:

$$v = \begin{bmatrix} v_1 \\ \vdots \\ v_r \end{bmatrix} \text{ and } w = \begin{bmatrix} w_1 & \dots & w_c \end{bmatrix}$$

Column vector $v$ is of type $1 \longrightarrow r$ (1 column and $r$ rows) and row vector $w$ is of type $c \longrightarrow 1$ (1 row and $c$ columns). The convention is that lowercase letters denote vectors and uppercase letters denote matrices.

**Mat** is a "dagger category" in the sense that every matrix $M$ can be transposed by swapping rows with columns. We denote the transposition (converse) of matrix $c \xrightarrow{M} r$ by $r \xrightarrow{M^\circ} c$. As expected, the idempotence law (5) and the contravariance law (6) hold.

#### 2.1.2 Bilinearity

Given two matrices $c \xrightarrow{M,N} r$ of the same type, it makes sense to add them entry-wise to obtain the matrix $M + N$, where the symbol $+$ promotes the underlying cell-level additive operator to the matrix-level. Likewise, the additive unit cell value 0 is lifted to the matrix 0 fully filled with 0s.

Matrix 0 is the *unit* element of matrix addition (8) and the *zero* (absorbent) element of matrix composition (9). The fact that composition is *bilinear* relative to $+$, as given by laws (10,11), is central to linear algebra as a whole.

In the same way that $M{+}N$ promotes the addition of matrix cells to the addition of the matrices themselves, the same promotion may take place with respect to the whole cell-level algebra. For example, cell value multiplication results in a matrix multiplication, denoted by $M{\times}N$ (for $M$ and $N$ of the same type), also known as the Hadamard product, which is commutative, associative and distributive over addition (i.e. bilinear).

#### 2.1.3 Products and co-products

From law (12) on-wards, Fig. 1 presents the basic algebra of matrix-block combinators that will be used throughout the rest of the paper. The concept of a *biproduct* is central to their characterisation, combining categorical products and co-products into a single construction.

The diagram below shows how biproducts relate to products and co-products in the **Mat** category, compare with (16) and (17) in Figure 1.

$$\begin{tikzcd} & d & \\ a \arrow[ur, bend left, "A"] \arrow[r, bend left, "i_1"] & a+b \arrow[u, "{[A|B]}"'] \arrow[l, bend left, "\pi_1"] \arrow[r, bend right, "\pi_2"'] & b \arrow[ul, bend right, "B"'] \arrow[l, bend right, "i_2"'] \\ & c \arrow[u, "{\left[\frac{C}{D}\right]}"'] \arrow[ul, bend left, "C"] \arrow[ur, bend right, "D"'] & \end{tikzcd}$$

Expressions $[A \mid B]$ and $\left[\frac{C}{D}\right]$ will be read "A join B" and "C fork D", respectively. These operators purport the effect of putting matrices side by side and on top of one another, respectively. Using such operators, projections $\pi_1$, $\pi_2$ and injections $i_1$, $i_2$ "decompose" the identity matrix through the so-called *reflection* laws (14,15).

The well-formedness of matrix block construction is therefore ensured by static type checking. To ensure correct typing, typed linear algebra practitioners are encouraged to draw the type diagrams associated to the expressions and equations in mind.

The laws of Figure 1 enable one to calculate standard linear algebra rules and algorithms. See as example the calculation alongside of the *divide-and-conquer* law of matrix multiplication (24), also known as block-multiplication.

$$\begin{array}{ll} & [A|B] \cdot \left[\frac{C}{D}\right] \\ = & \quad \{ \text{ fork definition (17) } \} \\ & [A|B] \cdot (i_1 \cdot C + i_2 \cdot D) \\ = & \quad \{ \text{ bilinearity (10) } \} \\ & [A|B] \cdot i_1 \cdot C + [A|B] \cdot i_2 \cdot D \\ = & \quad \{ \text{ cancellation (22) } \} \\ & A \cdot C + B \cdot D \end{array}$$

Looking at matrices as lists of lists, or arrays, or graphs is a rather poor perspective on linear algebra. Somewhat more useful is to focus on how matrices are built or partitioned and on which formal rules are there for handling their internal structure, as seen above. Explicit, painful, low-level matrix manipulation is a source of dubious code, which is error-prone and difficult to analyse. Looking at it from a higher, abstract point of view that relies on a sound mathematical theory provides both simplicity and reliability benefits.

## 3 Matrix data type

Putting theory into practice, a safe and minimalist inductive matrix data type is encoded in Haskell. This data type takes into account certain conditions needed in order to accommodate the advantages that a good type system and the

**Composition**

$$M \cdot (N \cdot Q) = (M \cdot N) \cdot Q \tag{3}$$
$$M \cdot id = M = id \cdot M \tag{4}$$

**Converse (transposition)**

$$(M^\circ)^\circ = M \tag{5}$$
$$(M \cdot N)^\circ = N^\circ \cdot M^\circ \tag{6}$$

**Additivity**

$$M + (N + Q) = (M + N) + Q \tag{7}$$
$$M + 0 = M = 0 + M \tag{8}$$
$$M \cdot 0 = 0 = 0 \cdot M \tag{9}$$

**Bilinearity**

$$M \cdot (N + P) = M \cdot N + M \cdot P \tag{10}$$
$$(N + P) \cdot M = N \cdot M + P \cdot M \tag{11}$$

**Universal properties**

$$X = [\ A \mid B\ ] \iff \left\{ \begin{array}{l} X \cdot i_1 = A \\ X \cdot i_2 = B \end{array} \right. \tag{12}$$
$$X = \left[ \frac{C}{D} \right] \iff \left\{ \begin{array}{l} \pi_1 \cdot X = C \\ \pi_2 \cdot X = D \end{array} \right. \tag{13}$$

**Reflection**

$$[i_1 | i_2] = id \tag{14}$$
$$\left[ \frac{\pi_1}{\pi_2} \right] = id \tag{15}$$

**Join and Fork**

$$[A|B] = A \cdot \pi_1 + B \cdot \pi_2 \tag{16}$$
$$\left[ \frac{C}{D} \right] = i_1 \cdot C + i_2 \cdot D \tag{17}$$

**Fusion**

$$P \cdot [A|B] = [P \cdot A | P \cdot B] \tag{18}$$
$$\left[ \frac{A}{B} \right] \cdot P = \left[ \frac{A \cdot P}{B \cdot P} \right] \tag{19}$$

**Absorption**

$$[A \mid B] \cdot (C \oplus D) = [A \cdot C \mid B \cdot D] \tag{20}$$
$$\left[ \frac{A}{B} \right] \cdot (C \oplus D) = \left[ \frac{A \cdot C}{B \cdot D} \right] \tag{21}$$

**Cancellation**

$$[A \mid B] \cdot i_1 = A\,,\ \ [A \mid B] \cdot i_2 = B \tag{22}$$
$$\pi_1 \cdot \left[ \frac{A}{B} \right] = A\,,\ \ \pi_2 \cdot \left[ \frac{A}{B} \right] = B \tag{23}$$

**Divide and conquer**

$$[A \mid B] \cdot \left[ \frac{C}{D} \right] = A \cdot C + B \cdot D \tag{24}$$

**Converse duality**

$$[A \mid B]^\circ = \left[ \frac{A^\circ}{B^\circ} \right] \tag{25}$$

**Exchange** ("abide") law

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

**Blocked addition**

$$[A \mid B] + [C \mid D] = [A + C \mid B + D] \tag{27}$$
$$\left[ \frac{A}{B} \right] + \left[ \frac{C}{D} \right] = \left[ \frac{A + C}{B + D} \right] \tag{28}$$

**Structural equality**

$$[A \mid B] = [C \mid D] \iff A = C \wedge B = D \tag{29}$$
$$\left[ \frac{A}{B} \right] = \left[ \frac{C}{D} \right] \iff A = C \wedge B = D \tag{30}$$

**Figure 1.** Summary of the laws of block-linear-algebra enabled by biproducts. (Valid only if over the same biproduct.)

quantitative extension of the algebra of programming have to offer: be polymorphic, have statically typed dimensions, be type-inference friendly, be correct-by-construction and amenable to elegant and calculational manipulation.

As already mentioned, matrices are typed according to their dimensions and dealing with natural numbers at the type level is something that is possible in some of the most recent functional typed languages [Bove et al. 2009; Brady 2013], including Haskell. Thus, a first approach to designing the inductive type could be:

```haskell
data Matrix e (c :: Nat) (r :: Nat) where
  One   :: e -> Matrix e 1 1
  Join  :: Matrix e a r -> Matrix e b r
        -> Matrix e (a + b) r
  Fork  :: Matrix e c a -> Matrix e c b
        -> Matrix e c (a + b)
```

However, when dealing with type families that are responsible for calculating simple natural number arithmetics such as (+), the compiler has a very hard time inferring the correct types when writing algorithms via pattern-matching, for example, given that addition is not injective [Stolarek et al. 2015]. The approach followed in this paper is then based on the following GADT [Peyton Jones et al. 2006]:

```haskell
data Matrix e c r where
  One   :: e -> Matrix e () ()
  Join  :: Matrix e a r -> Matrix e b r
        -> Matrix e (Either a b) r
  Fork  :: Matrix e c a -> Matrix e c b
        -> Matrix e c (Either a b)
```

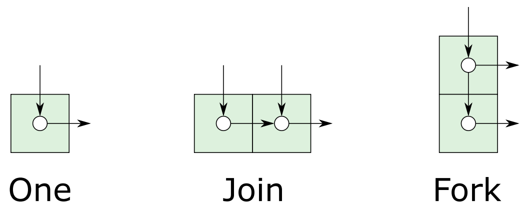


**Figure 2.** One-Join-Fork matrix constructors.

Here `One` is the inductive base case that construct the *one-element* matrix; `Join` and `Fork` are the two basic binary constructors available for building matrices out of other matrices as you can see in Fig. 2.

These simple four constructors provide a type safe inductive definition where it is not possible to construct a malformed matrix such as

```
Matrix { matrix = [[1,0],[0,1]], dimensions = (4,4) }
```

for example.

The trick is to define a recursive data type in which dimensions are typed by algebraic data types and use a GADT in order to control the output type dimensions and maintain the `Join` and `Fork` dimension invariants across the data structure.

## 3.1 Constructing matrices

This solution was built on the notion that algebraic data types are isomorphic to their cardinality, e.g. $|$`Void`$| \cong 0$, $|$`()`$| \cong 1$, $|$`Either a b`$| \cong |a| + |b|$ and so on. Furthermore, this GADT guarantees that a matrix will always have valid dimensions, i.e. is type-correct by construction. For instance, the $2 \times 2$ identity matrix $\left[\begin{array}{c|c} \frac{1}{0} & \frac{0}{1} \end{array}\right]$ can be expressed by:

```
iden2x2 :: Num e
        => Matrix e (Either () ()) (Either () ())
iden2x2 = Fork (Join (One 1) (One 0))
               (Join (One 0) (One 1))
```

For larger matrices, declaring type signatures by hand becomes troublesome. So, two type families were created to make it easier to work with matrix dimension definition:

```
type family Count (d :: Type) :: Nat where
  Count Void          = 0
  Count ()            = 1
  Count (Either a b) = (+) (Count a) (Count b)
  Count (a, b)        = (*) (Count a) (Count b)
  -- Generics
  -- (...)

type family FromNat (n :: Nat) :: Type where
  FromNat 0 = Void
  FromNat 1 = ()
  FromNat n = FromNat' (Mod n 2 == 0)
                       (FromNat (Div n 2))
```

```
type family FromNat' (b :: Bool) (m :: Type) where
  FromNat' 'True m  = Either m m
  FromNat' 'False m = Either () (Either m m)
```

The purpose of the `Count` type family will be used to compute the normalised dimension types, as will be seen in §7.1. If one wants to be able to have matrix typed by generic arbitrary data types (as we'll see in §9), one needs to add support for generic types. We leave the trivial implementation out of the paper for simplicity. The `FromNat` type family builds a balanced tree of `Either`s at the type-level, from a type-level natural, that is supposed to be the matrix dimension.

These type families take advantage of the algebraic data type cardinality isomorphism and provide a conversion mechanism *from* and *to* data types/type-level naturals. So the same $2 \times 2$ identity matrix can now be defined as:

```
iden2x2 :: Num e => Matrix e (FromNat 2) (FromNat 2)
iden2x2 = Fork (Join (One 1) (One 0))
               (Join (One 0) (One 1))
```

Notice that type families can be seen as type-level functions that are called at compile-time. So a type-level demanding program will take longer to compile, specially if it declares several large matrices. Thankfully, `FromNat` takes advantage of the possibility of reducing the number of computations needed to calculate the normalised dimension type that provides a significant speedup.

## 4 Matrix manipulation and transformation

The proposed inductive matrix data type enables the use of pattern matching and the laws of the linear algebra of programming (§2.1.3) to write total, efficient and statically typed manipulation and transformation functions. In view of this, matrix composition (*aka* matrix-matrix multiplication) can be defined elegantly and in a more calculational way, in contrast with the partial, (ugly) low-level nested for-loop implementation, which can be found in most imperative languages:

```
comp :: Num e => Matrix e cr rows
     -> Matrix e cols cr -> Matrix e cols rows
comp (One a) (One b)         = One (a * b)
comp (Join a b) (Fork c d)  = comp a c + comp b d
comp (Fork a b) c            =
    Fork (comp a c) (comp b c)
comp c (Join a b)            =
    Join (comp c a) (comp c b)
```

Like matrix multiplication, other common operations, such as matrix transposition, benefit from a block-oriented structure that leads to a simple and natural divide-and-conquer algorithmic solution. Performance wise, this means that without much effort we can obtain optimal cache-oblivious algorithms[3] [Frigo et al. 1999]. The basic philosophy in designing a cache-oblivious algorithm is to use a recursive approach that repeatedly divides the dataset until it eventually becomes cache resident and thus cache optimal, as we can see from the definition of the comp function above.

In the type-class mechanics of Haskell [Hall et al. 1996], matrix entry-wise addition, multiplication and subtraction fit nicely by defining a **Num** instance, as well as entry-wise matrix comparison and equality, by defining **Ord** and **Eq** instances, respectively. Sadly, there are no fusion laws that help defining an appropriate inductive definition of matrix equality when the two matrices have different valid permutations of **Join**s and **Fork**s, for example. This can still be done, however, by taking advantage of the exchange ("*abide*") law (26):

```
-- Abide Join-Fork
abideJF :: Matrix e cols rows -> Matrix e cols rows
abideJF (Join (Fork a c) (Fork b d)) =
    Fork
        (Join (abideJF a) (abideJF b))
        (Join (abideJF c) (abideJF d))
abideJF (One e)    = One e
abideJF (Join a b) = Join (abideJF a) (abideJF b)
abideJF (Fork a b) = Fork (abideJF a) (abideJF b)

instance Eq e => Eq (Matrix e cols rows) where
  (One a) == (One b)             = a == b
  (Join a b) == (Join c d)       = a == c && b == d
  (Fork a b) == (Fork c d)       = a == c && b == d
  x@(Fork _ _) == y@(Join _ _) = x == abideJF y
  x@(Join _ _) == y@(Fork _ _) = abideJF x == y
```

It is worth noting that one major downside of this encoding is that `Either (Either () ()) ()` is different from `Either () (Either () ())`, for example, even though both represent 3. This implies that two matrices with the same dimension can have different types, and thus, for example, they can not be tested for equality. One way to address this downside is shown in §7.1, by introducing a new-type matrix wrapper with generic arbitrary dimensions. Another approach is to manually write an identity matrix that exposes this isomorphism and use it to obtain a matrix of the same dimensions.

## 5 Matrix construction

It is not possible to construct typed matrices depending on the type of the dimensions desired, because GHC is not able to infer the correct GADT types. One way to achieve this is by using type-classes. Type-classes can be seen as functions from types to values and they are the solution for writing methods that allow building matrices from other representations (e.g. list of lists of elements) in the same inductive way, as is the case of class **FromLists**.

```
class FromLists e cols rows where
  fromLists :: [[e]] -> Matrix e cols rows
```

Note that fromLists can fail at run-time if the input list is not complacent with the desired matrix dimensions. It is possible to offer wrapper functions around it that guarantee it won't fail at compile time.

## 6 Category instance

In §1 we showed how matrices form a category where objects are dimensions and morphisms are the matrices themselves.

The proposed inductive structure arranges its type parameters in order to be able to provide a type-class instance for **Category**. It helps the end-user to make better use of code and reason about it, by using more readable notation and by allowing the compiler to infer which type-class instances to use. Throughout the rest of the paper we will use id and (.) interchangeably to mean either function or matrix composition/identity.

Given that the identity matrix needs certain type constrains on the dimension types, only a constrained version of the **Category** type-class can be offered.[4] In this context, the following instance is given:

```
class Category k where
    type Object k o :: Constraint
    type Object k o = ()
    id :: Object k a => k a a
    (.) :: k b c -> k a b -> k a c

instance Category (Matrix e) where
    type Object (Matrix e) a =
        (FromLists e a a, Countable a)
    id = iden
    (.) = comp
```

Note that `Countable = KnownNat (Count a)`.

## 7 End-user interface

As one can imagine, working with very large matrices at the type level can be an unpleasant experience. This section presents two new-type wrappers around the canonical data type that aim to improve end-user interfacing, also giving an overview of the available library API and how error messages look like when using the generalised dimensions wrapper.

### 7.1 Matrix new type wrappers

It is possible to abstract the use of the **FromNat** type family, obtaining a new-type matrix wrapper which dimensions are type level naturals (provided by the TypeLits library).

[3]A cache-oblivious algorithm is such that no variables that depend on hardware parameters, such as the size of the cache and the length of the cache-line, need to be tuned to achieve optimality.

[4]Note that this limitation also happens when trying to implement idiomatic instances of the **Functor** hierarchy in Haskell.

```
import qualified Matrix.Internal as I

newtype Matrix e (cols :: Nat) (rows :: Nat) =
    M (I.Matrix e
        (I.FromNat cols)
        (I.FromNat rows))
```

Thanks to the type family defined below, it is possible to attain a matrix typed by arbitrary generic data types. This will be the new type wrapper used throughout the rest of the paper.

```
type family Normalize (d :: Type) :: Type where
  Normalize (Either a b) = Either (Normalize a)
                                  (Normalize b)
  Normalize d            = FromNat (Count d)

newtype Matrix e (cols :: Type) (rows :: Type) =
    M (I.Matrix e
        (I.Normalize cols)
        (I.Normalize rows))
```

`Normalize` needs to preserve the `Either`-based structure to comply with the type signature of `Join` and `Fork`, balancing the tree in all other cases.

This new-type captures the matrix type generalisation proposed by Oliveira [2012]. In short, objects in categories of matrices can be generalised from numeric dimensions ($n, m \in \mathbb{N}_0$) to arbitrary denumerable types ($A, B$), taking disjoint union $A + B$ for $m + n$, Cartesian product $A \times B$ for $m \times n$, unit type () for number 1, etc.

## 7.2 Algebra of Programming API

The `Matrix` data type and the aforementioned new-type wrappers are a part of the LAoP programming library developed in Haskell.[5] This library provides an API for construction and manipulation of these types. The API offers the main combinators of the linear algebra of programming wherefrom other linear algebra operations can be derived.

Listing 1 presents the set of most important function signatures that are part of the generalised dimensions module API. As an example of using this API to write matrix operations by composing functions, the following function provides a `join`/`fork` version of matrix entry-wise addition, where (-|-) can be seen as the matrix direct sum operation:

```
addition :: (...) => Matrix e cols rows
         -> Matrix e cols rows -> Matrix e cols rows
addition a b =
    (join id id) . (a -|- b) . (fork id id)
```

This expresses the relationship between the underlying additive operator and direct sum. The correctness of `addition` is granted by the absorption law (20), among others:

```
      (join id id) . (a -|- b) . (fork id id)
=         { absorption law (20) ; identity (2) }
      (join a b) . (fork id id)
=         { divide-and-conquer - (24); identity (2) }
      (a + b)
```

Note that all the combinators on dimension types are polymorphic. This is a feature that does not exist or is rather fragile in other programming matrix libraries. However, the expected type constraints and type-level mechanisms (type-level natural and type-families) that make the type polymorphism work can make the type-signatures convoluted. For space economy, these are omitted from the above-mentioned API and throughout the paper. Nonetheless, this downside is minimised by providing a set of aliases of type in order to reduce the length and improve the readability and, in some cases, reasoning of the required restrictions. For illustrative purposes, we give the full type signature of the `addition` function presented above with and without the syntactic sugar:

```
addition ::
  (Num e, FromLists c c, FromLists m m,
   KnownNat (Count c),  KnownNat (Count m))
  => Matrix e m c -> Matrix e m c  -> Matrix e m c

addition ::
  (Num e, FL c c, FL m m, CountableDims c m)
  => Matrix e m c -> Matrix e m c  -> Matrix e m c
```

When doing type-level programming error messages easily become cumbersome and hard to read. By using typed matrices and a strongly typed language most dimension check errors can be caught at compile time. However, if the error messages are not clear, this fact does not impose any benefit. Gladly, error messages in our library do not suffer from the type-level machinery involved as we can see from the simple example, where we try to join two matrices with different row types:

```
-- x :: Matrix Float Bool Bool
-- y :: Matrix Float Bool Ordering
error:
- Couldn't match type 'Ordering' with 'Bool'
Expected type: Matrix Float Bool Bool
Actual type: Matrix Float Bool Ordering
- In the second argument of 'join', namely 'y'
In the expression: join x y
In an equation for 'it': it = join x y
```

The matrix programming library developed in the scope of this paper can be found in the Hackage repository along with its API documentation[6].

[5]LAoP stands for "Linear Algebra of Programming".

[6]https://hackage.haskell.org/package/laop

```
one           :: e -> Matrix e () () -- Unit constructor
join          :: Matrix e a r -> Matrix e b r -> Matrix e (Either a b) r -- Join constructor
fork          :: Matrix e c a -> Matrix e c b -> Matrix e c (Either a b) -- Fork constructor
matrixBuilder :: (...) => ((a, b) -> e) -> Matrix e a b                  -- Builder function
fromF         :: (...) => (a -> b) -> Matrix e a b                       -- Lifts functions
point         :: (...) => a -> Matrix e () a                             -- Point vector
comp          :: Num e => Matrix e cr r -> Matrix e c cr -> Matrix e c r -- Composition (MMM)
(.|)          :: Num e => e -> Matrix e c r -> Matrix e c r        -- Scalar multiplication
(./)          :: Fractional e => Matrix e c r -> e -> Matrix e c r -- Scalar division
iden          :: (...) => Matrix e c c            -- Identity matrix
tr            :: Matrix e c r -> Matrix e r c     -- Transposition
abideJF       :: Matrix e c r -> Matrix e c r     -- Join-Fork exchange
abideFJ       :: Matrix e c r -> Matrix e c r     -- Fork-Join exchange
p1            :: (...) => Matrix e (Either m n) m -- First projection
p2            :: (...) => Matrix e (Either m n) n -- Second projection
i1            :: (...) => Matrix e m (Either m n) -- First injection
i2            :: (...) => Matrix e n (Either m n) -- Second injection
fstM          :: (...) => Matrix e (m, k) m  -- Pairing first projection
sndM          :: (...) => Matrix e (m, k) k  -- Pairing second projection
kr            :: (...) => Matrix e c a -> Matrix e c b -> Matrix e c (a, b)
              -- Pairing (Khatri-Rao Product)
(-|-)         :: (...) => Matrix e n k -> Matrix e m j -> Matrix e (Either n m) (Either k j)
              -- Co-product bi-functor (Direct Sum)
(><)          :: (...)
              => Matrix e m p -> Matrix e n q -> Matrix e (m, n) (p, q)
              -- Product bi-functor (Kronecker)
```

**Listing 1.** LAoP API

## 8 Equational reasoning

This section shows how to use equational reasoning and the laws of the linear algebra of programming to prove properties of functions on matrices and/or to obtain more efficient programs.

A good example of this is the `select` operator inspired by the `Selective` interface. Selective Functors are a recent abstraction in functional programming. The argument in favour of these Selective Functors advocates them as solving the limitation of Applicatives and Monads in the context of static analysis, allowing over-approximation and under-approximation of effects in a circuit with conditional branches [Mokhov et al. 2019]. From a linear algebra perspective, this abstraction allows the study of conditional probability calculations when dealing with left stochastic matrices [Santos 2020].

From an abstract point-of-view, the diagram alongside corresponds to the `ArrowChoice` implementation of `select` where, in the case of stochastic matrices, `m` can be seen as a probability distribution that outputs either `a` or `b`, and `y` is only computed for values of type `a`, all others are skipped.

$$\begin{tikzcd}
 & b & \\
a \arrow[ur, bend left, "y"] \arrow[r, "i_1"] & a+b \arrow[u, "{[y \mid id]}"'] & b \arrow[l, "i_2"'] \arrow[ul, bend right, "id"'] \\
 & c \arrow[u, "m"'] &
\end{tikzcd}$$

This leads to a straightforward implementation of `select` in terms of matrices:

```
select :: (...) => Matrix e c (Either a b)
       -> Matrix e a b -> Matrix e c b
select m y = join y id . m
```

We know upfront from the definition that a (possibly) expensive computation is taking place where one of the matrices is the `id`entity. But, from the type of `m` we know that it can be `m = Fork x z` for some `x` and `z` (12) and the implementation can take advantage of this:

```
     join y id . m
=        { m = Fork x z }
     join y id . Fork x z
=        { divide-and-conquer (24) }
     y . x + id . z
=        { identity law (2) }
     y . x + z
```

Thus one gets

```
select (Fork x z) y = y . x + z
```

gaining in efficiency because `x` is necessarily smaller than the original `m`. Note that `x` and `z` above can be, on their own, joins. In this case, by the abide law (26) one gets `m = Join (Fork x c) (Fork z d)` which lets us pattern match one level deeper and, benefiting from the divide-and-conquer law, end up with:

```
    join y id . m
=       { m = Join (Fork x c) (Fork z d) }
    join y id . Join (Fork x c) (Fork z d)
=       { fusion (18) }
    Join (join y id . Fork x c) (join y id . Fork z d)
=       { divide-and-conquer (24) twice; identity (2) twice }
    Join (y . x + c) (y . z + d)
```

Putting everything together, one gets the following more efficient implementation:

```
select :: (...) => Matrix e c (Either a b)
       -> Matrix e a b -> Matrix e c b
select (Fork x z) y                  = y . x + z
select (Join (Fork x c) (Fork z d)) y =
    join (y . x + c) (y . z + d)
select m y                           =
    join y id . m
```

By exploring linear algebra properties, the compiler can actually be instructed to optimise the program by defining correct rewrite rules [Jones et al. 2001]. GHC's rewrite rules allow the programmer to directly communicate to the compiler ways of optimising a program that is not obvious to it, for instance:

```
{-# RULES
   "prod/cancel1" forall a b. p1 . (fork a b) = a;
   "co-p/cancel1" forall a b. (join a b) . i1 = a;
#-}
```

This tells the compiler to skip computations wherever it encounters an instance of the cancellation laws (22,23). Other useful laws such as converse duality (25) or fusion (18,19) can be used for the same purpose. In possession of such rules, the compiler becomes aware of possible non trivial optimisations and applies a certain degree of equational reasoning to produce more efficient code.

## 9 Applications and benchmarks

This section starts by giving two simple examples of application of the LAoP library.

### 9.1 Probabilistic programming

Probabilistic programming arises naturally from functional programming once we replace "sharp" functions by probabilistic ones, represented by stochastic matrices, also known as Markov chains [Oliveira 2012]. Let us show this taking an example from the Wikipedia [2020]. Suppose we define the following predicates modelling the behaviour of a sprinkler, where `S` (sprinkler on/off), `R` (raining or not) and `G` (grass wet or not) are Booleans:

```
sprinkler :: R -> S          grass :: (S, R) -> G
sprinkler r = not r          grass (s,r) = s || r
```

The second predicate tells that the grass will be wet if and only if either it is raining or the sprinkler is on. The first tells that the sprinkler is on *iff* it is not raining. Composing these two predicates we see that rain completely determines the state of the grass:

```
grass (sprinkler s, rain) = not rain || rain
∴ True
```

Looking at the diagram alongside, where (▿) can be seen as equal to (&&&)[7], we see that the system has two possible states in `(G, (S, R))` — either `(True, (True, False))` or `(True, (False, True))` — the grass being wet in both. So it will melt because of being wet all the time.

$$\begin{tikzcd}(G,(S,R)) \\ (S,R) \arrow[u, "grass^{\triangledown} id"'] \\ R \arrow[u, "sprinkler^{\triangledown} id"'] \\ () \arrow[u, "rain"'] \end{tikzcd}$$

Clearly, this deterministic interpretation of the diagram does not correspond to reality, but its stochastic interpretation will do. For this, we just need to regard the arrows as denoting stochastic matrices and not pure functions, for instance[8]

$$R \xrightarrow{sprinkler} S \;=\; \left[\begin{array}{c|c} 0.60 & 0.99 \\ \hline 0.40 & 0.01 \end{array}\right]$$

$$(S,R) \xrightarrow{grass} G \;=\; \begin{bmatrix} 1.00 & 0.20 & 0.10 & 0.01 \\ 0 & 0.80 & 0.90 & 0.99 \end{bmatrix}$$

This describes a probabilistic system *reactive* to the rain. Once its distribution becomes known, eg.

$$1 \xrightarrow{rain} R \;=\; \begin{bmatrix} 0.80 \\ 0.20 \end{bmatrix}$$

one immediately gets the distribution of the overall state, given by column vector

$$1 \xrightarrow{state} (G,(S,R)) \;=\; \begin{array}{lll|l} G & S & R & \\ \hline \text{dry} & \text{off} & no & 0.4800 \\ & & yes & 0.0396 \\ & \text{on} & no & 0.0320 \\ & & yes & 0.0000 \\ \hline \text{wet} & \text{off} & no & 0.0000 \\ & & yes & 0.1584 \\ & \text{on} & no & 0.2880 \\ & & yes & 0.0020 \\ \hline \end{array} \qquad (31)$$

which is calculated following the diagram. Consider the following matrices

```
rain :: Matrix Prob () R
sprinkler :: Matrix Prob R S
grass :: Matrix Prob (S, R) G
```

(where `type Prob = Double`) encoded in the LAoP library, where we also free the types involved from the strict Boolean

[7]From `Control.Arrow`, specialised to `(->)`

[8]For easy reference we follow the Wikipedia example closely.

model, already visible in (31).[9] The distribution of the overall state displayed above is given by the expression

```
state = compose grass sprinkler rain
```

where

```
compose :: (...)
        => Matrix e (c, d) b
        -> Matrix e d c
        -> Matrix e a d
        -> Matrix e a (b, (c, d))
compose g s r = tag g . tag s . r

tag :: (...) => Matrix e a b -> Matrix e a (b, a)
tag f = kr f id
```

Note the role of the *tag* operation, which for functions amounts to `tag f x = (f x, x)`, that is, the output of $f$ is paired with its input. Combinator `compose` iterates this operation across compositions so as to get an account of all inputs and outputs, as is usual in Bayesian networks.[10]

Let `wet :: Matrix Prob () G`, `dry :: Matrix Prob () G`, `no :: Matrix Prob () R` (and so on) be the *points* of the data types involved in the model. Also remember projections `fstM` and `sndM`, introduced in Listing 1. Evaluating the overall probability of the grass being wet is given by the scalar[11]

```
grass_wet = tr wet . fstM . state -- = 44.84%
```

## 9.2 Reversible (quantum) programming

The main purpose of this brief example is to show the role of parametricity in reversible computing, with application to quantum programming. It is well-known that quantum circuits are denoted by unitary, complex matrices. Classic quantum gates are the unitary 0, 1-matrices, that is, matrices representing bijections.

The standard way of building quantum circuits proceeds by composing so-called *universal* gates, typically the Toffoli gate, the C-NOT gate, the Hadamard gate and so on. Such universal gates are given as primitive, but in fact they can be derived from more elementary units via a generic process called *minimal complementation* [Oliveira 2018].

Let us express this using our LAoP library. Suppose one is given a binary function $f :: (A, B) \to B$ that is not *injective* — and therefore not reversible — but it is such that

$$f\ (a, b) = f\ (a, b') \Rightarrow b = b'$$

holds. That is, $f$ is *injective* on the *second* argument once the *first* is fixed (i.e. $f$ is a left-cancelative function). Many functions are of this kind. For instance, multiplication is not injective (cf. $0*a = b*0 = 0$ for different $a, b$) but the function $f\ (x) = a * x$ (fixing the first multiplicand to $a$) is injective (only when $a \neq 0$). Take the exclusive-or Boolean operator as another example, represented by the usual matrix:

$$(Bool, Bool) \xrightarrow{xor} Bool \ = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

This is not injective (cf. e.g. $xor\ (False, True) = xor\ (True, False)$) but, should the first input be restricted to *False* it behaves like the identity on the other input; and if restricted to *True* the behaviour is that of logical negation, cf. in Haskell:

```
xor :: (Bool, Bool) -> Bool
xor (False, b) = b
xor (True, b)  = not b
```

So `xor` is also left-cancelative.

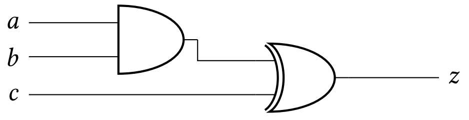
As it can be easily shown below, pairing an arbitrary left-cancellative function $f :: (A, B) \to B$ with projection $fst :: (A, B) \to A$ yields an injective (reversible) function. So we define the following *generic* matrix combinator, where `fstc` abbreviates *first-complement*:

```
fstc :: (...)
     => Matrix e (a, b) b
     -> Matrix e (a,b) (a,b)
fstc m = kr fstM m
```

By applying `fstc` to `xorM = fromF xor`[12] we obtain the reversible matrix

$$(Bool, Bool) \xrightarrow{\texttt{fstc xorM}} (Bool, Bool) \ = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

which is nothing but the so-called C-NOT gate (for *"controlled not"*) which is ubiquitous in quantum programming and is usually depicted as in Fig. 3a. Likewise, the so-called Toffoli gate (Fig. 3b) arises from applying `fstc` to the circuit below.

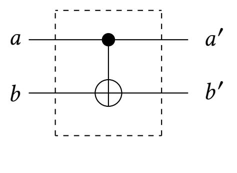


The following piece of code defines both gates in LAoP syntax:

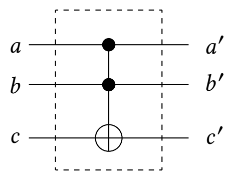
```
toffoli :: (Num e, Ord e)
        => Matrix e ((Bool, Bool), Bool)
                    ((Bool, Bool), Bool)
toffoli = fstc xorM
  where
    xorM = fromF xor
    xor = xor . (uncurry (&&) *** id)
    f (***) g (a,b) = (f a, g b)

cnot :: (Num e, Ord e)
```

[9] So instead of `G = Bool` we have `G = Dry | Wet` and so on.
[10] This generic combinator is inspired in the *left tagging* relational operator of [Bussche 2001].
[11] Recall that scalars are matrices of type () → ().

[12] `fromF`, shown in Listing 1, lifts a function to a matrix, giving 0 or 1 if the output is generated by a given input.



(a) C-NOT gate



(b) Toffoli gate

**Figure 3.** Circuit depictions of the C-NOT and Toffoli gates.

```
     => Matrix e (Bool, Bool) (Bool, Bool)
cnot = fstc (fromF xor)
```

Note the constructive approach here: instead of postulating the (universal quantum) gates and then showing how to use them to implement other logic functions, we start from such logic functions in the first place and then wrap them into a reversible "envelope" using minimal complements.

Also note the numeric parameter `e` free in both gates to be instantiated as needed — typically in the complex numbers, in quantum programming. This is what happens when generating so-called Bell states,

```
bell :: Matrix (Complex Double)
               (Bool, Bool)
               (Bool, Bool)
bell = cnot . (had >< id)
```

where `had` is the Hadamard gate[13]:

```
had :: Matrix (Complex Double) Bool Bool
had = (1/sqrt 2) .| matrixBuilder f
  where
    f (False, _) = 1
    f (True, m)  = bool (-1) 1 m
```

### 9.3 Evaluation

To check whether the inductive approach brings any kind of efficiency benefits, we compared the performance of matrix multiplication algorithms presented in other Haskell libraries and the one proposed in this paper.

By analysing the current ecosystem at the time of writing, namely by filtering data obtained from the Hackage repository, three libraries providing efficient matrix implementations stand out as the most embraced by the community: *hmatrix*, *matrix* and *linear*. The *Criterion* library was used to benchmark the different algorithms on randomly generated square matrices with dimensions ranging between 10 and 1600.

Fig. 4 shows the key features of the testbed environment.

[13] `matrixBuilder`, shown in Listing 1, builds a matrix out of a function that outputs a given cell value.

| | |
|---|---|
| Model | Intel(R) Core(TM)2 Duo CPU P8600 |
| Base clock freq | 2.40GHz |
| L1 cache | 64 KiB |
| L2 cache | 3 MiB |
| RAM | 2 x 4096MB (DDR3) |
| OS | Arch Linux |

**Figure 4.** Testbed environment

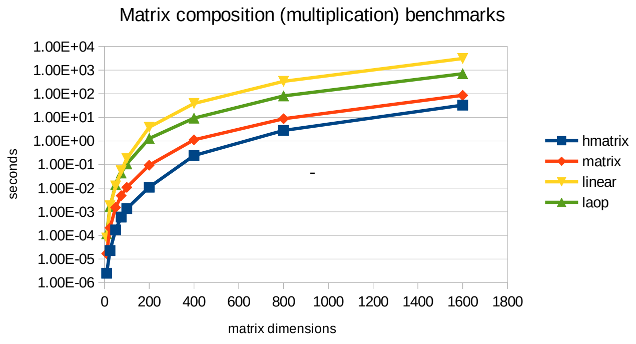


**Figure 5.** Matrix composition benchmarks

As can be seen in the plot of Figure 5, the *hmatrix* and *matrix* libraries are those that perform better. By observing their internal structure, one realises that they are a suitable representation for BLAS/LAPACK computations [Anderson et al. 1999], that is, they have been designed to efficiently exploit caches on modern cache-based architectures. A matrix in the *linear* library is defined as `Vector cols (Vector rows Double)` and does not take into account cache lengths or sizes, so it behaves much worse than the previous ones. Our structure does not take into account any low-level optimisations either, being unable to compete with those that do. Nevertheless, the implementation is *performant for a cache-oblivious approach* and behaves better (almost one order of magnitude better) than other types of simpler definitions.

## 10 Related Work

*Algebraic graphs* [Mokhov 2017] were developed as an alternative to traditional graph representations, such as adjacency lists, with a focus on making it impossible to describe "malformed graphs" where an edge refers to a non-existing vertex. Algebraic graphs come with the following inductive definition:

```
data Graph a where
    Empty   :: Graph a
    Vertex  :: a -> Graph a
    Overlay :: Graph a -> Graph a -> Graph a
    Connect :: Graph a -> Graph a -> Graph a
```

This data type is remarkably similar to our data type `Matrix`: both have the "singleton" primitives, as well as a pair of binary operations. The resulting algebraic structures, however, are very different; instead of relations, algebraic graphs correspond to *endo-relations*, i.e. relations whose domain and codomain coincide. As such, algebraic graphs are not directly applicable to our purposes. Interestingly, in the other direction one can use `Matrix e c r` for representing *weighted bipartite graphs*, i.e. graphs whose vertices are split into two parts `c` and `r` and edges have weights `e`.

Speaking of graphs and their relation to matrices, it is worth mentioning a functional pearl by Dolan [2013] that describes how classic techniques from linear algebra can be used to solve a variety of graph (and non-graph) problems by formulating them as problems on *matrices over semirings*. The paper mostly focuses on semirings and the reuse of ideas across multiple problem domains, making use of a very simple matrix representation:

```
data Matrix a = Scalar a | Matrix [[a]]
```

What is particularly relevant to our work is that the algorithms used in the paper rely on the decomposition of matrices into so-called "block matrices":

```
type BlockMatrix a = (Matrix a, Matrix a,
                      Matrix a, Matrix a)

msplit :: Matrix a -> BlockMatrix a
```

Since matrix dimensions are untyped, the implementation of `msplit` and other matrix-manipulating functions described in the paper requires a great deal of care. By switching to our matrix representation, one can benefit from static correctness guarantees, as well as exploit the inductive matrix definition whenever it needs to be decomposed into blocks.

As far as typed matrices are concerned, Augustsson and Ågren [2016] and Shaikhha and Parreaux [2019] suggest approaches with similar objectives to ours. The results presented are in the form of a relational algebra library around a C++ library and a Scala matrix DSL that is polymorphic only in the content of the matrix. Relations can be seen as matrices and thus can be represented in our library as well, with all the advantages that are to be expected from using the inductive structure. The DSL approach allows one to provide several definitions of the semiring/ring operation relative to the contents of the matrices, but they do not manipulate the matrices inductively, nor are their matrices dimensions polymorphic and statically typed.

Elliott [2018] presents a vocabulary of matrices that introduces a minimal set of rules to build "arbitrary matrices". This vocabulary is consistent with our inductive matrix structure. According to [Elliott 2018], the vocabulary needed from generalised linear maps is exactly that of classes `Category`, `Cartesian`, `Cocartesian` and `Scalable`. Our API could evolve in this direction too.

## 11 Conclusions and Future Work

This paper proposes an inductive, block-oriented matrix definition that can be elegantly manipulated thanks to the algebraic laws of typed linear algebra. A solid, polymorphic, type-safe abstract API is provided that, thanks to the underlying theory, shows that one can create robust and efficient programs that rely on complex matrix manipulations. The examples show how well typed linear algebra blends with the functional programming style, providing an overall type-safe framework for the increased need in linear algebra based applications of our days, in areas such as data analysis and machine learning.

The proposed typed inductive matrix definition leads to efficient, cache-oblivious algorithms, such as matrix composition and transposition, allowing one to reason about code and write it in an elegant, type correct way.

We conclude that, although our encoding does not turn out to be better in terms of performance compared to more efficient libraries, it does not fall short of expectations and the expressiveness and simplicity that it offers justify its adoption.

In future work we plan to fine-tune the implementation in several directions. The block-oriented matrix type brings to mind quadtrees [Samet 1984] and their savings wrt. repetitive cells (pixels). For sparse matrices, which have large blocks of zeros, an improved matrix definition catering for sparsity could be more efficient. Inspired by some of the limitations of the proposed inductive structure, such as the inability to have an unrestricted instance of `Category` and to express large repetitive blocks, alternative formulations are to be explored.

A possible approach could be based on making the type constructors more polymorphic. A new `Identity` constructor could be added and `One` could represent constant blocks of arbitrary sizes, by changing the unit type for polymorphic ones. Since the identity matrix should no longer possess associated dimension type restrictions, it should be possible to implement an unconstrained `Category` instance offering a more efficient composition operation. While we prefer our approach for its simplicity, a more complex inductive type promises advantages in type-safeness and space/time complexity.

Further to researching on a possibly extended encoding, studying how to take advantage of a fully type safe representation via linear map semantics or parallelization strategies to improve performance is also an interesting future direction.

## Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

We would like to thank Andrey Mokhov for his guidance and criticism, and specially for our numerous conversations on different possible interpretations of matrices and their relation to his algebraic graphs data type, as well as its relation with Dolan's matrix representation. This work would likely have remained unpublished without his help and refinement. Friends and colleagues Cláudia Correia, João Pereira, Eduardo Barbosa and several anonymous reviewers provided constructive feedback on this paper, helping to substantially improve it. Last but not least, we want to thank the whole Functional Programming community for having answered a number of code-related questions when developing my library.

## References

Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, and Danny Sorensen. 1999. *LAPACK Users' guide*. Vol. 9. Siam.

Lennart Augustsson and Mårten Ågren. 2016. Experience report: types for a relational algebra library. In *ACM SIGPLAN Notices*. ACM, 127–132. https://doi.org/10.1145/2976002.2976016

Steve Awodey. 2010. *Category Theory* (2nd ed.). Oxford University Press, Inc., New York, NY, USA.

Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A brief overview of Agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 73–78.

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming* 23, 5 (2013), 552–593.

Jan Van den Bussche. 2001. Applications of Alfred Tarski's Ideas in Database Theory. In *CSL'01*. Springer-Verlag, London, UK, 20–37.

Stephen Dolan. 2013. Fun with semirings: a functional pearl on the abuse of linear algebra. In *ICFP'13, Boston*. ACM SIGPLAN Notices, 101–110.

Conal Elliott. 2018. The simple essence of automatic differentiation. *PACMPL* 2, ICFP (2018), 70:1–70:29. https://doi.org/10.1145/3236765

Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*. IEEE, 285–297.

Cordelia V. Hall, Kevin Hammond, Simon L Peyton Jones, and Philip L Wadler. 1996. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18, 2 (1996), 109–138.

Ralf Hinze. 2013. Adjoint folds and unfolds — An extended study. *Science of Computer Programming* 78, 11 (2013), 2108–2159.

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell workshop*, Vol. 1. ACM, 203–233.

Jeff Kramer. 2007. Is Abstraction the Key to Computing? *Commun. ACM* 50, 4 (April 2007), 37–42.

Hugo D. Macedo and José N. Oliveira. 2013. Typing linear algebra: A biproduct-oriented approach. *SCP* 78, 11 (2013), 2160–2191. https://doi.org/10.1016/j.scico.2012.07.012

Saunders MacLane. 1971. *Categories for the Working Mathematician*. Springer-Verlag.

Bartosz Milewski. 2018. *Category theory for programmers*. Textbook available on-line: http://tiny.cc/5letkz.

Andrey Mokhov. 2017. Algebraic Graphs with Class (Functional Pearl). In *Haskell 2017*. ACM, New York, NY, USA, 2–13. https://doi.org/10.1145/3122955.3122956

Andrey Mokhov, Georgy Lukyanov, Simon Marlow, and Jeremie Dimino. 2019. Selective Applicative Functors. *Proc. ACM Program. Lang.* 3, ICFP, Article 90 (July 2019), 29 pages. https://doi.org/10.1145/3341694

José N. Oliveira. 2012. Towards a linear algebra of programming. *Formal Aspects of Computing* 24, 4-6 (2012), 433–458.

José N. Oliveira. 2018. Compiling quantamorphisms for the IBM Q-Experience. Talk at the IFIP WG 2.1 #77 Meeting, Brandenburg (Germany). Joint work with A. Neri and R.S. Barbosa.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. *ACM SIGPLAN Notices* 41, 9 (2006), 50–61.

Hanan Samet. 1984. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)* 16, 2 (1984), 187–260.

Armando Santos. 2020. Selective Functors & Probabilistic Programming. https://github.com/bolt12/master-thesis. (In progress).

Kaz Sato, Cliff Young, and David Patterson. 2017. An in-depth look at Google's first Tensor Processing Unit (TPU). In Google Cloud Big Data and Machine Learning Blog. Access date: June 27, 2020.

Amir Shaikhha and Lionel Parreaux. 2019. Finally, a polymorphic linear algebra language. (2019).

Jan Stolarek, Simon Peyton Jones, and Richard A Eisenberg. 2015. Injective type families for Haskell. *ACM SIGPLAN Notices* 50, 12 (2015), 118–128.

Vasily Volkov and James Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *Proc. SC 2008 ACM/IEEE Conf. on High Performance Computing*. IEEE/ACM, 31. https://doi.org/10.1109/SC.2008.5214359

Wikipedia. 2020. Bayesian network. https://en.wikipedia.org/wiki/Bayesian_network (Accessed: 2020-02-16).