Functional Programming + Dependent Types $\equiv$ Verified Linear Algebra

github.com/ryanorendorff/functional-linear-algebra-talk

Ryan Orendorff

2021

Brief intro: Ryan Orendorff

Research Scientist at Facebook Reality Labs (FRL).

- Passionate about theorem proving, programming language theory and dependently typed languages such as Agda.
- Repos and other talks can be found here: github.com/ryanorendorff/

Disclaimer: This work is done on personal time/equipment and is not sponsored by my employer (past or present), nor is this talk related to work at any employer.

Goal of the talk: correct by construction matrices

We want to be able to enforce that a user cannot create an incorrect matrix, or use a matrix improperly.

Goal of the talk: correct by construction matrices

We want to be able to enforce that a user cannot create an incorrect matrix, or use a matrix improperly.

There are a few ways things can go wrong:

- Improper sizing

```haskell
data Matrix a = Matrix [[a]]
testMatrix = Matrix [[1, 2, 3], [3, 4]]
```

Goal of the talk: correct by construction matrices

We want to be able to enforce that a user cannot create an incorrect matrix, or use a matrix improperly.

There are a few ways things can go wrong:

- Improper sizing

```
data Matrix a = Matrix [[a]]
testMatrix = Matrix [[1, 2, 3], [3, 4]]
```

- Improper data types

```
testMatrix = Matrix [ (Just 5) ]
```

Goal of the talk: correct by construction matrices

We want to be able to enforce that a user cannot create an incorrect matrix, or use a matrix improperly.

There are a few ways things can go wrong:

- Improper sizing

```haskell
data Matrix a = Matrix [[a]]
testMatrix = Matrix [[1, 2, 3], [3, 4]]
```

- Improper data types

```haskell
testMatrix = Matrix [ (Just 5) ]
```

Plus a few more surprising errors to get to later!

First step: define a type for a matrix

A matrix is often a table of numbers, which we can encode in Agda as

```
data MatrixOfNumbers (A : Set) : Set where
    ConstructMatrixOfNumbers : List (List A) → MatrixOfNumbers A
```

First step: define a type for a matrix

A matrix is often a table of numbers, which we can encode in Agda as

```
data MatrixOfNumbers (A : Set) : Set where
    ConstructMatrixOfNumbers : List (List A) → MatrixOfNumbers A
```

which is equivalent to the Haskell

```
data MatrixOfNumbers a = ConstructMatrixOfNumbers [[a]]
```

Given this constructor we can create the following matrix.

$$M_n = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

```
Mₙ : MatrixOfNumbers ℕ -- Natural numbers
Mₙ = ConstructMatrixOfNumbers [ [ 1 , 2 , 3 ] , [ 4 , 5 , 6 ] ]
```

First step: define a type for a matrix

Given this constructor we can create the following matrix.

$$M_n = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

```
Mₙ : MatrixOfNumbers ℕ -- Natural numbers
Mₙ = ConstructMatrixOfNumbers [ [ 1 , 2 , 3 ] , [ 4 , 5 , 6 ] ]
```

Conventions used in this talk : `A` is a type, M_i is a matrix, `m n p q` are natural numbers, and `u v x y` are vectors.

What can we do with a matrix?

A matrix can be used in a few different cases:

1. Multiply a matrix with a vector (matrix-vector multiply): Mx

What can we do with a matrix?

A matrix can be used in a few different cases:

1. Multiply a matrix with a vector (matrix-vector multiply): Mx
2. Transform a matrix to get a new matrix (transpose): $M^T x$

What can we do with a matrix?

A matrix can be used in a few different cases:

1. Multiply a matrix with a vector (matrix-vector multiply): Mx
2. Transform a matrix to get a new matrix (transpose): $M^T x$
3. Combine matrices (matrix-matrix multiply): $M_1 * M_2$

What is matrix-vector multiplication?

Matrix-vector multiply transforms one vector into another through multiplication and addition.

$$\begin{array}{ccccccc} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} & * & \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} & = & \begin{bmatrix} (1*1)+(2*2)+(3*3) \\ (4*1)+(5*2)+(6*3) \end{bmatrix} & = & \begin{bmatrix} 14 \\ 32 \end{bmatrix} \\ M & * & x & = & & & y \end{array}$$

What is matrix-vector multiplication?

Matrix-vector multiply transforms one vector into another through multiplication and addition.

$$\begin{array}{ccccccc} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} & * & \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} & = & \begin{bmatrix} (1*1)+(2*2)+(3*3) \\ (4*1)+(5*2)+(6*3) \end{bmatrix} & = & \begin{bmatrix} 14 \\ 32 \end{bmatrix} \\ M & * & x & = & & & y \end{array}$$

Another way to think of matrix-vector multiplication: M is a *function* from vectors of size 3 to vectors of size 2. This function is sometimes called a *linear map*.

Example of a matrix as a function: identity

The identity matrix converts a vector into the same vector.

$$I * v = \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix} * v = v$$

Example of a matrix as a function: identity

The identity matrix converts a vector into the same vector.

$$I * v = \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix} * v = v$$

If we were to write out the identity matrix as a function, it would be the same as the identity function.

```
list-identity : List A → List A
list-identity l = l
```

Example of a matrix as a function: diag

The diagonal matrix element-wise multiplies one vector with another (written as $*^V$).

$$diag(u) * v = \begin{bmatrix} u_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & u_n \end{bmatrix} * v = u *^V v$$

Example of a matrix as a function: diag

The diagonal matrix element-wise multiplies one vector with another (written as $*^V$).

$$diag(u) * v = \begin{bmatrix} u_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & u_n \end{bmatrix} * v = u *^V v$$

written as a function, this would look like

```
diag : List A → (List A → List A)
diag u = λ v → u *ᵛ v
```

or alternatively as

```
diag u = λ v → zipWith _ (_*_) u v
-- zipWith in Agda has an extra parameter we don't need
```

Let’s define a matrix as a function!

We can define a matrix as a function that takes a vector and returns a new vector.

```
data FunctionalMatrix (A : Set) : Set where
    ConstructFunctionalMatrix : (List A → List A) → FunctionalMatrix A
```

Let's define a matrix as a function!

We can define a matrix as a function that takes a vector and returns a new vector.

```
data FunctionalMatrix (A : Set) : Set where
    ConstructFunctionalMatrix : (List A → List A) → FunctionalMatrix A
```

Now we can construct the identity matrix as follows:

```
Mᵢ : FunctionalMatrix A
Mᵢ = ConstructFunctionalMatrix (list-identity)
```

Let's define a matrix as a function!

We can define a matrix as a function that takes a vector and returns a new vector.

```
data FunctionalMatrix (A : Set) : Set where
    ConstructFunctionalMatrix : (List A → List A) → FunctionalMatrix A
```

Now we can construct the identity matrix as follows:

```
Mᵢ : FunctionalMatrix A
Mᵢ = ConstructFunctionalMatrix (list-identity)
```

This addresses the matrix-vector ability of a matrix, what else can we tackle functionally?

We have matrix-vector multiply down, can we do more?

With our functional definition of a matrix, we can do other operations like matrix-matrix multiply.

$$(M_1 * M_2)v = M_1(M_2(v))$$

We have matrix-vector multiply down, can we do more?

With our functional definition of a matrix, we can do other operations like matrix-matrix multiply.

$$(M_1 * M_2)v = M_1(M_2(v))$$

```
_·ᶠ_ : FunctionalMatrix A → List A → List A
ConstructFunctionalMatrix f ·ᶠ l = f l
```

We have matrix-vector multiply down, can we do more?

With our functional definition of a matrix, we can do other operations like matrix-matrix multiply.

$$(M_1 * M_2)v = M_1(M_2(v))$$

```
_·ᶠ_ : FunctionalMatrix A → List A → List A
ConstructFunctionalMatrix f ·ᶠ l = f l

apply_two_matrices : FunctionalMatrix A → FunctionalMatrix A
                   → List A → List A
apply_two_matrices M₁ M₂ v = M₁ ·ᶠ M₂ ·ᶠ v
```

We have matrix-vector multiply down, can we do more?

With our functional definition of a matrix, we can do other operations like matrix-matrix multiply.

$$(M_1 * M_2)v = M_1(M_2(v))$$

```
_·ᶠ_ : FunctionalMatrix A → List A → List A
ConstructFunctionalMatrix f ·ᶠ l = f l

apply_two_matrices : FunctionalMatrix A → FunctionalMatrix A
                   → List A → List A
apply_two_matrices M₁ M₂ v = M₁ ·ᶠ M₂ ·ᶠ v
```

Hmm that looks a lot like composition:

```
_∘ᶠ_ : FunctionalMatrix A → FunctionalMatrix A → FunctionalMatrix A
M₁ ∘ᶠ M₂ = ConstructFunctionalMatrix (apply_two_matrices M₁ M₂)
```

We often need the transpose matrix at the same time

This type encapsulates the function nature of a matrix, but we often need the transpose as well.

```
data FunctionalMatrixWithTranspose (A : Set) : Set where
    ConstructFMT :  (List A → List A) -- Forward function
                 → (List A → List A) -- Transpose function
                 → FunctionalMatrixWithTranspose A
```

We often need the transpose matrix at the same time

This type encapsulates the function nature of a matrix, but we often need the transpose as well.

```
data FunctionalMatrixWithTranspose (A : Set) : Set where
    ConstructFMT :  (List A → List A) -- Forward function
                 → (List A → List A) -- Transpose function
                 → FunctionalMatrixWithTranspose A
```

We can now define the identity matrix with the transpose matrix function, which is also the identity.

```
Mᵢ,ₜ : FunctionalMatrixWithTranspose A
Mᵢ,ₜ = ConstructFMT (list-identity) (list-identity)
```

Intuition check : convert a functional matrix into a number matrix

For the rest of linear algebra to work, we should always be able to define an equivalent functions using only multiplication and addition.

For example our original identity function

```
identity' : List A → List A
identity' v = v
```

Intuition check : convert a functional matrix into a number matrix

For the rest of linear algebra to work, we should always be able to define an equivalent functions using only multiplication and addition.

For example our original identity function

```
identity' : List A → List A
identity' v = v
```

could be written as

```
identity' : List A → List A
identity' v = replicate (len v) 1 *ⱽ v
```

where `replicate` creates a list of 1s and `*ⱽ` multiplies element-wise.

Is `FunctionalMatrixWithTranspose` “correct by construction”?

Our original goal was

> *Correct by construction linear algebra; equivalent to $\mathbb{R}$ matrices*

Is this true for `FunctionalMatrixWithTranspose`?

Is `FunctionalMatrixWithTranspose` "correct by construction"?

Our original goal was

Correct by construction linear algebra; equivalent to $\mathbb{R}$ matrices

Is this true for `FunctionalMatrixWithTranspose`?

```
f₁ : List ℕ → List ℕ
f₁ v = []
```

```
Mᵣ : FunctionalMatrixWithTranspose ℕ
Mᵣ = ConstructFMT f₁ f₁
```

Is `FunctionalMatrixWithTranspose` “correct by construction”?

Our original goal was

> *Correct by construction linear algebra; equivalent to $\mathbb{R}$ matrices*

Is this true for `FunctionalMatrixWithTranspose`?

```
f₁ : List ℕ → List ℕ
f₁ v = []


Mᵣ : FunctionalMatrixWithTranspose ℕ
Mᵣ = ConstructFMT f₁ f₁
```

Hmm, intuition check: can we write M_r as a matrix of numbers?

Is `FunctionalMatrixWithTranspose` "correct by construction"?

Our original goal was

Correct by construction linear algebra; equivalent to $\mathbb{R}$ matrices

Is this true for `FunctionalMatrixWithTranspose`?

```
f₁ : List ℕ → List ℕ
f₁ v = []


Mᵣ : FunctionalMatrixWithTranspose ℕ
Mᵣ = ConstructFMT f₁ f₁
```

Hmm, intuition check: can we write $\mathtt{M_r}$ as a matrix of numbers?

$$M_r = \begin{bmatrix} \text{throw away data} \\ \text{throw away data} \end{bmatrix}$$

Encoding the length of the vector in the type

Agda allows us to specify what the length of a vector as part of the type.[1]

```
v : Vec ℕ 3
v = [ 1 , 2 , 3 ]ᵛ
```

[1]This is a bit of a misnomer; the difference between term and type is muddled in many dependently typed languages.

Encoding the length of the vector in the type

Agda allows us to specify what the length of a vector as part of the type.[1]

```
v : Vec ℕ 3
v = [ 1 , 2 , 3 ]ᵛ
```

If we try to create a vector of the wrong length, Agda will tell us.

```
v₂ : Vec ℕ 2
v₂ = v
-- Get the following error: 3 ≠ 2 of type ℕ
```

[1]This is a bit of a misnomer; the difference between term and type is muddled in many dependently typed languages.

Encoding the length of the vector in the type

Agda allows us to specify what the length of a vector as part of the type.[1]

```
v : Vec ℕ 3
v = [ 1 , 2 , 3 ]ᵛ
```

If we try to create a vector of the wrong length, Agda will tell us.

```
v₂ : Vec ℕ 2
v₂ = v
-- Get the following error: 3 ≠ 2 of type ℕ
```

`Vec` is a *dependent type* because its type *depends on a value*.

[1]This is a bit of a misnomer; the difference between term and type is muddled in many dependently typed languages.

Use functions on Vec to ensure that the shapes match

We can define a matrix type where the shapes are preserved.

```
data SizedMatrix (A : Set) (m n : ℕ) : Set where
    ConstructSizedMatrix :  (Vec A n → Vec A m) -- Forward function
                         → (Vec A m → Vec A n) -- Transpose function
                         → SizedMatrix A m n
```

Previously this would be done with a runtime check.

Use functions on Vec to ensure that the shapes match

We can define a matrix type where the shapes are preserved.

```
data SizedMatrix (A : Set) (m n : ℕ) : Set where
    ConstructSizedMatrix :  (Vec A n → Vec A m) -- Forward function
                         → (Vec A m → Vec A n) -- Transpose function
                         → SizedMatrix A m n
```

Previously this would be done with a runtime check.

In Haskell, we would write this as

```
data SizedMatrix (A :: *) (m :: Nat) (n :: Nat) where
    ConstructSizedMatrix :: (KnownNat m, KnownNat n)
                         ⇒ (Vec A n → Vec A m) -- Forward function
                         → (Vec A m → Vec A n) -- Transpose function
                         → SizedMatrix A m n
```

Use functions on Vec to ensure that the shapes match

We can define a matrix type where the shapes are preserved.

```
data SizedMatrix (A : Set) (m n : ℕ) : Set where
    ConstructSizedMatrix :  (Vec A n → Vec A m) -- Forward function
                         → (Vec A m → Vec A n) -- Transpose function
                         → SizedMatrix A m n
```

Use functions on Vec to ensure that the shapes match

We can define a matrix type where the shapes are preserved.

```
data SizedMatrix (A : Set) (m n : ℕ) : Set where
    ConstructSizedMatrix :  (Vec A n → Vec A m) -- Forward function
                          → (Vec A m → Vec A n) -- Transpose function
                          → SizedMatrix A m n
```

We can now define our identity matrix again.

```
id : (A : Set) → A → A
```

```
Mᵢ,ₛ : SizedMatrix A n n
Mᵢ,ₛ = ConstructSizedMatrix id id -- id : Vec A n → Vec A n
```

Intuition check: can we encode matrices that are not possible to write out?

Our original goal was

> *Correct by construction linear algebra; equivalent to $\mathbb{R}$ matrices*

Intuition check: can we encode matrices that are not possible to write out?

Our original goal was

> *Correct by construction linear algebra; equivalent to $\mathbb{R}$ matrices*

We could write a matrix for handling playing cards.

```
data Card : Set where
  ♠ ♣ ♡ ♢ : Card
```

Intuition check: can we encode matrices that are not possible to write out?

Our original goal was

> *Correct by construction linear algebra; equivalent to $\mathbb{R}$ matrices*

We could write a matrix for handling playing cards.

```
data Card : Set where
  ♠ ♣ ♡ ♢ : Card
M♠ : SizedMatrix Card n n
M♠ = ConstructSizedMatrix (λ v → replicate ♠) (λ v → replicate ♡)
```

Intuition check: can we encode matrices that are not possible to write out?

Our original goal was

> *Correct by construction linear algebra; equivalent to $\mathbb{R}$ matrices*

We could write a matrix for handling playing cards.

```
data Card : Set where
  ♠ ♣ ♡ ◊ : Card
M♠ : SizedMatrix Card n n
M♠ = ConstructSizedMatrix (λ v → replicate ♠) (λ v → replicate ♡)
```

If we wanted to convert this to multiplication and addition only....

$$M\spadesuit = \begin{bmatrix} \clubsuit & \heartsuit \\ \diamondsuit & \spadesuit \end{bmatrix}$$

Matrices cannot contain just anything! The elements have to be able to be added/multiplied.

Matrices are defined over fields

To check our intuition we have been trying to determine if our function could be written using multiplication and addition. Formally, this is equivalent to saying the elements of a matrix form a *Field*.

Matrices are defined over fields

To check our intuition we have been trying to determine if our function could be written using multiplication and addition. Formally, this is equivalent to saying the elements of a matrix form a *Field*.

```
record Field (A : Set) : Set where
  field
    _+_ : A → A → A -- 3 + 4
    _*_ : A → A → A -- 3 * 4
```

Matrices are defined over fields

To check our intuition we have been trying to determine if our function could be written using multiplication and addition. Formally, this is equivalent to saying the elements of a matrix form a *Field*.

```
record Field (A : Set) : Set where
  field
    _+_ : A → A → A -- 3 + 4
    _*_ : A → A → A -- 3 * 4

    -_  : A → A -- + inverse, - 4
    _⁻¹ : A → A -- * inverse, 4 ⁻¹
```

Matrices are defined over fields

To check our intuition we have been trying to determine if our function could be written using multiplication and addition. Formally, this is equivalent to saying the elements of a matrix form a *Field*.

```
record Field (A : Set) : Set where
  field
    _+_ : A → A → A -- 3 + 4
    _*_ : A → A → A -- 3 * 4

    -_  : A → A -- + inverse, - 4
    _⁻¹ : A → A -- * inverse, 4 ⁻¹

    0ᶠ  : A -- Identity of _+_, 4 + 0f = 4
    1ᶠ  : A -- Identity of _*_, 4 * 1f = 4
```

We can define matrices that operate on Fields only

We can restrict our `A` type to having a defined version of `+` and `*`.

```
data SizedFieldMatrix (A : Set) ⦃ F : Field A ⦄ (m n : ℕ) : Set where
    ConstructSFM :  (Vec A n → Vec A m) -- Forward function
                 → (Vec A m → Vec A n) -- Transpose function
                 → SizedFieldMatrix A m n
```

We can define matrices that operate on Fields only

We can restrict our A type to having a defined version of + and *.

```
data SizedFieldMatrix (A : Set) ⦃ F : Field A ⦄ (m n : ℕ) : Set where
    ConstructSFM :  (Vec A n → Vec A m) -- Forward function
                 → (Vec A m → Vec A n) -- Transpose function
                 → SizedFieldMatrix A m n
```

in Haskell this would be written as

```
data SizedFieldMatrix A (m :: Nat) (n :: Nat) where
    ConstructSFM :: (KnownNat m, KnownNat n, Field A)
                 => (Vec A n → Vec A m) -- Forward function
                 -> (Vec A m → Vec A n) -- Transpose function
                 -> SizedFieldMatrix A m n
```

We can define matrices that operate on Fields only

We can restrict our A type to having a defined version of + and *.

```
data SizedFieldMatrix (A : Set) ⦃ F : Field A ⦄ (m n : ℕ) : Set where
    ConstructSFM :  (Vec A n → Vec A m) -- Forward function
                 → (Vec A m → Vec A n) -- Transpose function
                 → SizedFieldMatrix A m n
```

The card example can no longer be constructed, but the identity matrix still can be constructed.

```
-- + and * must be defined on A
Mₛᶠᵢ : ⦃ F : Field A ⦄ → SizedFieldMatrix A n n
Mₛᶠᵢ = ConstructSFM id id
```

Intuition check: can we encode matrices that are not possible to write out?

Our original goal was

Correct by construction linear algebra; equivalent to $\mathbb{R}$ *matrices*

Intuition check: can we encode matrices that are not possible to write out?

Our original goal was

Correct by construction linear algebra; equivalent to ℝ matrices

```
Mᵣₑₚ : ⦃ F : Field A ⦄ → SizedFieldMatrix A m n
Mᵣₑₚ = ConstructSFM (λ v → replicate 1ᶠ) (λ v → replicate 1ᶠ)
```

Intuition check: can we encode matrices that are not possible to write out?

Our original goal was

> *Correct by construction linear algebra; equivalent to $\mathbb{R}$ matrices*

```
Mᵣₑₚ : ⦃ F : Field A ⦄ → SizedFieldMatrix A m n
Mᵣₑₚ = ConstructSFM (λ v → replicate 1ᶠ) (λ v → replicate 1ᶠ)
```

Using only multiplication and addition does not allow us to *produce constant outputs for any input.*

Matrices are linear functions

Matrices have the following properties that we'd like to preserve:

- Linearity: $M(u +^V v) = M(u) +^V M(v)$

Matrices are linear functions

Matrices have the following properties that we'd like to preserve:

- Linearity: $M(u +^V v) = M(u) +^V M(v)$
- Homogeneity : $M(c \circ^V v) = c \circ^V M(v)$

Matrices are linear functions

Matrices have the following properties that we'd like to preserve:

- Linearity: $M(u +^V v) = M(u) +^V M(v)$
- Homogeneity : $M(c \circ^V v) = c \circ^V M(v)$

Currently we could define a matrix like so, which has neither property.

```
_ : ⦃ F : Field A ⦄ → SizedFieldMatrix A n n
_ = ConstructSFM (λ v → replicate 1ᶠ) (λ v → replicate 1ᶠ)
```

Matrices are linear functions

Matrices have the following properties that we'd like to preserve:

- Linearity: $M(u +^V v) = M(u) +^V M(v)$
- Homogeneity : $M(c \circ^V v) = c \circ^V M(v)$

Currently we could define a matrix like so, which has neither property.

```
_ : ⦃ F : Field A ⦄ → SizedFieldMatrix A n n
_ = ConstructSFM (λ v → replicate 1ᶠ) (λ v → replicate 1ᶠ)
```

- Linearity : $f(u +^V v) = \vec{1} \;\neq\; f(u) +^V f(v) = \vec{1} +^V \vec{1} = \vec{2}$

Matrices are linear functions

Matrices have the following properties that we'd like to preserve:

- Linearity: $M(u +^V v) = M(u) +^V M(v)$
- Homogeneity : $M(c \circ^V v) = c \circ^V M(v)$

Currently we could define a matrix like so, which has neither property.

```
_ : ⦃ F : Field A ⦄ → SizedFieldMatrix A n n
_ = ConstructSFM (λ v → replicate 1ᶠ) (λ v → replicate 1ᶠ)
```

- Linearity : $f(u +^V v) = \vec{1} \;\neq\; f(u) +^V f(v) = \vec{1} +^V \vec{1} = \vec{2}$
- Homogeneity : $f(c \circ^V v) = \vec{1} \;\neq\; c \circ^V f(\vec{v}) = c \circ^V \vec{1} = \vec{c}$

How do we ensure that our functions are linear?

For our matrices to make sense, we need the functions that are used for the forward and transpose functions to be linear functions.

```
-- A linear function (aka a linear map)
record _⊸_ {A : Set} ⦃ F : Field A ⦄ (m n : ℕ) : Set where
  field
    f : (Vec A m → Vec A n)
```

How do we ensure that our functions are linear?

For our matrices to make sense, we need the functions that are used for the forward and transpose functions to be linear functions.

```
-- A linear function (aka a linear map)
record _⊸_ {A : Set} ⦃ F : Field A ⦄ (m n : ℕ) : Set where
  field
    f : (Vec A m → Vec A n)
    f[u+v]≡f[u]+f[v] : (u v : Vec A m) → f (u +ⱽ v) ≡ f u +ⱽ f v
```

How do we ensure that our functions are linear?

For our matrices to make sense, we need the functions that are used for the forward and transpose functions to be linear functions.

```
-- A linear function (aka a linear map)
record _⊸_ {A : Set} ⦃ F : Field A ⦄ (m n : ℕ) : Set where
  field
    f : (Vec A m → Vec A n)
    f[u+v]≡f[u]+f[v] : (u v : Vec A m) → f (u +ⱽ v) ≡ f u +ⱽ f v
    f[c*v]≡c*f[v] : (c : A) → (v : Vec A m) → f (c ∘ⱽ v) ≡ c ∘ⱽ (f v)
```

How do we ensure that our functions are linear?

For our matrices to make sense, we need the functions that are used for the forward and transpose functions to be linear functions.

```
-- A linear function (aka a linear map)
record _⊸_ {A : Set} ⦃ F : Field A ⦄ (m n : ℕ) : Set where
  field
    f : (Vec A m → Vec A n)
    f[u+v]≡f[u]+f[v] : (u v : Vec A m) → f (u +ⱽ v) ≡ f u +ⱽ f v
    f[c*v]≡c*f[v] : (c : A) → (v : Vec A m) → f (c ∘ⱽ v) ≡ c ∘ⱽ (f v)
```

with this we could define our matrices using linear functions.

```
data LinearMatrix {A : Set} ⦃ F : Field A ⦄ (m n : ℕ) : Set where
  ConstructLinearMatrix : (n ⊸ m) → (m ⊸ n) → LinearMatrix m n
```

What is this ≡ thing?

The ≡ sign means that two things are equal[2] in the sense that the left and the right side are written with the same order of constructors[3].

[2] Homogenously
[3] Their normal forms are equivalent

What is this ≡ thing?

The ≡ sign means that two things are equal[2] in the sense that the left and the right side are written with the same order of constructors[3].

The definition of ≡ is

```
data _≡_ (x : A) : A → Set where
  refl : x ≡ x
```

[2] Homogenously

[3] Their normal forms are equivalent

Fields must follow some properties on top of defining + and *

Fields define more than just + and *; a field has some properties.

```
+-assoc   : (a b c : A) → a + (b + c) ≡ (a + b) + c
+-comm    : (a b : A)   → a + b ≡ b + a
+0ᶠ       : (a : A)     → a + 0ᶠ ≡ a
+-inv     : (a : A)     → (- a) + a ≡ 0ᶠ
```

Fields must follow some properties on top of defining + and *

Fields define more than just + and *; a field has some properties.

```
+-assoc   : (a b c : A) → a + (b + c) ≡ (a + b) + c
+-comm    : (a b : A)   → a + b ≡ b + a
+0ᶠ       : (a : A)     → a + 0ᶠ ≡ a
+-inv     : (a : A)     → (- a) + a ≡ 0ᶠ

*-assoc   : (a b c : A) → a * (b * c) ≡ (a * b) * c
*-comm    : (a b : A)   → a * b ≡ b * a
*1ᶠ       : (a : A)     → a * 1ᶠ ≡ a
*-inv     : (a : A)     → (a ≢ 0ᶠ) → (a ⁻¹) * a ≡ 1ᶠ
```

Fields must follow some properties on top of defining + and *

Fields define more than just + and *; a field has some properties.

```
+-assoc    : (a b c : A) → a + (b + c) ≡ (a + b) + c
+-comm     : (a b : A)   → a + b ≡ b + a
+0ᶠ        : (a : A)     → a + 0ᶠ ≡ a
+-inv      : (a : A)     → (- a) + a ≡ 0ᶠ

*-assoc    : (a b c : A) → a * (b * c) ≡ (a * b) * c
*-comm     : (a b : A)   → a * b ≡ b * a
*1ᶠ        : (a : A)     → a * 1ᶠ ≡ a
*-inv      : (a : A)     → (a ≢ 0ᶠ) → (a ⁻¹) * a ≡ 1ᶠ

*-distr-+ : (a b c : A) → a * (b + c) ≡ (a * b) + (a * c)
```

How do we generate these proofs? By composing them!

We can prove `(b + 0ᶠ) * 1ᶠ ≡ b` from simpler (axiomatic) statements.

```
new_proof : (b : A) → (b + 0ᶠ) * 1ᶠ ≡ b
new_proof b = begin
  (b + 0ᶠ) * 1ᶠ
```

How do we generate these proofs? By composing them!

We can prove `(b + 0ᶠ) * 1ᶠ ≡ b` from simpler (axiomatic) statements.

```
new_proof : (b : A) → (b + 0ᶠ) * 1ᶠ ≡ b
new_proof b = begin
  (b + 0ᶠ) * 1ᶠ
  ≡⟨ *1ᶠ (b + 0ᶠ) ⟩ -- *1ᶠ : (a : A) → a * 1ᶠ ≡ a
```

How do we generate these proofs? By composing them!

We can prove `(b + 0ᶠ) * 1ᶠ ≡ b` from simpler (axiomatic) statements.

```
new_proof : (b : A) → (b + 0ᶠ) * 1ᶠ ≡ b
new_proof b = begin
  (b + 0ᶠ) * 1ᶠ
  ≡⟨ *1ᶠ (b + 0ᶠ) ⟩ -- *1ᶠ : (a : A) → a * 1ᶠ ≡ a
  b + 0ᶠ
```

How do we generate these proofs? By composing them!

We can prove `(b + 0f) * 1f ≡ b` from simpler (axiomatic) statements.

```
new_proof : (b : A) → (b + 0f) * 1f ≡ b
new_proof b = begin
  (b + 0f) * 1f
  ≡⟨ *1f (b + 0f) ⟩ -- *1f : (a : A) → a * 1f ≡ a
  b + 0f
  ≡⟨ +0f b ⟩ -- +0f : (a : A) → a + 0f ≡ a
```

How do we generate these proofs? By composing them!

We can prove `(b + 0ᶠ) * 1ᶠ ≡ b` from simpler (axiomatic) statements.

```
new_proof : (b : A) → (b + 0ᶠ) * 1ᶠ ≡ b
new_proof b = begin
  (b + 0ᶠ) * 1ᶠ
  ≡⟨ *1ᶠ (b + 0ᶠ) ⟩ -- *1ᶠ : (a : A) → a * 1ᶠ ≡ a
  b + 0ᶠ
  ≡⟨ +0ᶠ b ⟩ -- +0ᶠ : (a : A) → a + 0ᶠ ≡ a
  b ∎
```

Proving that the identity function is linear

The linear identity function is simple

```
id₁ : ⦃ F : Field A ⦄ → n ⊸ n
id₁ = record
  { f = id -- Vec A n → Vec A n
```

Proving that the identity function is linear

The linear identity function is simple

```
id₁ : ⦃ F : Field A ⦄ → n ⊸ n
id₁ = record
  { f = id -- Vec A n → Vec A n

  ; f[u+v]≡f[u]+f[v] = λ u v → refl -- id (u +ⱽ v) ≡ id u +ⱽ id v
  -- Reflexive because Agda can apply all three `id` and conclude
  -- u +ⱽ v ≡ u +ⱽ v
```

Proving that the identity function is linear

The linear identity function is simple

```
id₁ : ⦃ F : Field A ⦄ → n ⊸ n
id₁ = record
  { f = id -- Vec A n → Vec A n

  ; f[u+v]≡f[u]+f[v] = λ u v → refl -- id (u +ⱽ v) ≡ id u +ⱽ id v
  -- Reflexive because Agda can apply all three `id` and conclude
  -- u +ⱽ v ≡ u +ⱽ v

  ; f[c*v]≡c*f[v] = λ c v → refl -- id (c ∘ⱽ v) ≡ c ∘ⱽ id v
  -- Reflexivity for same reason as above proof.
  }
```

Proving that the **diag** function is linear

Now let's try to define the **diag** function as a linear function

```
diag₁ : ⦃ F : Field A ⦄ → Vec A n → n ⊸ n
diag₁ d = record
  { f = d *ⱽ_
```

Proving that the **diag** function is linear

Now let's try to define the **diag** function as a linear function

```
diag₁ : ⦃ F : Field A ⦄ → Vec A n → n ⊸ n
diag₁ d = record
  { f = d *ⱽ_

  -- *ⱽ-distr-+ⱽ : d *ⱽ (u +ⱽ v) ≡ d *ⱽ u +ⱽ d *ⱽ v
  ; f[u+v]≡f[u]+f[v] = λ u v → *ⱽ-distr-+ⱽ d u v
```

Proving that the **diag** function is linear

Now let's try to define the **diag** function as a linear function

```
diag₁ : ⦃ F : Field A ⦄ → Vec A n → n ⊸ n
diag₁ d = record
  { f = d *ⱽ_

  -- *ⱽ-distr-+ⱽ : d *ⱽ (u +ⱽ v) ≡ d *ⱽ u +ⱽ d *ⱽ v
  ; f[u+v]≡f[u]+f[v] = λ u v → *ⱽ-distr-+ⱽ d u v

  -- *ⱽ∘ⱽ≡∘ⱽ*ⱽ : d *ⱽ (c ∘ⱽ v) ≡ c ∘ⱽ (d *ⱽ v)
  ; f[c*v]≡c*f[v] = λ c v → *ⱽ∘ⱽ≡∘ⱽ*ⱽ c d v
  }
```

Let's go through the linearity proof for **diag**

To prove linearity for `diag`, let's step through the proof.

```
*ᵛ-distr-+ᵛ' : (d u v : Vec A n)
             → d *ᵛ (u +ᵛ v) ≡ d *ᵛ u +ᵛ d *ᵛ v
```

Let's go through the linearity proof for **diag**

To prove linearity for **diag**, let's step through the proof.

```
*ⱽ-distr-+ⱽ' : (d u v : Vec A n)
            → d *ⱽ (u +ⱽ v) ≡ d *ⱽ u +ⱽ d *ⱽ v
*ⱽ-distr-+ⱽ' []ⱽ []ⱽ []ⱽ = refl
```

Let's go through the linearity proof for **diag**

To prove linearity for **diag**, let's step through the proof.

```
*ⱽ-distr-+ⱽ' : (d u v : Vec A n)
            → d *ⱽ (u +ⱽ v) ≡ d *ⱽ u +ⱽ d *ⱽ v
*ⱽ-distr-+ⱽ' []ⱽ []ⱽ []ⱽ = refl
*ⱽ-distr-+ⱽ' (d₀ ::ⱽ dᵣ) (u₀ ::ⱽ uᵣ) (v₀ ::ⱽ vᵣ) = begin
    (d₀ * (u₀ + v₀)) ::ⱽ (dᵣ *ⱽ (uᵣ +ⱽ vᵣ))
```

Let's go through the linearity proof for **diag**

To prove linearity for **diag**, let's step through the proof.

```
*ᵛ-distr-+ᵛ' : (d u v : Vec A n)
            → d *ᵛ (u +ᵛ v) ≡ d *ᵛ u +ᵛ d *ᵛ v
*ᵛ-distr-+ᵛ' []ᵛ []ᵛ []ᵛ = refl
*ᵛ-distr-+ᵛ' (d₀ ::ᵛ dᵣ) (u₀ ::ᵛ uᵣ) (v₀ ::ᵛ vᵣ) = begin
    (d₀ * (u₀ + v₀)) ::ᵛ (dᵣ *ᵛ (uᵣ +ᵛ vᵣ))
  ≡⟨ cong ((d₀ * (u₀ + v₀)) ::ᵛ_) (*ᵛ-distr-+ᵛ' dᵣ uᵣ vᵣ) ⟩
```

Let's go through the linearity proof for **diag**

To prove linearity for **diag**, let's step through the proof.

```
*ⱽ-distr-+ⱽ' : (d u v : Vec A n)
             → d *ⱽ (u +ⱽ v) ≡ d *ⱽ u +ⱽ d *ⱽ v
*ⱽ-distr-+ⱽ' []ⱽ []ⱽ []ⱽ = refl
*ⱽ-distr-+ⱽ' (d₀ ::ⱽ dᵣ) (u₀ ::ⱽ uᵣ) (v₀ ::ⱽ vᵣ) = begin
    (d₀ * (u₀ + v₀)) ::ⱽ (dᵣ *ⱽ (uᵣ +ⱽ vᵣ))
  ≡⟨ cong ((d₀ * (u₀ + v₀)) ::ⱽ_) (*ⱽ-distr-+ⱽ' dᵣ uᵣ vᵣ) ⟩
    (d₀ * (u₀ + v₀)) ::ⱽ (dᵣ *ⱽ uᵣ +ⱽ dᵣ *ⱽ vᵣ)
```

Let's go through the linearity proof for **diag**

To prove linearity for **diag**, let's step through the proof.

```
*ⱽ-distr-+ⱽ' : (d u v : Vec A n)
            → d *ⱽ (u +ⱽ v) ≡ d *ⱽ u +ⱽ d *ⱽ v
*ⱽ-distr-+ⱽ' []ⱽ []ⱽ []ⱽ = refl
*ⱽ-distr-+ⱽ' (d₀ ::ⱽ dᵣ) (u₀ ::ⱽ uᵣ) (v₀ ::ⱽ vᵣ) = begin
    (d₀ * (u₀ + v₀)) ::ⱽ (dᵣ *ⱽ (uᵣ +ⱽ vᵣ))
  ≡⟨ cong ((d₀ * (u₀ + v₀)) ::ⱽ_) (*ⱽ-distr-+ⱽ' dᵣ uᵣ vᵣ) ⟩
    (d₀ * (u₀ + v₀)) ::ⱽ (dᵣ *ⱽ uᵣ +ⱽ dᵣ *ⱽ vᵣ)
  ≡⟨ cong (_::ⱽ (dᵣ *ⱽ uᵣ +ⱽ dᵣ *ⱽ vᵣ)) (*-distr-+ d₀ u₀ v₀) ⟩
```

Let's go through the linearity proof for **diag**

To prove linearity for **diag**, let's step through the proof.

```
*ⱽ-distr-+ⱽ' : (d u v : Vec A n)
             → d *ⱽ (u +ⱽ v) ≡ d *ⱽ u +ⱽ d *ⱽ v
*ⱽ-distr-+ⱽ' []ⱽ []ⱽ []ⱽ = refl
*ⱽ-distr-+ⱽ' (d₀ ::ⱽ dᵣ) (u₀ ::ⱽ uᵣ) (v₀ ::ⱽ vᵣ) = begin
    (d₀ * (u₀ + v₀)) ::ⱽ (dᵣ *ⱽ (uᵣ +ⱽ vᵣ))
  ≡⟨ cong ((d₀ * (u₀ + v₀)) ::ⱽ_) (*ⱽ-distr-+ⱽ' dᵣ uᵣ vᵣ) ⟩
    (d₀ * (u₀ + v₀)) ::ⱽ (dᵣ *ⱽ uᵣ +ⱽ dᵣ *ⱽ vᵣ)
  ≡⟨ cong (_::ⱽ (dᵣ *ⱽ uᵣ +ⱽ dᵣ *ⱽ vᵣ)) (*-distr-+ d₀ u₀ v₀) ⟩
    (d₀ * u₀ + d₀ * v₀) ::ⱽ (dᵣ *ⱽ uᵣ +ⱽ dᵣ *ⱽ vᵣ)
```

Let’s go through the linearity proof for **diag**

To prove linearity for **diag**, let’s step through the proof.

```
*ⱽ-distr-+ⱽ' : (d u v : Vec A n)
             → d *ⱽ (u +ⱽ v) ≡ d *ⱽ u +ⱽ d *ⱽ v
*ⱽ-distr-+ⱽ' []ⱽ []ⱽ []ⱽ = refl
*ⱽ-distr-+ⱽ' (d₀ ::ⱽ dᵣ) (u₀ ::ⱽ uᵣ) (v₀ ::ⱽ vᵣ) = begin
    (d₀ * (u₀ + v₀)) ::ⱽ (dᵣ *ⱽ (uᵣ +ⱽ vᵣ))
  ≡⟨ cong ((d₀ * (u₀ + v₀)) ::ⱽ_) (*ⱽ-distr-+ⱽ' dᵣ uᵣ vᵣ) ⟩
    (d₀ * (u₀ + v₀)) ::ⱽ (dᵣ *ⱽ uᵣ +ⱽ dᵣ *ⱽ vᵣ)
  ≡⟨ cong (_::ⱽ (dᵣ *ⱽ uᵣ +ⱽ dᵣ *ⱽ vᵣ)) (*-distr-+ d₀ u₀ v₀) ⟩
    (d₀ * u₀ + d₀ * v₀) ::ⱽ (dᵣ *ⱽ uᵣ +ⱽ dᵣ *ⱽ vᵣ)
  ≡⟨⟩
    (d₀ ::ⱽ dᵣ) *ⱽ (u₀ ::ⱽ uᵣ) +ⱽ (d₀ ::ⱽ dᵣ) *ⱽ (v₀ ::ⱽ vᵣ) ∎
```

We can finally define a matrix made up of linear functions.

```
data LinearMatrix {A : Set} ⦃ F : Field A ⦄ (m n : ℕ) : Set where
  ConstructLinearMatrix : (n ⊸ m) → (m ⊸ n) → LinearMatrix m n

id-linear : ⦃ F : Field A ⦄ → LinearMatrix n n
id-linear = ConstructLinearMatrix idₗ idₗ
```

How we can finally define LinearMatrix!

We can finally define a matrix made up of linear functions.

```
data LinearMatrix {A : Set} ⦃ F : Field A ⦄ (m n : ℕ) : Set where
  ConstructLinearMatrix : (n ⊸ m) → (m ⊸ n) → LinearMatrix m n

id-linear : ⦃ F : Field A ⦄ → LinearMatrix n n
id-linear = ConstructLinearMatrix idₗ idₗ
```

Have we reached “Correct by construction linear algebra”?

Does the transpose match?

Say we defined a matrix as so

```
Mₙₒ : ⦃ F : Field A ⦄ → LinearMatrix n n
Mₙₒ = ConstructLinearMatrix (id₁) (diag₁ (replicate 0ᶠ))
```

Does the transpose match?

Say we defined a matrix as so

```
Mₙₒ : ⦃ F : Field A ⦄ → LinearMatrix n n
Mₙₒ = ConstructLinearMatrix (id₁) (diag₁ (replicate 0ᶠ))
```

We have mixed up the forward/transpose pairing.

$$I = I^T$$

$$diag(v) = diag(v)^T$$

Does the transpose match?

Say we defined a matrix as so

```
Mₙₒ : ⦃ F : Field A ⦄ → LinearMatrix n n
Mₙₒ = ConstructLinearMatrix (id₁) (diag₁ (replicate 0ᶠ))
```

We have mixed up the forward/transpose pairing.

$$I = I^T$$

$$diag(v) = diag(v)^T$$

To solve this problem, we can show that for forward function `M` and transpose function `Mᵀ` that the following property holds.

$$\forall xy. \langle x, My \rangle = \langle y, M^T x \rangle$$

$$\langle a, b \rangle = \mathrm{sum}(a *^V b) = \sum_i^n a_i * b_i$$

Finally we reach our goal of correct by construction matrices!

If we require the user to prove the inner product property, we can *finally* create a “correct by construction” functional matrix.

```
data Mat_×_ {A : Set} ⦃ F : Field A ⦄ (m n : ℕ) : Set where
  ⟦_,_,_⟧ :  (M  : n ⊸ m )
          → (Mᵀ : m ⊸ n )
          → (p : (x : Vec A m) → (y : Vec A n)
                → ⟨ x , M ·ˡ y ⟩ ≡ ⟨ y , Mᵀ ·ˡ x ⟩ )
          → Mat m × n
```

where the inner product (⟨⟩) is defined as

```
⟨_,_⟩ : ⦃ F : Field A ⦄ → Vec A n → Vec A n → A
⟨ x , y ⟩ = sum (x *ⱽ y)
-- Sum of the element-wise multiply of two vectors
```

The final identity functional matrix

With this, we can finally define the identity matrix in a way that is not possible to make an error.

```
Mᴵ : ⦃ F : Field A ⦄ → Mat n × n
Mᴵ = ⟦ id₁ , id₁ , id-transpose  ⟧
  where

    id-transpose : ⦃ F : Field A ⦄ (x y : Vec A n)
                 → ⟨ x , id y ⟩ ≡ ⟨ y , id x ⟩
```

The final identity functional matrix

With this, we can finally define the identity matrix in a way that is not possible to make an error.

```
Mᴵ : ⦃ F : Field A ⦄ → Mat n × n
Mᴵ = ⟦ id₁ , id₁ , id-transpose  ⟧
  where

    id-transpose : ⦃ F : Field A ⦄ (x y : Vec A n)
                 → ⟨ x , id y ⟩ ≡ ⟨ y , id x ⟩

    id-transpose x y = begin
      ⟨ x , id y ⟩ ≡⟨⟩
      ⟨ x , y ⟩    ≡⟨ ⟨⟩-comm x y ⟩
      ⟨ y , x ⟩    ≡⟨⟩
      ⟨ y , id x ⟩ ∎
```

What can we do with a matrix

We can do a few things with a matrix:

1. Multiply the matrix with a vector (matrix-vector multiply): Mx
2. Transform the matrix to get a new matrix (transpose): $M^T x$
3. Combine matrices (matrix-matrix multiply): $M_1 * M_2$

We can do a few things with a matrix:

1. Multiply the matrix with a vector (matrix-vector multiply): Mx
2. Transform the matrix to get a new matrix (transpose): $M^T x$
3. Combine matrices (matrix-matrix multiply): $M_1 * M_2$

We have not done matrix-matrix multiplication, can we implement it with our new definition?

Implementing matrix-matrix multiply on functional matrices

Previously, we were able to define matrix-matrix multiplication

$$M_1 * M_2$$

using function composition

```
apply_two_matrices : FunctionalMatrix A → FunctionalMatrix A
                   → List A → List A
apply_two_matrices F G v = F ·ᶠ G ·ᶠ v

_∘ᶠ_ : FunctionalMatrix A → FunctionalMatrix A → FunctionalMatrix A
F ∘ᶠ G = ConstructFunctionalMatrix (apply_two_matrices F G)
```

Implementing matrix-matrix multiply on functional matrices

Previously, we were able to define matrix-matrix multiplication

$$M_1 * M_2$$

using function composition

```
apply_two_matrices : FunctionalMatrix A → FunctionalMatrix A
                   → List A → List A
apply_two_matrices F G v = F ·ᶠ G ·ᶠ v

_∘ᶠ_ : FunctionalMatrix A → FunctionalMatrix A → FunctionalMatrix A
F ∘ᶠ G = ConstructFunctionalMatrix (apply_two_matrices F G)
```

We can do the same with our new definition, by performing composition of the linear functions.

Matrix-matrix multiply toolbox: function extraction

We are going to need a few functions to implement matrix multiply. First we define a helper function to extract the forward function.

```
Mat-to-⊸ : ⦃ F : Field A ⦄ → Mat m × n → n ⊸ m
Mat-to-⊸ ⟦ f , t , p ⟧ = f
```

Matrix-matrix multiply toolbox: function extraction

We are going to need a few functions to implement matrix multiply. First we define a helper function to extract the forward function.

```
Mat-to-⊸ : ⦃ F : Field A ⦄ → Mat m × n → n ⊸ m
Mat-to-⊸ ⟦ f , t , p ⟧ = f
```

And then we can extract the transpose by generating the transpose matrix and running `Mat-to-⊸`.

```
_ᵀ : ⦃ F : Field A ⦄ → Mat m × n → Mat n × m
⟦ f , a , p ⟧ ᵀ = ⟦ a , f , (λ x y → sym (p y x)) ⟧
```

Matrix-matrix multiply toolbox: linear function composition

And we need to compose linear functions to compose the functions in matrices.

```
_∘ˡ_ : ⦃ F : Field A ⦄ → n ⊸ p → m ⊸ n → m ⊸ p
g ∘ˡ h = record {
    f = λ v → g ·ˡ (h ·ˡ v)
```

Matrix-matrix multiply toolbox: linear function composition

And we need to compose linear functions to compose the functions in matrices.

```
_∘ˡ_ : ⦃ F : Field A ⦄ → n ⊸ p → m ⊸ n → m ⊸ p
g ∘ˡ h = record {
    f = λ v → g ·ˡ (h ·ˡ v)
  ; f[u+v]≡f[u]+f[v] = Homework
  ; f[c*v]≡c*f[v] = Homework }
```

Defining matrix-matrix multiply

We can now define matrix-matrix multiply.

$$\begin{aligned} \text{forward}: &\; M_1 * M_2 \\ \text{transpose}: &\; (M_1 * M_2)^T = M_2^T * M_1^T \end{aligned}$$

Defining matrix-matrix multiply

We can now define matrix-matrix multiply.

$$\text{forward}: \; M_1 * M_2$$
$$\text{transpose}: \; (M_1 * M_2)^T = M_2^T * M_1^T$$

Which we can directly encode in Agda.

```
_*ᴹ_ : ⦃ F : Field A ⦄ → Mat m × n → Mat n × p → Mat m × p
M₂ *ᴹ M₁ =
  ⟦ (Mat-to-⊸ M₂) ∘ˡ (Mat-to-⊸ M₁)
  , (Mat-to-⊸ (M₁ ᵀ)) ∘ˡ (Mat-to-⊸ (M₂ ᵀ))
```

Defining matrix-matrix multiply

We can now define matrix-matrix multiply.

$$\text{forward}: \; M_1 * M_2$$
$$\text{transpose}: \; (M_1 * M_2)^T = M_2^T * M_1^T$$

Which we can directly encode in Agda.

```
_*ᴹ_ : ⦃ F : Field A ⦄ → Mat m × n → Mat n × p → Mat m × p
M₂ *ᴹ M₁ =
  ⟦ (Mat-to-⊸ M₂) ∘ˡ (Mat-to-⊸ M₁)
  , (Mat-to-⊸ (M₁ ᵀ)) ∘ˡ (Mat-to-⊸ (M₂ ᵀ))

  , MoreHomework!
  ⟧
```

What do we have so far with this encoding?

We have gained some nice benefits by moving to a proven type constructor.

- We can define a performant, functional version of matrix algebra.

What do we have so far with this encoding?

We have gained some nice benefits by moving to a proven type constructor.

- We can define a performant, functional version of matrix algebra.
- We can guarantee that our implementation is correct.

What do we have so far with this encoding?

We have gained some nice benefits by moving to a proven type constructor.

- We can define a performant, functional version of matrix algebra.
- We can guarantee that our implementation is correct.
- We can use equational reasoning to prove two implementations are equivalent.

What do we have so far with this encoding?

We have gained some nice benefits by moving to a proven type constructor.

- We can define a performant, functional version of matrix algebra.
- We can guarantee that our implementation is correct.
- We can use equational reasoning to prove two implementations are equivalent.

What do we have so far with this encoding?

We have gained some nice benefits by moving to a proven type constructor.

- We can define a performant, functional version of matrix algebra.
- We can guarantee that our implementation is correct.
- We can use equational reasoning to prove two implementations are equivalent.

But there is a cost. For one file in the library that implements this idea, out of 213 lines of code:

What do we have so far with this encoding?

We have gained some nice benefits by moving to a proven type constructor.

- We can define a performant, functional version of matrix algebra.
- We can guarantee that our implementation is correct.
- We can use equational reasoning to prove two implementations are equivalent.

But there is a cost. For one file in the library that implements this idea, out of 213 lines of code:

24 lines are function definitions (11.3% of the code). *Everything else is a proof, a type signature, or an import/control statement.*

Algorithms using Linear Algebra

What are the benefits of a functional approach to linear algebra?

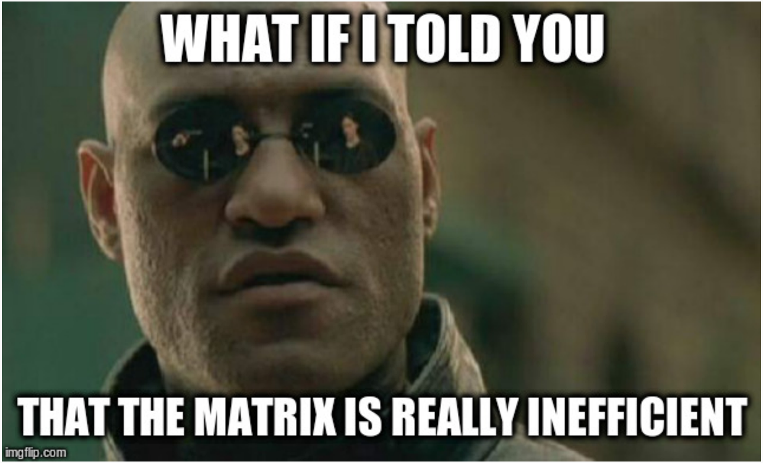
We gain a few benefits from using functions directly.

- Write out the model for a process in a more direct manner.

What are the benefits of a functional approach to linear algebra?

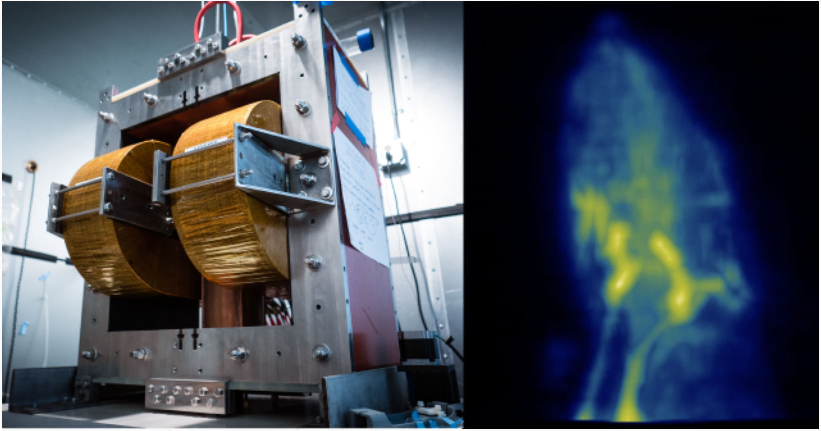
We gain a few benefits from using functions directly.

- Write out the model for a process in a more direct manner.
- Speed and time benefits.



Magnetic Particle Imaging reconstructs images using functional matrices

A model of how the device (left) generates signals from the sample (rat, right) is encoded as “matrix-free” functions in Python using PyOp, the python implementation of this idea.



Matrix-free methods enable significant time and space savings

We get a sizeable improvement in image reconstruction performance using a matrix-free method.

Matrix-free methods enable significant time and space savings

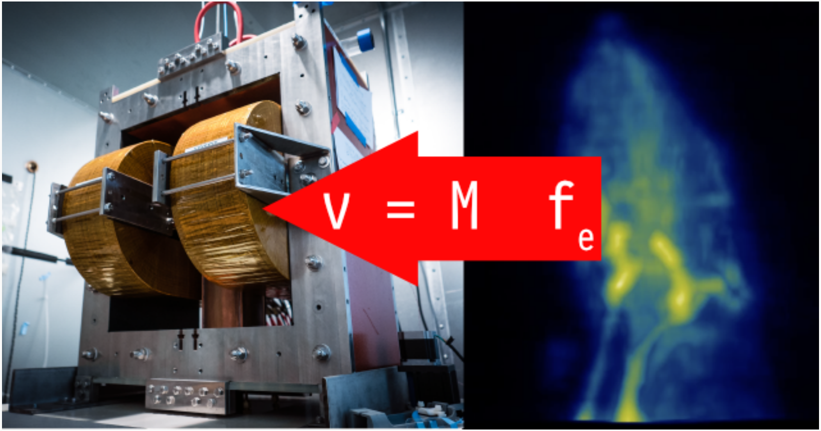
We get a sizeable improvement in image reconstruction performance using a matrix-free method.

Metric	Matrix	Matrix-Free	Improvement
Space	150 GB	bytes	10^9x
Time	60 min	2 min	$30x$
Use of functional concepts	No	Yes	Priceless

Magnetic Particle Imaging can be modeled using linear algebra

In MPI, we are attempting to detect where iron is within a sample.

- v : the voltages coming off of the device.
- f_e : the distribution of iron.
- M : a *function* that converts iron distributions into voltages.



Solving linear equations finds what input produces an observed result

We can write this process of converting iron distributions to voltages as

$$Mf_e = v$$

If we have the voltages v coming off the device, we want to find the iron distribution f_e that produced that signal.

Solving linear equations finds what input produces an observed result

We can write this process of converting iron distributions to voltages as

$$Mf_e = v$$

If we have the voltages v coming off the device, we want to find the iron distribution f_e that produced that signal.

To solve this problem, we want to compare how good our estimate of the input f_e is at producing the observed output v using the following function.

$$J(f_e) = f_e^T M^T M f_e - 2 f_e M^T v$$

Taking a **step** in the right direction

One simple way to find a better f_e than some initial guess is to update f_e *in the direction of steepest descent* ∇J.

$$f_{e,i+1} = f_{e,i} - \alpha \nabla J(f_{e,i})$$
$$f_{e,i+1} = f_{e,i} - \alpha(M^T(Mf_{e,i} - v))$$

Taking a **step** in the right direction

One simple way to find a better f_e than some initial guess is to update f_e *in the direction of steepest descent* ∇J.

$$f_{e,i+1} = f_{e,i} - \alpha \nabla J(f_{e,i})$$
$$f_{e,i+1} = f_{e,i} - \alpha (M^T (M f_{e,i} - v))$$

We can implement this in Agda as

```
step : ⦃ F : Field A ⦄
     → (α : A) → (M : Mat m × n)
     → (v : Vec A m) → (fₑ : Vec A n) → Vec A n
step α M v = λ fₑ → fₑ -ⱽ α ∘ⱽ (M ᵀ · (M · fₑ -ⱽ v))
```

Gradient descent is just running **step** a bunch of times

From there, we can find the value of f_e that best matches v by iterating.

```
gradient-descent : ⦃ F : Field A ⦄
                 → (n : ℕ)         -- Number of iterations to run
                 → (α : A)         -- Scale factor
                 → (M : Mat m × n) -- Model of system
                 → (v : Vec A m)   -- Data
                 → (fₑ : Vec A n)  -- Initial estimate
                 → List (Vec A n)  -- Results (farther is better)
gradient-descent n α M v fₑ = iterate n fₑ (step α M v)
-- iterate _ x f = [x, f x, f (f x), ... ]
```

We can define equivalent forms of a linear equation

We had defined our step function as

```
step α M v fₑ = fₑ -ⱽ α ∘ⱽ (M ᵀ · (M · fₑ -ⱽ v))
```

is there another way to write this function?

We had defined our step function as

```
step α M v fₑ = fₑ -ⱽ α ∘ⱽ (M ᵀ · (M · fₑ -ⱽ v))
```

is there another way to write this function?

yes!

```
step' : ⦃ F : Field A ⦄
      → (α : A) → (M : Mat m × n)
      → (v : Vec A m) → (fₑ : Vec A n) → Vec A n
step' α M v fₑ = fₑ -ⱽ α ∘ⱽ (M ᵀ · M · fₑ -ⱽ M ᵀ · v)
```

Proving the two steps are in lock step for better performance

We can prove **step** and **step'** are the same by demonstrating that the same inputs to **step** and **step'** lead to the same result.

```
proof :  ⦃ F : Field A ⦄ → (α : A)
      → (M : Mat m × n) → (v : Vec A m) → (fₑ : Vec A n)
      → step α M v fₑ ≡ step' α M v fₑ
proof α M v fₑ = begin
  fₑ -ⱽ α ∘ⱽ (M ᵀ · (M · fₑ -ⱽ v))
```

Proving the two **steps** are in lock step for better performance

We can prove **step** and **step'** are the same by demonstrating that the same inputs to **step** and **step'** lead to the same result.

```
proof : ⦃ F : Field A ⦄ → (α : A)
      → (M : Mat m × n) → (v : Vec A m) → (fₑ : Vec A n)
      → step α M v fₑ ≡ step' α M v fₑ
proof α M v fₑ = begin
  fₑ -ⱽ α ∘ⱽ (M ᵀ · (M · fₑ -ⱽ v))

  -- M-distr-–ⱽ : M (fₑ -ⱽ v) ≡ M fₑ -ⱽ M v
  ≡⟨ cong (λ z → fₑ -ⱽ α ∘ⱽ z) (M-distr-–ⱽ (M ᵀ) (M · fₑ) v) ⟩
  fₑ -ⱽ α ∘ⱽ (M ᵀ · M · fₑ -ⱽ M ᵀ · v) ∎
```

Proving the two **steps** are in lock step for better performance

We can prove **step** and **step'** are the same by demonstrating that the same inputs to **step** and **step'** lead to the same result.

```
proof : ⦃ F : Field A ⦄ → (α : A)
      → (M : Mat m × n) → (v : Vec A m) → (fₑ : Vec A n)
      → step α M v fₑ ≡ step' α M v fₑ
proof α M v fₑ = begin
  fₑ -ⱽ α ∘ⱽ (M ᵀ · (M · fₑ -ⱽ v))

  -- M-distr--ⱽ : M (fₑ -ⱽ v) ≡ M fₑ -ⱽ M v
  ≡⟨ cong (λ z → fₑ -ⱽ α ∘ⱽ z) (M-distr--ⱽ (M ᵀ) (M · fₑ) v) ⟩
  fₑ -ⱽ α ∘ⱽ (M ᵀ · M · fₑ -ⱽ M ᵀ · v) ∎
```

We can *rewrite/optimize our program while preserving correctness.*

Have we accomplished our goal?

Our original goal was

Correct by construction linear algebra; equivalent to $\mathbb{R}$ *matrices*

Have we accomplished our goal?

Our original goal was

Correct by construction linear algebra; equivalent to $\mathbb{R}$ *matrices*

we certainly achieved that!

Have we accomplished our goal?

Our original goal was

Correct by construction linear algebra; equivalent to $\mathbb{R}$ *matrices*

we certainly achieved that!

- Eliminated wrong size result bugs.

Have we accomplished our goal?

Our original goal was

Correct by construction linear algebra; equivalent to $\mathbb{R}$ *matrices*

we certainly achieved that!

- Eliminated wrong size result bugs.
- Eliminated non-linear function bugs.

Have we accomplished our goal?

Our original goal was

Correct by construction linear algebra; equivalent to $\mathbb{R}$ *matrices*

we certainly achieved that!

- Eliminated wrong size result bugs.
- Eliminated non-linear function bugs.
- Eliminated incorrect function pairing bugs.

Comparing the steps we went through

We could consider three levels of implementation that have different amounts of correctness built in.

- Regular functions (Python: `PyOp`/`PyLops`/many other libraries)

Comparing the steps we went through

We could consider three levels of implementation that have different amounts of correctness built in.

- Regular functions (Python: `PyOp`/`PyLops`/many other libraries)
- Size-typed functions (Haskell: `convex` library)

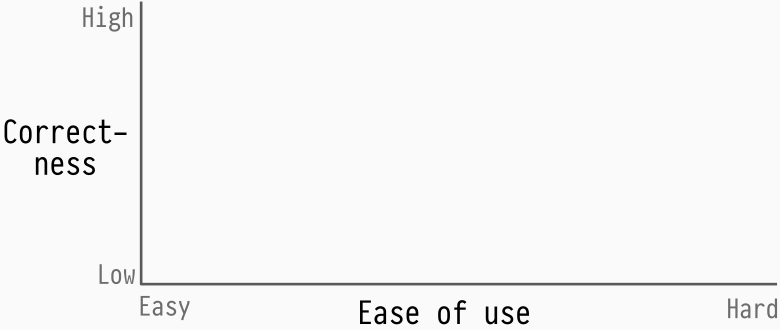
Comparing the steps we went through

We could consider three levels of implementation that have different amounts of correctness built in.

- Regular functions (Python: `PyOp`/`PyLops`/many other libraries)
- Size-typed functions (Haskell: `convex` library)
- Linear functions (Agda: `FLA` library)

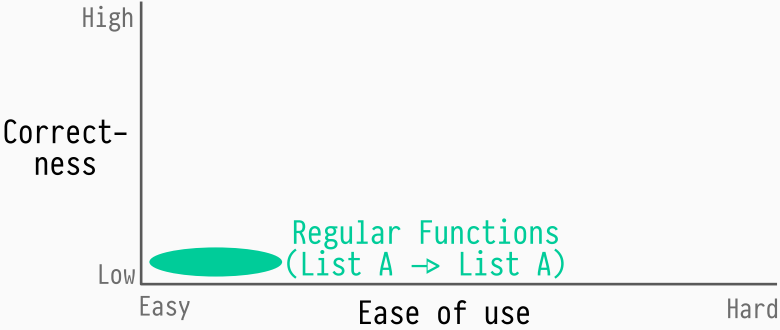
API comparison: how likely am I to use this?

Not every library is a blast to use. How do these three functional approaches stack up?



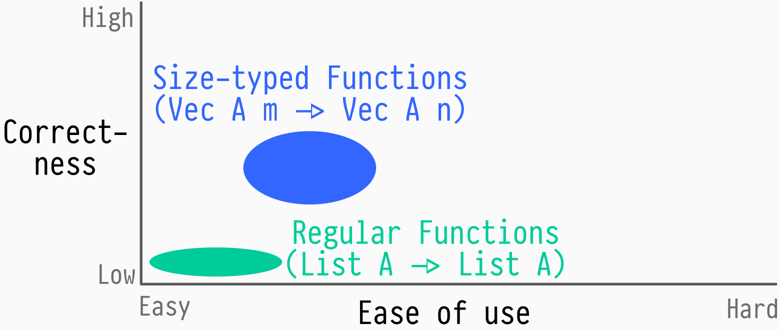
API comparison: how likely am I to use this?

Not every library is a blast to use. How do these three functional approaches stack up?



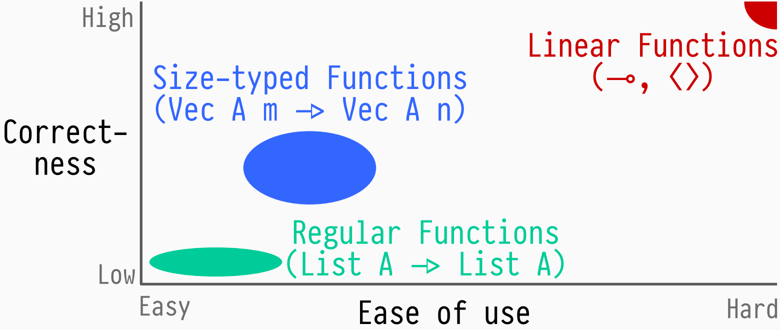
API comparison: how likely am I to use this?

Not every library is a blast to use. How do these three functional approaches stack up?



API comparison: how likely am I to use this?

Not every library is a blast to use. How do these three functional approaches stack up?



This presentation is a program!

This presentation is an Agda program! Instructions for how to load the presentation in Agda can be found at

github.com/ryanorendorff/functional-linear-algebra-talk

The full library that implements this style (without `TrustMe!`) can be found at

github.com/ryanorendorff/functional-linear-algebra

Questions?

Thanks for listening to my talk!

github.com/ryanorendorff/functional-linear-algebra-talk

Appendix

Instructions for how to run this presentation in Agda

If you have the Nix package manager installed, you can run

```
nix-shell
```

at the root of this presentation's repo and then launch emacs

```
emacs src/FormalizingLinearAlgebraAlgorithms.lagda.md
```

More information on the Agda emacs mode can be found https://agda.readthedocs.io/en/v2.6.1.1/tools/emacs-mode.html. If you use Spacemacs, the documentation for its Agda mode is https://www.spacemacs.org/layers/+lang/agda/README.html.