Denotational Design
from meanings to programs

Conal Elliott

LambdaJam 2015
Abstraction

The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.

- Edsger Dijkstra
Goals

- *Abstractions*: precise, elegant, reusable.
- *Implementations*: correct, efficient, maintainable.
- *Documentation*: clear, simple, accurate.
Conventional programming is precise only about how, not what.

*It is not only not right, it is not even wrong.*

- Wolfgang Pauli

*Everything is vague to a degree you do not realize till you have tried to make it precise.*

- Bertrand Russell

*What we wish, that we readily believe.*

- Demosthenes
Denotative programming

Peter Landin recommended “denotative” to replace ill-defined “functional” and “declarative”.

Properties:

- Nested expression structure.
- Each expression *denotes* something,
- depending only on denotations of subexpressions.

“...gives us a test for whether the notation is genuinely functional or merely masquerading.” (The Next 700 Programming Languages, 1966)
Denotational design

Design methodology for “genuinely functional” programming:

- Precise, simple, and compelling specification.
- Informs use and implementation without entangling them.
- Standard algebraic abstractions.
- Free of abstraction leaks.
- Laws for free.
- Principled construction of correct implementation.
Overview

- Broad outline:
  - Example, informally
  - *Pretty pictures*
  - Principles
  - More examples
  - Reflection

- Discussion throughout

- Try it on.
Example: image synthesis/manipulation

- How to start?

- What is success?
Functionality

- Import & export
- Spatial transformation:
  - Affine: translate, scale, rotate
  - Non-affine: swirls, lenses, inversions, ...
- Cropping
- Monochrome
- Overlay
- Blend
- Blur & sharpen
- Geometry, gradients, ..
type Image

over :: Image → Image → Image
transform :: Transform → Image → Image
crop :: Region → Image → Image
monochrome :: Color → Image
  -- shapes, gradients, etc.

fromBitmap :: Bitmap → Image
toBitmap :: Image → Bitmap
How to implement?

wrong first question
What to implement?

- What do these operations mean?

- More centrally: What do the *types* mean?
What is an image?

Specification goals:

- Adequate
- Simple
- Precise

Why these properties?
What is an image?

My answer: assignment of colors to 2D locations.

How to make precise?

\texttt{type Image}

Model:

\[ \mu :: \text{Image} \to (\text{Loc} \to \text{Color}) \]

What about regions?

\[ \mu :: \text{Region} \to (\text{Loc} \to \text{Bool}) \]
Specifying *Image* operations

\[
\begin{align*}
\mu (\text{over top bot}) & \equiv \ldots \\
\mu (\text{crop reg im}) & \equiv \ldots \\
\mu (\text{monochrome c}) & \equiv \ldots \\
\mu (\text{transform tr im}) & \equiv \ldots 
\end{align*}
\]
Specifying *Image* operations

\[
\mu (\text{over top bot}) \equiv \lambda p \rightarrow overC (\mu \text{top } p) (\mu \text{bot } p)
\]

\[
\mu (\text{crop reg im}) \equiv \lambda p \rightarrow \text{if } \mu \text{reg } p \text{ then } \mu \text{im } p \text{ else clear}
\]

\[
\mu (\text{monochrome } c) \equiv \lambda p \rightarrow c
\]

\[
\mu (\text{transform } tr \text{ im}) \equiv \text{-- coming up}
\]

\[
overC :: \text{Color} \rightarrow \text{Color} \rightarrow \text{Color}
\]

Note compositionality of \(\mu\).
Compositional semantics

Make more explicit:

\[ \mu \left( \text{over top bot} \right) \equiv \text{overS} \left( \mu \text{top} \right) \left( \mu \text{bot} \right) \]
\[ \mu \left( \text{crop reg im} \right) \equiv \text{cropS} \left( \mu \text{reg} \right) \left( \mu \text{im} \right) \]

\[ \text{overS} :: (\text{Loc} \to \text{Color}) \to (\text{Loc} \to \text{Color}) \to (\text{Loc} \to \text{Color}) \]
\[ \text{overS} \ f \ g = \lambda p \to \text{overC} \ (f \ p) \ (g \ p) \]

\[ \text{cropS} :: (\text{Loc} \to \text{Bool}) \to (\text{Loc} \to \text{Color}) \to (\text{Loc} \to \text{Color}) \]
\[ \text{cropS} \ f \ g = \lambda p \to \textbf{if} \ f \ p \ \textbf{then} \ g \ p \ \textbf{else} \ \textbf{clear} \]
Generalize and simplify

- What about transforming *regions*?
- Other pointwise combinations (lerp, threshold)?

Generalize:

```haskell
type Image a
type ImageC = Image Color
type Region = Image Bool
```

Now some operations become more general.
Generalize and simplify

\[
\begin{align*}
\text{transform} & : \text{Transform} \to \text{Image } a \to \text{Image } a \\
\text{cond} & : \text{Image } \text{Bool} \to \text{Image } a \to \text{Image } a \to \text{Image } a
\end{align*}
\]

\[
\begin{align*}
\text{lift}_0 & : a \to \text{Image } a \\
\text{lift}_1 & : (a \to b) \to (\text{Image } a \to \text{Image } b) \\
\text{lift}_2 & : (a \to b \to c) \to (\text{Image } a \to \text{Image } b \to \text{Image } c)
\end{align*}
\]

... 

Specializing,

\[
\begin{align*}
\text{monochrome} & = \text{lift}_0 \\
\text{over} & = \text{lift}_2 \text{ overC} \\
\text{crop } r \text{ im} & = \text{cond } r \text{ im emptyIm} \\
\text{cond} & = \text{lift}_3 \text{ ifThenElse}
\end{align*}
\]
Spatial transformation

\[ \mu :: \text{Transform} \rightarrow ?? \]

\[ \mu (\text{transform tr im}) \equiv ?? \]
Spatial transformation

\[ \mu :: \text{Transform} \rightarrow ?? \]

\[ \mu (\text{transform } tr \ im) \equiv \text{transformS} (\mu \ tr) (\mu \ im) \]

where

\[ \text{transformS} :: ?? \rightarrow (\text{Loc} \rightarrow \text{Color}) \rightarrow (\text{Loc} \rightarrow \text{Color}) \]
Spatial transformation

\[ \mu :: \text{Transform} \rightarrow (\text{Loc} \rightarrow \text{Loc}) \]

\[ \mu (\text{transform } tr \ im) \equiv \text{transform}_S (\mu tr) (\mu im) \]

where

\[ \text{transform}_S :: (\text{Loc} \rightarrow \text{Loc}) \rightarrow (\text{Loc} \rightarrow \text{Color}) \rightarrow (\text{Loc} \rightarrow \text{Color}) \]
\[ \text{transform}_S h f = \lambda p \rightarrow f (h p) \]

Subtle implications.

What is \textit{Loc}? My answer: continuous, infinite 2D space.

\textbf{type} \textit{Loc} = \mathbb{R}^2
Why continuous & infinite (vs discrete/finite) space?

Same benefits as for time (FRP):

- Transformation flexibility with simple & precise semantics.
- Modularity/reusability/composability:
  - Fewer assumptions, more uses (resolution-independence).
  - More info available for extraction.
- Integration and differentiation: natural, accurate, efficient.
- Quality/accuracy.
- Efficiency (adapative).
- Reconcile differing input sampling rates.

**Principle:** Approximations/prunings compose badly, so postpone.

See *Why Functional Programming Matters.*
Examples

Pan gallery
Using standard vocabulary

- We’ve created a domain-specific vocabulary.

- Can we reuse standard vocabularies instead?

- Why would we want to?
  - User knowledge.
  - Ecosystem support (multiplicative power).
  - Laws as sanity check.
  - Tao check.
  - Specification and laws for free, as we’ll see.

- In Haskell, standard type classes.
Monoid

Interface:

```haskell
class Monoid m where
  ε :: m           -- “mempty”
  (⊕) :: m → m → m -- “mappend”
```

Laws:

```haskell
a ⊕ ε    ≡ a
ε ⊕ b    ≡ b
a ⊕ (b ⊕ c) ≡ (a ⊕ b) ⊕ c
```


What monoids have we seen today?
**Image monoid**

```haskell
instance Monoid ImageC where
    ε   = lift₀ clear
    (⊕) = over
```

Is there a more general form on `Image a`?

```haskell
instance Monoid a ⇒ Monoid (Image a) where
    ε   = lift₀ ε
    (⊕) = lift₂ (⊕)
```

Do these instances satisfy the `Monoid` laws?
Functor

class Functor f where
  fmap :: (a → b) → (f a → f b)

For images?

  instance Functor Image where
    fmap = lift1

Laws?
Applicative

```haskell
class Functor f ⇒ Applicative f where
  pure :: a → f a
  (⊛) :: f (a → b) → f a → f b
```

For images?

```haskell
instance Applicative Image where
  pure = lift₀
  (⊛) = lift₂ ($)
```

From Applicative, where (⊛) = fmap:

```haskell
liftA₂ f p q = f <$> p <*> q
liftA₃ f p q r = f <$> p <*> q <*> r
  -- etc
```

Laws?

Conal Elliott  Denotational Design  LambdaJam 2015 29 / 58
Instance semantics

**Monoid:**

\[ \mu \varepsilon \equiv \lambda p \rightarrow \varepsilon \]

\[ \mu (top \oplus bot) \equiv \lambda p \rightarrow \mu top \ p \oplus \mu bot \ p \]

**Functor:**

\[ \mu (fmap f \ im) \equiv \lambda p \rightarrow f (\mu im \ p) \]

\[ \equiv f \circ \mu im \]

**Applicative:**

\[ \mu (\text{pure } a) \equiv \lambda p \rightarrow a \]

\[ \mu (\text{imf } \star imx) \equiv \lambda p \rightarrow (\mu imf \ p) \ (\mu imx \ p) \]
**Monad and Comonad**

```haskell
class Monad f where
    return :: a -> f a
    join    :: f (f a) -> f a

class Functor f => Comonad f where
    coreturn :: f a -> a
    cojoin  :: f a -> f (f a)
```

*Comonad* gives us neighborhood operations.
Monoid specification, revisited

Image monoid specification:

\[
\begin{align*}
\mu \varepsilon & \equiv \lambda p \to \varepsilon \\
\mu (\text{top} \oplus \text{bot}) & \equiv \lambda p \to \mu \text{top } p \oplus \mu \text{bot } p
\end{align*}
\]

Instance for the semantic model:

instance Monoid \( m \) ⇒ Monoid \((z \to m)\) where

\[
\begin{align*}
\varepsilon & = \lambda z \to \varepsilon \\
f \oplus g & = \lambda z \to f \ z \oplus g \ z
\end{align*}
\]

Refactoring,

\[
\begin{align*}
\mu \varepsilon & \equiv \varepsilon \\
\mu (\text{top} \oplus \text{bot}) & \equiv \mu \text{top} \oplus \mu \text{bot}
\end{align*}
\]

So \( \mu \) distributes over monoid operations, i.e., a monoid homomorphism.
Functor specification, revisited

Functor specification:

\[ \mu (fmap f \ im) \equiv f \circ \mu \ im \]

Instance for the semantic model:

\textbf{instance} Functor \((\rightarrow) \ u) \ \textbf{where}

\[ fmap f \ h = f \circ h \]

Refactoring,

\[ \mu (fmap f \ im) \equiv fmap f (\mu \ im) \]

So \(\mu\) is a \textit{functor} homomorphism.
Applicative specification, revisited

Applicative specification:

$$
\mu (\text{pure } a) \equiv \lambda p \to a \\
\mu (\text{imf } \star \text{imx}) \equiv \lambda p \to (\mu \text{imf } p)(\mu \text{imx } p)
$$

Instance for the semantic model:

instance Applicative ((→) u) where

pure a = λu → a

fs ◦ xs = λu → (fs u)(xs u)

Refactoring,

$$
\mu (\text{pure } a) \equiv \text{pure } a \\
\mu (\text{imf } \star \text{imx}) \equiv \mu \text{imf } \star \mu \text{imx}
$$

So $\mu$ is an applicative homomorphism.
Specifications for free

Semantic type class morphism (TCM) principle:

The instance’s meaning follows the meaning’s instance.

That is, the type acts like its meaning.

Every TCM failure is an abstraction leak.

Strong design principle.

Class laws necessarily hold, as we’ll see.
Laws for free

\[\mu \epsilon \equiv \epsilon\]
\[\mu (a \oplus b) \equiv \mu a \oplus \mu b\]

\[a \oplus \epsilon \equiv a\]
\[\epsilon \oplus b \equiv b\]
\[a \oplus (b \oplus c) \equiv (a \oplus b) \oplus c\]

where equality is semantic. Proofs:

<table>
<thead>
<tr>
<th>[\mu (a \oplus \epsilon)]</th>
<th>[\mu (\epsilon \oplus b)]</th>
<th>[\mu (a \oplus (b \oplus c))]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[\equiv \mu a \oplus \mu \epsilon]</td>
<td>[\equiv \mu \epsilon \oplus \mu b]</td>
<td>[\equiv \mu a \oplus (\mu b \oplus \mu c)]</td>
</tr>
<tr>
<td>[\equiv \mu a \oplus \epsilon]</td>
<td>[\equiv \epsilon \oplus \mu b]</td>
<td>[\equiv (\mu a \oplus \mu b) \oplus \mu c]</td>
</tr>
<tr>
<td>[\equiv \mu a]</td>
<td>[\equiv \mu b]</td>
<td>[\equiv \mu ((a \oplus b) \oplus c)]</td>
</tr>
</tbody>
</table>

Works for other classes as well.
Example: functional reactive programming

See previous talks:

- *The essence and origins of FRP*
- *A more elegant specification for FRP*
Example: uniform pairs

Type:

\[
\textbf{data } \textit{Pair} \ a = \ a :\# \ a
\]

API: \textit{Monoid}, \textit{Functor}, \textit{Applicative}, \textit{Monad}, \textit{Foldable}, \textit{Traversable}.

Specification follows from simple & precise denotation.
Uniform pairs — denotation

*Pair* is an *indexable* container. What’s the index type?

\[
\text{type } P \, a = \text{Bool} \rightarrow a
\]

\[
\mu :: Pair \, a \rightarrow P \, a
\]

\[
\mu \, (u :\# v) \, \text{False} = u
\]

\[
\mu \, (u :\# v) \, \text{True} = v
\]

API specification? Homomorphisms, as usual!
Uniform pairs — monoid

Monoid homomorphism:

\[ \mu \varepsilon \equiv \varepsilon \]
\[ \mu (u \oplus v) \equiv \mu \ u \oplus \mu \ v \]

In this case,

\textbf{instance} Monoid \( m \Rightarrow \text{Monoid} \ (z \rightarrow m) \) \textbf{where}

\[ \varepsilon = \lambda z \rightarrow \varepsilon \]
\[ f \oplus g = \lambda z \rightarrow f \ z \oplus g \ z \]

so

\[ \mu \varepsilon \equiv \lambda z \rightarrow \varepsilon \]
\[ \mu (u \oplus v) \equiv \lambda z \rightarrow \mu \ u \ z \oplus \mu \ v \ z \]

Implementation: solve for \( \varepsilon \) and \( \oplus \) on the left. Hint: find \( \mu^{-1} \).
Uniform pairs — other classes

Exercise: apply the same principle for

- Functor
- Applicative
- Monad
- Foldable
- Traversable
Example: streams

```haskell
data Stream a = Cons a (Stream a)
```

API: same classes as with Pair.

Denotation? Hint: Stream is also an indexable type.

```haskell
data S a = Nat → a

data Nat = Zero | Succ Nat
```

Interpret Stream as S:

```text
μ :: Stream a → S a
μ (Cons a _) Zero = a
μ (Cons _ as) (Succ n) = μ as n
```
Memo tries

Generalizes *Pair* and *Stream*:

\[
\text{type } a \rightarrow b \\
\mu :: (a \rightarrow b) \rightarrow (a \rightarrow b)
\]

API: classes as above, plus *Category*.

Exploit inverses to calculate instances, e.g.,

\[
\begin{align*}
\mu \ id & \equiv id \\
\leftarrow id & \equiv \mu^{-1} id
\end{align*}
\]

\[
\begin{align*}
\mu \ (g \circ f) & \equiv \mu \ g \circ \mu \ f \\
\leftarrow g \circ f & \equiv \mu^{-1} \ (\mu \ g \circ \mu \ f)
\end{align*}
\]

Then simplify/optimize.
Example: lists with a bonus

```
data ListX a b = Done b | Cons a (ListX a b)
```

Denotation:

```
\mu :: ListX a b \rightarrow ([a], b) \\
\mu (Done b) = ([], b) \\
\mu (Cons a asb) = (a : as, b) where (as, b) = \mu asb
```

Exercise: instances, including

```
instance Monad (ListX a) where ...
```

Then generalize from lists to arbitrary monoid.
Example: linear transformations

Assignment:

- Represent linear transformations
- Scalar, non-scalar domain & range, identity and composition

Plan:

- Interface
- Denotation
- Representation
- Calculation (implementation)
Interface and denotation

**Interface:**

- **type** $(\cdot \to \cdot) :: \cdot \to \cdot \to \cdot$

- $scale :: Num s \Rightarrow (s :\to s)$

- $\hat{id} :: a :\to a$

- $(\hat{\circ}) :: (b :\to c) \to (a :\to b) \to (a :\to c)$

...  

**Model:**

- **type** $a \to b$ -- Linear subset of $a \to b$

- $\mu :: (a :\to b) \to (a \to b)$

- $\mu (scale s) \equiv \lambda x \to s \times x$

- $\mu \hat{id} \equiv id$

- $\mu (g \circ \hat{f}) \equiv \mu g \circ \mu f$

...
Representation

Start with 1D. Recall partial specification:

\[ \mu (\text{scale } s) \equiv \lambda x \rightarrow s \times x \]

Try a direct data type representation:

\begin{verbatim}
data (\vdash \circ) :: \ast \rightarrow \ast \rightarrow \ast \ where
  Scale :: \text{Num } s \Rightarrow \text{s }\rightarrow (s :\vdash \circ \ s) \ -- \ ...
\mu :: (a :\vdash \circ \ b) \rightarrow (a \rightarrow b)
\mu \ (\text{Scale } s) = \lambda x \rightarrow s \times x
\end{verbatim}

Spec trivially satisfied by \( scale = \text{Scale}. \)

Others are more interesting.
Calculate an implementation

Specification:

\[
\mu \widehat{id} \equiv id \\
\mu (g \circ f) \equiv \mu g \circ \mu f
\]

Calculation:

\[
\begin{align*}
  id & \equiv \lambda x \rightarrow x \\
  & \equiv \lambda x \rightarrow 1 \times x \\
  & \equiv \mu (\text{Scale 1}) \\
\end{align*}
\]

\[
\begin{align*}
  \mu (\text{Scale } s) \circ \mu (\text{Scale } s') & \equiv (\lambda x \rightarrow s \times x) \circ (\lambda x' \rightarrow s' \times x') \\
  & \equiv \lambda x' \rightarrow s \times (s' \times x') \\
  & \equiv \lambda x' \rightarrow ((s \times s') \times x') \\
  & \equiv \mu (\text{Scale }(s \times s'))
\end{align*}
\]

Sufficient definitions:

\[
\begin{align*}
  \widehat{id} = \text{Scale 1} \\
  \text{Scale } s \circ \text{Scale } s' = \text{Scale } (s \times s')
\end{align*}
\]
Algebraic abstraction

In general,

- Replace ad hoc vocabulary with a standard abstraction.
- Recast semantics as homomorphism.
- Note that laws hold.

What standard abstraction to use for (\(\Rightarrow\))?
Category

Interface:

```haskell
class Category k where
  id :: k a a
  (○) :: k b c → k a b → k a c
```

Laws:

\[
\begin{align*}
  id \circ f & \equiv f \\
  g \circ id & \equiv g \\
  (h \circ g) \circ f & \equiv h \circ (g \circ f)
\end{align*}
\]
Linear transformation category

Linear map semantics:

$$\mu :: (a \to b) \to (a \to b)$$
$$\mu (\text{Scale } s) = \lambda x \to s \times x$$

Specification as homomorphism (no abstraction leak):

$$\mu \text{id} \equiv \text{id}$$
$$\mu (g \circ f) \equiv \mu g \circ \mu f$$

Correct-by-construction implementation:

```haskell
instance Category (\to) where
  id = Scale 1
  Scale s \circ Scale s' = Scale (s \times s')
```
Laws for free

\[
\begin{align*}
\mu \text{id} & \equiv \text{id} \\
\mu (g \circ f) & \equiv \mu g \circ \mu f
\end{align*}
\]

\[
\begin{align*}
\text{id} \circ f & \equiv f \\
g \circ \text{id} & \equiv g \\
(h \circ g) \circ f & \equiv h \circ (g \circ f)
\end{align*}
\]

where equality is semantic. Proofs:

\[
\begin{align*}
\mu (\text{id} \circ f) & \equiv \mu \text{id} \circ \mu f \\
& \equiv \text{id} \circ \mu f \\
& \equiv \mu f
\end{align*}
\]

\[
\begin{align*}
\mu (g \circ \text{id}) & \equiv \mu g \circ \mu \text{id} \\
& \equiv \mu g \circ \text{id} \\
& \equiv \mu g
\end{align*}
\]

\[
\begin{align*}
\mu ((h \circ g) \circ f) & \equiv (\mu h \circ \mu g) \circ \mu f \\
& \equiv \mu h \circ (\mu g \circ \mu f) \\
& \equiv \mu (h \circ (g \circ f))
\end{align*}
\]

Works for other classes as well.
Higher dimensions

Interface:

\[(\triangle) :: (a :\rightarrow c) \rightarrow (a :\rightarrow d) \rightarrow (a :\rightarrow c \times d)\]
\[(\nabla) :: (a :\rightarrow c) \rightarrow (b :\rightarrow c) \rightarrow (a \times b :\rightarrow c)\]

Semantics:

\[\mu (f \triangle g) \equiv \lambda a \rightarrow (f \ a, g \ a)\]
\[\mu (f \nabla g) \equiv \lambda (a, b) \rightarrow f \ a + g \ b\]
Products and coproducts

```haskell
class Category k ⇒ ProductCat k where
  type a ×ₖ b
  exl :: k (a ×ₖ b) a
  expr :: k (a ×ₖ b) b
  (△) :: k a c → k a d → k a (c ×ₖ d)

class Category k ⇒ CoproductCat k where
  type a +ₖ b
  inl :: k a (a +ₖ b)
  inr :: k b (a +ₖ b)
  (▽) :: k a c → k b c → k (a +ₖ b) c
```

Similar to Arrow and ArrowChoice classes.
Semantic morphisms

\[
\begin{align*}
\mu \text{ exl} & \equiv \text{ exl} \\
\mu \text{ exr} & \equiv \text{ exr} \\
\mu (f \triangle g) & \equiv \mu f \triangle \mu g
\end{align*}
\]

For \( a \rightarrow b \),

\[
\begin{align*}
\text{type } a \times (\_ \rightarrow) b & = a \times b \\
\text{exl } (a, b) & = a \\
\text{exr } (a, b) & = b \\
f \triangle g & = \lambda a \rightarrow (f \ a, g \ a)
\end{align*}
\]

\[
\begin{align*}
\text{type } a + (\_ \rightarrow) b & = a \times b \\
\text{inl } a & = (a, 0) \\
\text{inr } b & = (0, b) \\
f \nabla g & = \lambda (a, b) \rightarrow f \ a + g \ b
\end{align*}
\]

For calculation, see blog post *Reimagining matrices.*
**Full representation and denotation**

\[ \text{data} (\rightarrow) :: * \rightarrow * \rightarrow * \  \text{where} \]

\[ \text{Scale} :: \text{Num} s \Rightarrow s \rightarrow (s \rightarrow s) \]

\[ (\wedge) :: (a \rightarrow c) \rightarrow (a \rightarrow d) \rightarrow (a \rightarrow c \times d) \]

\[ (\lor) :: (a \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow (a \times b \rightarrow c) \]

\[ \mu :: (a \rightarrow b) \rightarrow (a \rightarrow b) \]

\[ \mu (\text{Scale } s) = \lambda x \rightarrow s \times x \]

\[ \mu (f : \wedge g) = \lambda a \rightarrow (f a, g a) \]

\[ \mu (f : \lor g) = \lambda (a, b) \rightarrow f a + g b \]
Design methodology for typed, purely functional programming:

- Precise, simple, and compelling specification.
- Informs *use* and *implementation* without entangling.
- Standard algebraic abstractions.
- Free of abstraction leaks.
- Laws for free.
- Principled construction of correct implementation.
References

- *Denotational design with type class morphisms*
- *Push-pull functional reactive programming*
- Functional images (Pan) page with pictures & papers.
- Posts on type class morphisms
- *Reimagining matrices*
- This workshop