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.

Not even wrong

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, ...

API first pass

type *Image*

$$
\begin{aligned}
&\mathit{over} &&:: \mathit{Image} \rightarrow \mathit{Image} \rightarrow \mathit{Image} \\
&\mathit{transform} &&:: \mathit{Transform} \rightarrow \mathit{Image} \rightarrow \mathit{Image} \\
&\mathit{crop} &&:: \mathit{Region} \rightarrow \mathit{Image} \rightarrow \mathit{Image} \\
&\mathit{monochrome} &&:: \mathit{Color} \rightarrow \mathit{Image}
\end{aligned}
$$

  -- shapes, gradients, etc.

$$
\begin{aligned}
&\mathit{fromBitmap} &&:: \mathit{Bitmap} \rightarrow \mathit{Image} \\
&\mathit{toBitmap} &&:: \mathit{Image} \rightarrow \mathit{Bitmap}
\end{aligned}
$$

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?

type *Image*

Model:

$$\mu :: \mathit{Image} \to (\mathit{Loc} \to \mathit{Color})$$

What about regions?

$$\mu :: \mathit{Region} \to (\mathit{Loc} \to \mathit{Bool})$$

Specifying *Image* operations

$$
\begin{aligned}
\mu\ (over\ top\ bot) &\equiv \ldots \\
\mu\ (crop\ reg\ im) &\equiv \ldots \\
\mu\ (monochrome\ c) &\equiv \ldots \\
\mu\ (transform\ tr\ im) &\equiv \ldots
\end{aligned}
$$

Specifying *Image* operations

$$
\begin{aligned}
\mu\ (over\ top\ bot) &\equiv \lambda p \rightarrow overC\ (\mu\ top\ p)\ (\mu\ bot\ p) \\
\mu\ (crop\ reg\ im) &\equiv \lambda p \rightarrow \mathbf{if}\ \mu\ reg\ p\ \mathbf{then}\ \mu\ im\ p\ \mathbf{else}\ clear \\
\mu\ (monochrome\ c) &\equiv \lambda p \rightarrow c \\
\mu\ (transform\ tr\ im) &\equiv \quad \text{-- coming up}
\end{aligned}
$$

$$overC :: Color \rightarrow Color \rightarrow Color$$

Note compositionality of μ.

Compositional semantics

Make more explicit:

$$\mu\ (over\ top\ bot) \equiv overS\ (\mu\ top)\ (\mu\ bot)$$
$$\mu\ (crop\ reg\ im)\ \equiv cropS\ (\mu\ reg)\ (\mu\ im)$$

$$overS :: (Loc \to Color) \to (Loc \to Color) \to (Loc \to Color)$$
$$overS\ f\ g = \lambda p \to overC\ (f\ p)\ (g\ p)$$

$$cropS :: (Loc \to Bool) \to (Loc \to Color) \to (Loc \to Color)$$
$$cropS\ f\ g = \lambda p \to \textbf{if}\ f\ p\ \textbf{then}\ g\ p\ \textbf{else}\ clear$$

Generalize and simplify

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

Generalize:

type *Image a*
type *ImageC* = *Image Color*
type *Region* = *Image Bool*

Now some operations become more general.

Generalize and simplify

$$\begin{aligned}
\mathit{transform} &:: \mathit{Transform} \rightarrow \mathit{Image}\ a \rightarrow \mathit{Image}\ a \\
\mathit{cond} &:: \mathit{Image}\ \mathit{Bool} \rightarrow \mathit{Image}\ a \rightarrow \mathit{Image}\ a \rightarrow \mathit{Image}\ a
\end{aligned}$$

$$\begin{aligned}
&\mathit{lift}_0 :: a \rightarrow \mathit{Image}\ a \\
&\mathit{lift}_1 :: (a \rightarrow b) \rightarrow (\mathit{Image}\ a \rightarrow \mathit{Image}\ b) \\
&\mathit{lift}_2 :: (a \rightarrow b \rightarrow c) \rightarrow (\mathit{Image}\ a \rightarrow \mathit{Image}\ b \rightarrow \mathit{Image}\ c) \\
&\ \ldots
\end{aligned}$$

Specializing,

$$\begin{aligned}
\mathit{monochrome} &= \mathit{lift}_0 \\
\mathit{over} &= \mathit{lift}_2\ \mathit{overC} \\
\mathit{crop}\ r\ \mathit{im} &= \mathit{cond}\ r\ \mathit{im}\ \mathit{emptyIm} \\
\mathit{cond} &= \mathit{lift}_3\ \mathit{ifThenElse}
\end{aligned}$$

Spatial transformation

$$\mu :: \mathit{Transform} \rightarrow ??$$

$$\mu\ (\mathit{transform}\ \mathit{tr}\ \mathit{im}) \equiv ??$$

Spatial transformation

$$\mu :: \mathit{Transform} \rightarrow ??$$

$$\mu\ (\mathit{transform}\ \mathit{tr}\ \mathit{im}) \equiv \mathit{transformS}\ (\mu\ \mathit{tr})\ (\mu\ \mathit{im})$$

where

$$\mathit{transformS} :: ?? \rightarrow (\mathit{Loc} \rightarrow \mathit{Color}) \rightarrow (\mathit{Loc} \rightarrow \mathit{Color})$$

Spatial transformation

$$\mu :: \mathit{Transform} \rightarrow (\mathit{Loc} \rightarrow \mathit{Loc})$$

$$\mu\ (\mathit{transform}\ \mathit{tr}\ \mathit{im}) \equiv \mathit{transformS}\ (\mu\ \mathit{tr})\ (\mu\ \mathit{im})$$

where

$$\begin{aligned}&\mathit{transformS} :: (\mathit{Loc} \rightarrow \mathit{Loc}) \rightarrow (\mathit{Loc} \rightarrow \mathit{Color}) \rightarrow (\mathit{Loc} \rightarrow \mathit{Color})\\ &\mathit{transformS}\ h\ f = \lambda p \rightarrow f\ (h\ p)\end{aligned}$$

Subtle implications.

What is *Loc*? My answer: continuous, infinite 2D space.

$$\mathbf{type}\ \mathit{Loc} = \mathbb{R}^2$$

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:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monoid}\ m\ \textbf{where} \\
\quad \varepsilon \quad :: m \qquad\qquad\qquad \text{-- “mempty”} \\
\quad (\oplus) :: m \to m \to m \quad \text{-- “mappend”}
\end{array}
$$

Laws:

$$
\begin{array}{lcl}
a \oplus \varepsilon & \equiv & a \\
\varepsilon \oplus b & \equiv & b \\
a \oplus (b \oplus c) & \equiv & (a \oplus b) \oplus c
\end{array}
$$

Why do laws *matter*? Compositional (modular) reasoning.

What monoids have we seen today?

Image monoid

$$\begin{array}{l}\textbf{instance}\ \mathit{Monoid}\ \mathit{ImageC}\ \textbf{where} \\ \quad \varepsilon \quad = \mathit{lift}_0\ \mathit{clear} \\ \quad (\oplus) = \mathit{over}\end{array}$$

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

$$\begin{array}{l}\textbf{instance}\ \mathit{Monoid}\ a \Rightarrow \mathit{Monoid}\ (\mathit{Image}\ a)\ \textbf{where} \\ \quad \varepsilon \quad = \mathit{lift}_0\ \varepsilon \\ \quad (\oplus) = \mathit{lift}_2\ (\oplus)\end{array}$$

Do these instances satisfy the *Monoid* laws?

Functor

$$\textbf{class}\ \mathit{Functor}\ f\ \textbf{where}$$
$$\quad \mathit{fmap} :: (a \to b) \to (f\ a \to f\ b)$$

For images?

$$\textbf{instance}\ \mathit{Functor}\ \mathit{Image}\ \textbf{where}$$
$$\quad \mathit{fmap} = \mathit{lift}_1$$

Laws?

Applicative

$$
\begin{aligned}
&\textbf{class } \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f\ \textbf{where}\\
&\quad \mathit{pure}\ \ :: a \to f\ a\\
&\quad (\circledast) :: f\ (a \to b) \to f\ a \to f\ b
\end{aligned}
$$

For images?

$$
\begin{aligned}
&\textbf{instance } \mathit{Applicative}\ \mathit{Image}\ \textbf{where}\\
&\quad \mathit{pure}\ \ = \mathit{lift}_0\\
&\quad (\circledast) = \mathit{lift}_2\ (\$)
\end{aligned}
$$

From *Applicative*, where $(<\$>) = \mathit{fmap}$:

$$
\begin{aligned}
&\mathit{liftA}_2\ f\ p\ q\ \ \ \ = f <\$> p \circledast q\\
&\mathit{liftA}_3\ f\ p\ q\ r = f <\$> p \circledast q \circledast r\\
&\quad \text{-- etc}
\end{aligned}
$$

Laws?

Instance semantics

Monoid:

$$
\begin{aligned}
\mu\ \varepsilon &\equiv \lambda p \rightarrow \varepsilon \\
\mu\ (top \oplus bot) &\equiv \lambda p \rightarrow \mu\ top\ p \oplus \mu\ bot\ p
\end{aligned}
$$

Functor:

$$
\begin{aligned}
\mu\ (fmap\ f\ \ im) &\equiv \lambda p \rightarrow f\ (\mu\ im\ p) \\
&\equiv f \circ \mu\ im
\end{aligned}
$$

Applicative:

$$
\begin{aligned}
\mu\ (pure\ \ a) &\equiv \lambda p \rightarrow a \\
\mu\ (imf \circledast imx) &\equiv \lambda p \rightarrow (\mu\ imf\ \ p)\ (\mu\ imx\ \ p)
\end{aligned}
$$

Monad and *Comonad*

$$
\begin{aligned}
&\textbf{class } \mathit{Monad}\ f\ \textbf{where}\\
&\quad \mathit{return} :: a \rightarrow f\ a\\
&\quad \mathit{join} \quad :: f\ (f\ a) \rightarrow f\ a
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{class } \mathit{Functor}\ f \Rightarrow \mathit{Comonad}\ f\ \textbf{where}\\
&\quad \mathit{coreturn} :: f\ a \rightarrow a\\
&\quad \mathit{cojoin} \quad :: f\ a \rightarrow f\ (f\ a)
\end{aligned}
$$

Comonad gives us neighborhood operations.

Monoid specification, revisited

Image monoid specification:

$$\begin{aligned} \mu\;\varepsilon &\equiv \lambda p \to \varepsilon \\ \mu\;(top \oplus bot) &\equiv \lambda p \to \mu\;top\;p \oplus \mu\;bot\;p \end{aligned}$$

Instance for the semantic model:

$$\begin{aligned} &\textbf{instance}\; Monoid\; m \Rightarrow Monoid\; (z \to m)\; \textbf{where} \\ &\quad \varepsilon \quad\;\; = \lambda z \to \varepsilon \\ &\quad f \oplus g = \lambda z \to f\;z \oplus g\;z \end{aligned}$$

Refactoring,

$$\begin{aligned} \mu\;\varepsilon &\equiv \varepsilon \\ \mu\;(top \oplus bot) &\equiv \mu\;top \oplus \mu\;bot \end{aligned}$$

So μ *distributes* over monoid operations, i.e., a monoid homomorphism.

Functor specification, revisited

Functor specification:

$$\mu\ (\mathit{fmap}\ f\ \mathit{im}) \equiv f \circ \mu\ \mathit{im}$$

Instance for the semantic model:

$$\textbf{instance}\ \mathit{Functor}\ ((\rightarrow)\ u)\ \textbf{where}$$
$$\quad \mathit{fmap}\ f\ h = f \circ h$$

Refactoring,

$$\mu\ (\mathit{fmap}\ f\ \mathit{im}) \equiv \mathit{fmap}\ f\ (\mu\ \mathit{im})$$

So μ is a *functor* homomorphism.

Applicative specification, revisited

Applicative specification:

$$\mu\ (pure\ a) \equiv \lambda p \to a$$
$$\mu\ (imf \circledast imx) \equiv \lambda p \to (\mu\ imf\ p)\ (\mu\ imx\ p)$$

Instance for the semantic model:

$$\textbf{instance}\ \mathit{Applicative}\ ((\to)\ u)\ \textbf{where}$$
$$\quad pure\ a = \lambda u \to a$$
$$\quad fs \circledast xs = \lambda u \to (fs\ u)\ (xs\ u)$$

Refactoring,

$$\mu\ (pure\ a) \equiv pure\ a$$
$$\mu\ (imf \circledast imx) \equiv \mu\ imf \circledast \mu\ imx$$

So μ 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

$$\begin{aligned} \mu\ \varepsilon &\equiv \varepsilon \\ \mu\ (a \oplus b) &\equiv \mu\ a \oplus \mu\ b \end{aligned} \quad \Rightarrow \quad \begin{aligned} a \oplus \varepsilon &\equiv a \\ \varepsilon \oplus b &\equiv b \\ a \oplus (b \oplus c) &\equiv (a \oplus b) \oplus c \end{aligned}$$

where equality is *semantic*. Proofs:

$$\begin{aligned} &\mu\ (a \oplus \varepsilon) \\ \equiv\ &\mu\ a \oplus \mu\ \varepsilon \\ \equiv\ &\mu\ a \oplus \varepsilon \\ \equiv\ &\mu\ a \end{aligned}$$

$$\begin{aligned} &\mu\ (\varepsilon \oplus b) \\ \equiv\ &\mu\ \varepsilon \oplus \mu\ b \\ \equiv\ &\varepsilon \oplus \mu\ b \\ \equiv\ &\mu\ b \end{aligned}$$

$$\begin{aligned} &\mu\ (a \oplus (b \oplus c)) \\ \equiv\ &\mu\ a \oplus (\mu\ b \oplus \mu\ c) \\ \equiv\ &(\mu\ a \oplus \mu\ b) \oplus \mu\ c \\ \equiv\ &\mu\ ((a \oplus b) \oplus c) \end{aligned}$$

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:

$$\textbf{data}\ \mathit{Pair}\ a = a :\!\#\ a$$

API: *Monoid*, *Functor*, *Applicative*, *Monad*, *Foldable*, *Traversable*.

Specification follows from simple & precise denotation.

Uniform pairs — denotation

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

$$\textbf{type}\ P\ a = Bool \to a$$

$$\mu :: Pair\ a \to P\ a$$
$$\mu\ (u :\#\ v)\ False = u$$
$$\mu\ (u :\#\ v)\ True = v$$

API specification? Homomorphisms, as usual!

Uniform pairs — monoid

Monoid homomorphism:

$$\begin{aligned} \mu\ \varepsilon &\equiv \varepsilon \\ \mu\ (u \oplus v) &\equiv \mu\ u \oplus \mu\ v \end{aligned}$$

In this case,

$$\begin{aligned} &\textbf{instance}\ \mathit{Monoid}\ m \Rightarrow \mathit{Monoid}\ (z \to m)\ \textbf{where} \\ &\quad \varepsilon \qquad = \lambda z \to \varepsilon \\ &\quad f \oplus g = \lambda z \to f\ z \oplus g\ z \end{aligned}$$

so

$$\begin{aligned} \mu\ \varepsilon &\equiv \lambda z \to \varepsilon \\ \mu\ (u \oplus v) &\equiv \lambda z \to \mu\ u\ z \oplus \mu\ v\ z \end{aligned}$$

Implementation: solve for ε and $(\oplus)$ on the left. Hint: find μ^{-1}.

Uniform pairs — other classes

Exercise: apply the same principle for

- *Functor*
- *Applicative*
- *Monad*
- *Foldable*
- *Traversable*

Example: streams

$$\textbf{data}\ \mathit{Stream}\ a = \mathit{Cons}\ a\ (\mathit{Stream}\ a)$$

API: same classes as with *Pair*.

Denotation? Hint: *Stream* is also an indexable type.

$$\textbf{data}\ S\ a = \mathit{Nat} \to a$$

$$\textbf{data}\ \mathit{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat}$$

Interpret *Stream* as S:

$$\begin{aligned}
&\mu :: \mathit{Stream}\ a \to S\ a \\
&\mu\ (\mathit{Cons}\ a\ _)\ \ \mathit{Zero} \qquad = a \\
&\mu\ (\mathit{Cons}\ _\ as)\ (\mathit{Succ}\ n) = \mu\ as\ n
\end{aligned}$$

Memo tries

Generalizes *Pair* and *Stream*:

$$\textbf{type}\ a \twoheadrightarrow b$$

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

API: classes as above, plus *Category*.

Exploit inverses to calculate instances, e.g.,

$$\mu\ id \equiv id$$
$$\Leftarrow id \equiv \mu^{-1}\ id$$

$$\mu\ (g \circ f) \equiv \mu\ g \circ \mu\ f$$
$$\Leftarrow g \circ f \equiv \mu^{-1}\ (\mu\ g \circ \mu\ f)$$

Then simplify/optimize.

Example: lists with a bonus

$$\textbf{data}\ ListX\ a\ b = Done\ b \mid Cons\ a\ (ListX\ a\ b)$$

Denotation:

$$\begin{aligned}
&\mu :: ListX\ a\ b \rightarrow ([a], b) \\
&\mu\ (Done\ b) \qquad = ([\,], b) \\
&\mu\ (Cons\ a\ asb) = (a : as, b)\ \textbf{where}\ (as, b) = \mu\ asb
\end{aligned}$$

Exercise: instances, including

$$\textbf{instance}\ Monad\ (ListX\ a)\ \textbf{where} \ldots$$

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:

$$\textbf{type}\ (:\!\multimap) :: * \to * \to *$$

$$scale :: Num\ s \Rightarrow (s :\!\multimap s)$$
$$\widehat{id} \quad :: a :\!\multimap a$$
$$(\hat{\circ}) \quad :: (b :\!\multimap c) \to (a :\!\multimap b) \to (a :\!\multimap c)$$
$$\ldots$$

Model:

$$\textbf{type}\ a \multimap b \quad \text{-- Linear subset of } a \to b$$

$$\mu :: (a :\!\multimap b) \to (a \multimap b)$$

Specification:

$$\mu\ (scale\ s) \equiv \lambda x \to s \times x$$
$$\mu\ \widehat{id} \qquad \equiv id$$
$$\mu\ (g\ \hat{\circ}\ f) \quad \equiv \mu\ g \circ \mu\ f$$
$$\ldots$$

Representation

Start with 1D. Recall partial specification:

$$\mu\ (scale\ s) \equiv \lambda x \rightarrow s \times x$$

Try a direct data type representation:

$$\textbf{data}\ (:\multimap) :: * \rightarrow * \rightarrow *\ \ \textbf{where}$$
$$\quad Scale :: Num\ s \Rightarrow s \rightarrow (s :\multimap s) \quad \text{-- ...}$$
$$\mu :: (a :\multimap b) \rightarrow (a \multimap b)$$
$$\mu\ (Scale\ s) = \lambda x \rightarrow s \times x$$

Spec trivially satisfied by $scale = Scale$.

Others are more interesting.

Calculate an implementation

Specification:

$$\mu\ \widehat{id} \equiv id$$

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

Calculation:

$$\begin{aligned} & id \\ \equiv\ & \lambda x \rightarrow x \\ \equiv\ & \lambda x \rightarrow 1 \times x \\ \equiv\ & \mu\ (Scale\ 1) \end{aligned}$$

$$\begin{aligned} & \mu\ (Scale\ s) \circ \mu\ (Scale\ s') \\ \equiv\ & (\lambda x \rightarrow s \times x) \circ (\lambda x' \rightarrow s' \times x') \\ \equiv\ & \lambda x' \rightarrow s \times (s' \times x') \\ \equiv\ & \lambda x' \rightarrow ((s \times s') \times x') \\ \equiv\ & \mu\ (Scale\ (s \times s')) \end{aligned}$$

Sufficient definitions:

$$\widehat{id} = Scale\ 1$$

$$Scale\ s\ \hat{\circ}\ Scale\ s' = Scale\ (s \times s')$$

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 $(:\!\!-\!\!\circ)$?

Category

Interface:

$$\textbf{class } \mathit{Category}\ k \textbf{ where}$$
$$\quad \mathit{id} :: k\ a\ a$$
$$\quad (\circ) :: k\ b\ c \rightarrow k\ a\ b \rightarrow k\ a\ c$$

Laws:

$$\mathit{id} \circ f \equiv f$$
$$g \circ \mathit{id} \equiv g$$
$$(h \circ g) \circ f \equiv h \circ (g \circ f)$$

Linear transformation category

Linear map semantics:

$$\mu :: (a :\multimap b) \to (a \multimap b)$$
$$\mu\ (Scale\ s) = \lambda x \to s \times x$$

Specification as homomorphism (no abstraction leak):

$$\mu\ id \equiv id$$
$$\mu\ (g \circ f) \equiv \mu\ g \circ \mu\ f$$

Correct-by-construction implementation:

$$\textbf{instance}\ Category\ (:\multimap)\ \textbf{where}$$
$$id = Scale\ 1$$
$$Scale\ s \circ Scale\ s' = Scale\ (s \times s')$$

Laws for free

$$\begin{aligned} \mu\ id &\equiv id \\ \mu\ (g \circ f) &\equiv \mu\ g \circ \mu\ f \end{aligned} \quad \Rightarrow \quad \begin{aligned} id \circ f &\equiv f \\ g \circ id &\equiv g \\ (h \circ g) \circ f &\equiv h \circ (g \circ f) \end{aligned}$$

where equality is *semantic*. Proofs:

$$\begin{aligned} &\mu\ (id \circ f) \\ \equiv\ &\mu\ id \circ \mu\ f \\ \equiv\ &id \circ \mu\ f \\ \equiv\ &\mu\ f \end{aligned} \qquad \begin{aligned} &\mu\ (g \circ id) \\ \equiv\ &\mu\ g \circ \mu\ id \\ \equiv\ &\mu\ g \circ id \\ \equiv\ &\mu\ g \end{aligned} \qquad \begin{aligned} &\mu\ ((h \circ g) \circ f) \\ \equiv\ &(\mu\ h \circ \mu\ g) \circ \mu\ f \\ \equiv\ &\mu\ h \circ (\mu\ g \circ \mu\ f) \\ \equiv\ &\mu\ (h \circ (g \circ f)) \end{aligned}$$

Works for other classes as well.

Higher dimensions

Interface:

$$(\triangle) :: (a :\!\multimap c) \rightarrow (a :\!\multimap d) \rightarrow (a :\!\multimap c \times d)$$
$$(\triangledown) :: (a :\!\multimap c) \rightarrow (b :\!\multimap c) \rightarrow (a \times b :\!\multimap c)$$

Semantics:

$$\mu\ (f \vartriangle g) \equiv \lambda a \rightarrow (f\ a, g\ a)$$
$$\mu\ (f \triangledown g) \equiv \lambda (a, b) \rightarrow f\ a + g\ b$$

Products and coproducts

$$
\begin{array}{l}
\textbf{class } \mathit{Category}\ k \Rightarrow \mathit{ProductCat}\ k \textbf{ where} \\
\quad \textbf{type } a \times_k b \\
\quad \mathit{exl} :: k\ (a \times_k b)\ a \\
\quad \mathit{exr} :: k\ (a \times_k b)\ b \\
\quad (\vartriangle) :: k\ a\ c \rightarrow k\ a\ d \rightarrow k\ a\ (c \times_k d)
\end{array}
$$

$$
\begin{array}{l}
\textbf{class } \mathit{Category}\ k \Rightarrow \mathit{CoproductCat}\ k \textbf{ where} \\
\quad \textbf{type } a +_k b \\
\quad \mathit{inl} :: k\ a\ (a +_k b) \\
\quad \mathit{inr} :: k\ b\ (a +_k b) \\
\quad (\triangledown) :: k\ a\ c \rightarrow k\ b\ c \rightarrow k\ (a +_k b)\ c
\end{array}
$$

Similar to *Arrow* and *ArrowChoice* classes.

Semantic morphisms

$$\begin{aligned} \mu\ \mathit{exl} &\equiv \mathit{exl} \\ \mu\ \mathit{exr} &\equiv \mathit{exr} \\ \mu\ (f \vartriangle g) &\equiv \mu\ f \vartriangle \mu\ g \end{aligned}$$

$$\begin{aligned} \mu\ \mathit{inl} &\equiv \mathit{inl} \\ \mu\ \mathit{inr} &\equiv \mathit{inr} \\ \mu\ (f \triangledown g) &\equiv \mu\ f \triangledown \mu\ g \end{aligned}$$

For $a \multimap b$,

$$\begin{aligned} &\textbf{type}\ a \times_{(\multimap)} b = a \times b \\ &\mathit{exl}\ \ (a, b) = a \\ &\mathit{exr}\ (a, b) = b \\ &f \vartriangle g = \lambda a \rightarrow (f\ \ a, g\ \ a) \end{aligned}$$

$$\begin{aligned} &\textbf{type}\ a +_{(\multimap)} b = a \times b \\ &\mathit{inl}\ \ a = (a, 0) \\ &\mathit{inr}\ b = (0, b) \\ &f \triangledown g = \lambda(a, b) \rightarrow f\ \ a + g\ \ b \end{aligned}$$

For calculation, see blog post *Reimagining matrices*.

Full representation and denotation

$$
\begin{aligned}
&\textbf{data}\ (:\multimap) :: * \to * \to *\ \ \textbf{where} \\
&\quad \mathit{Scale} :: \mathit{Num}\ s \Rightarrow s \to (s :\multimap s) \\
&\quad (:\vartriangle) :: (a :\multimap c) \to (a :\multimap d) \to (a :\multimap c \times d) \\
&\quad (:\triangledown) :: (a :\multimap c) \to (b :\multimap c)\ \to (a \times b :\multimap c)
\end{aligned}
$$

$$
\begin{aligned}
&\mu :: (a :\multimap b) \to (a \multimap b) \\
&\mu\ (\mathit{Scale}\ s) = \lambda x \to s \times x \\
&\mu\ (f :\vartriangle g) \quad = \lambda a \to (f\ a, g\ a) \\
&\mu\ (f :\triangledown g) \quad = \lambda (a, b) \to f\ a + g\ b
\end{aligned}
$$

Denotational design

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