Denotational Design

from meanings to programs

Conal Elliott

Tabula

July, 2014

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*

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

-- shapes, gradients, etc.

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

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?

$$\textbf{type}\ \mathit{Image}$$

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

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

$$\mathit{overC} :: \mathit{Color} \to \mathit{Color} \to \mathit{Color}$$

Note compositionality of μ.

Compositional semantics

Make more explicit:

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

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

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

Generalize and simplify

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

Generalize:

$$
\begin{aligned}
&\textbf{type}\ \mathit{Image}\ a \\
&\textbf{type}\ \mathit{ImageC} = \mathit{Image}\ \mathit{Color} \\
&\textbf{type}\ \mathit{Region}\ \ = \mathit{Image}\ \mathit{Bool}
\end{aligned}
$$

Now some operations become more general.

Generalize and simplify

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

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

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

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

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

Spatial transformation

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

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

where

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

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?

- Flexible transformation with simple & precise semantics
- Efficiency (adapative)
- Quality/accuracy
- Modularity/composability:
  - Fewer assumptions, more uses (resolution-independence).
  - More information available for extraction.
  - Same benefits as pure, non-strict functional programming.
    See *Why Functional Programming Matters*.

Approximations/prunings *compose* badly, so postpone.

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:

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

Laws:

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

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

What monoids have we seen today?

Image monoid

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

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

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

Do these instances satisfy the *Monoid* laws?

Functor

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

For images?

$$\textbf{instance}\ \mathit{Functor}\ \mathit{Image}\ \textbf{where}$$
$$\quad (\mathbin{<\$>}) = \mathit{lift}_1$$

Laws?

Applicative

$$
\begin{array}{l}
\textbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f\ \textbf{where} \\
\quad \mathit{pure}\ \ :: a \rightarrow f\ a \\
\quad (\circledast) :: f\ (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b
\end{array}
$$

For images?

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

From *Applicative*,

$$
\begin{array}{l}
\mathit{liftA}_2\ f\ p\ q\ \ \ \ = f \mathbin{\langle\$\rangle} p \circledast q \\
\mathit{liftA}_3\ f\ p\ q\ r = f \mathbin{\langle\$\rangle} p \circledast q \circledast r \\
\quad \text{-- etc}
\end{array}
$$

Laws?

Instance semantics

Monoid:

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

Functor:

$$
\begin{aligned}
\mu\ (f \mathbin{<\$>} im) &\equiv \lambda p \to f\ (im\ p) \\
&\equiv f \circ im
\end{aligned}
$$

Applicative:

$$
\begin{aligned}
\mu\ (pure\ a) &\equiv \lambda p \to a \\
\mu\ (imf \mathbin{<*>} imx) &\equiv \lambda p \to (imf\ p)\ (imx\ p)
\end{aligned}
$$

Monad and *Comonad*

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

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

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:

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

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:

$$\mu\ (f \mathbin{<\$>} im) \equiv f \circ \mu\ im$$

Instance for the semantic model:

$$\begin{array}{l}\mathbf{instance}\ \mathit{Functor}\ ((\rightarrow)\ u)\ \mathbf{where} \\ \quad f \mathbin{<\$>} h = f \circ h\end{array}$$

Refactoring,

$$\mu\ (f \mathbin{<\$>} im) \equiv f \mathbin{<\$>} \mu\ im$$

So μ is a *functor* homomorphism.

Applicative specification, revisited

Applicative specification:

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

Instance for the semantic model:

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

Refactoring,

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

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

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

where equality is *semantic*. Proofs:

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

Works for other classes as well.

Example – linear transformations

Assignment:

- Represent linear transformations
- Implement identity and composition

Plan:

- Interface
- Denotation
- Representation
- Calculation (implementation)

Interface and denotation

Interface:

$$\begin{array}{l}
\textbf{type}\ (:\!\multimap) :: * \to * \to * \\[4pt]
\mathit{scale} :: \mathit{Num}\ s \Rightarrow (s :\!\multimap s) \\
\widehat{\mathit{id}} \quad :: a :\!\multimap a \\
(\hat{\circ}) \quad :: (b :\!\multimap c) \to (a :\!\multimap b) \to (a :\!\multimap c) \\
\ldots
\end{array}$$

Model:

$$\begin{array}{l}
\textbf{type}\ a \multimap b \quad \text{-- Linear subset of } a \to b \\[4pt]
\mu :: (a :\!\multimap b) \to (a \multimap b)
\end{array}$$

Specification:

$$\begin{array}{ll}
\mu\ (\mathit{scale}\ s) & \equiv \lambda x \to s \times x \\
\mu\ \widehat{\mathit{id}} & \equiv \mathit{id} \\
\mu\ (g\ \hat{\circ}\ f) & \equiv \mu\ g \circ \mu\ f \\
\ldots &
\end{array}$$

Representation

Start with 1D. Recall partial specification:

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

Try a direct data type representation:

$$\begin{array}{l} \mathbf{data}\ (:\!\multimap) :: * \rightarrow * \rightarrow * \ \ \mathbf{where} \\ \quad Scale :: Num\ s \Rightarrow s \rightarrow (s :\!\multimap s) \quad \text{-- ...} \end{array}$$

$$\begin{array}{l} \mu :: (a :\!\multimap b) \rightarrow (a \multimap b) \\ \mu\ (Scale\ s) = \lambda x \rightarrow s \times x \end{array}$$

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:

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

Laws:

$$
\begin{array}{ll}
\mathit{id} \circ f & \equiv f \\
g \circ \mathit{id} & \equiv g \\
(h \circ g) \circ f & \equiv h \circ (g \circ f)
\end{array}
$$

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:

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

Laws for free

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

where equality is *semantic*. Proofs:

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

Works for other classes as well.

Higher dimensions

Interface:

$$(\vartriangle) :: (a :\!\multimap c) \to (a :\!\multimap d) \to (a :\!\multimap c \times d)$$
$$(\triangledown) :: (a :\!\multimap c) \to (b :\!\multimap c) \to (a \times b :\!\multimap c)$$

Semantics:

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

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 $a \multimap b$,

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

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

For calculation, see blog post *Reimagining matrices*.

Full representation and denotation

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

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

Functional reactive programming

Two essential properties:

- *Continuous* time! (Natural & composable.)
- Denotational design. (Elegant & rigorous.)

Deterministic, continuous "concurrency".

More aptly, *"Denotative continuous-time programming"* (DCTP).

Warning: many modern "FRP" systems have neither property.

Denotational design

Central type:

$$\textbf{type}\ \mathit{Behavior}\ a$$

Model:

$$\mu :: \mathit{Behavior}\ a \to (\mathbb{R} \to a)$$

Suggests API and semantics (via morphisms).

What standard algebraic abstractions does the model inhabit?

Monoid, *Functor*, *Applicative*, *Monad*, *Comonad*.

Functor

$$\textbf{instance}\ \mathit{Functor}\ ((\rightarrow)\ t)\ \textbf{where}$$
$$\quad f <\$> h = f \circ h$$

Morphism:

$$
\begin{aligned}
&\ \ \mu\ (f <\$> b) \\
&\equiv f <\$> \mu\ b \\
& \\
&\equiv f \circ \mu\ b
\end{aligned}
$$

Applicative

$$\textbf{instance}\ \mathit{Applicative}\ ((\rightarrow)\ t)\ \textbf{where}$$
$$\mathit{pure}\ a \;\;= \lambda t \rightarrow a$$
$$g \circledast h = \lambda t \rightarrow (g\ t)\ (h\ t)$$

Morphisms:

$$\begin{array}{l} \quad \mu\ (\mathit{pure}\ a) \\ \equiv \mathit{pure}\ a \\ \\ \equiv \lambda t \rightarrow a \end{array}$$

$$\begin{array}{l} \quad \mu\ (\mathit{fs} \circledast \mathit{xs}) \\ \equiv \mu\ \mathit{fs} \circledast \mu\ \mathit{xs} \\ \\ \equiv \lambda t \rightarrow (\mu\ \mathit{fs}\ t)\ (\mu\ \mathit{xs}\ t) \end{array}$$

Corresponds exactly to the original FRP denotation.

Monad

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ ((\to)\ t)\ \textbf{where} \\
\quad \mathit{join}\ \mathit{ff} = \lambda t \to \mathit{ff}\ t\ t
\end{array}
$$

Morphism:

$$
\boxed{
\begin{array}{l}
\quad \mu\ (\mathit{join}\ bb) \\
\equiv \mathit{join}\ (\mu \mathbin{<\$>} \mu\ bb) \\
\\
\equiv \mathit{join}\ (\mu \circ \mu\ bb) \\
\equiv \lambda t \to (\mu \circ \mu\ bb)\ t\ t \\
\equiv \lambda t \to \mu\ (\mu\ bb\ t)\ t
\end{array}
}
$$

Comonad

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

Functions:

$$
\begin{aligned}
&\textbf{instance } \mathit{Monoid}\ t \Rightarrow \mathit{Comonad}\ ((\rightarrow)\ t)\ \textbf{where} \\
&\quad \mathit{coreturn} :: (t \rightarrow a) \rightarrow a \\
&\quad \mathit{coreturn}\ f = f\ \varepsilon \\
&\quad \mathit{cojoin}\ f = \lambda t\ t' \rightarrow f\ (t \oplus t')
\end{aligned}
$$

Suggest a relative time model.

Why continuous & infinite (vs discrete/finite) time?

- Transformation flexibility with simple & precise semantics
- Efficiency (adapative)
- Quality/accuracy
- Modularity/composability:
  - Fewer assumptions, more uses (resolution-independence).
  - More info available for extraction.
  - Same benefits as pure, non-strict functional programming. See *Why Functional Programming Matters*.
- Integration and differentiation: natural, accurate, efficient.
- Reconcile differing input sampling rates.

Approximations/prunings compose badly, so postpone.

Memo tries

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

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

This time, μ has an inverse.

Exploit inverses to calculate instances. Example:

$$\boxed{\begin{array}{l} \quad \mu\ id \equiv id \\ \Leftarrow id \equiv \mu^{-1}\ id \end{array}}$$

$$\boxed{\begin{array}{l} \quad \mu\ (g \circ f) \equiv \mu\ g \circ \mu\ f \\ \Leftarrow g \circ f \equiv \mu^{-1}\ (\mu\ g \circ \mu\ f) \end{array}}$$

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
- This talk