

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.

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

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

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)

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

Overview

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

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

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

API first pass

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?

How to implement?

wrong first question

What to implement?

What to implement?

- What do these operations mean?

What to implement?

- What do these operations mean?
- More centrally: What do the *types* mean?

What is an image?

What is an image?

Specification goals:

What is an image?

Specification goals:

- Adequate
- Simple
- Precise

What is an image?

Specification goals:

- Adequate
- Simple
- Precise

Why these properties?

What is an image?

What is an image?

My answer: assignment of colors to 2D locations.

What is an image?

My answer: assignment of colors to 2D locations.

How to make precise?

type *Image*

What is an image?

My answer: assignment of colors to 2D locations.

How to make precise?

type *Image*

Model:

$\mu :: \textit{Image} \rightarrow (\textit{Loc} \rightarrow \textit{Color})$

What is an image?

My answer: assignment of colors to 2D locations.

How to make precise?

type *Image*

Model:

$\mu :: Image \rightarrow (Loc \rightarrow Color)$

What about regions?

What is an image?

My answer: assignment of colors to 2D locations.

How to make precise?

type *Image*

Model:

$$\mu :: \textit{Image} \rightarrow (\textit{Loc} \rightarrow \textit{Color})$$

What about regions?

$$\mu :: \textit{Region} \rightarrow (\textit{Loc} \rightarrow \textit{Bool})$$

Specifying *Image* operations

μ (*over top bot*) $\equiv \dots$

μ (*crop reg im*) $\equiv \dots$

μ (*monochrome c*) $\equiv \dots$

μ (*transform tr im*) $\equiv \dots$

Specifying *Image* operations

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

$\text{overC} :: \text{Color} \rightarrow \text{Color} \rightarrow \text{Color}$

Specifying *Image* operations

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

$\text{overC} :: \text{Color} \rightarrow \text{Color} \rightarrow \text{Color}$

Note compositionality of μ .

Compositional semantics

Make more explicit:

$$\mu (\textit{over top bot}) \equiv \textit{overS} (\mu \textit{top}) (\mu \textit{bot})$$
$$\mu (\textit{crop reg im}) \equiv \textit{cropS} (\mu \textit{reg}) (\mu \textit{im})$$
$$\textit{overS} :: (\textit{Loc} \rightarrow \textit{Color}) \rightarrow (\textit{Loc} \rightarrow \textit{Color}) \rightarrow (\textit{Loc} \rightarrow \textit{Color})$$
$$\textit{overS} f g = \lambda p \rightarrow \textit{overC} (f p) (g p)$$
$$\textit{cropS} :: (\textit{Loc} \rightarrow \textit{Bool}) \rightarrow (\textit{Loc} \rightarrow \textit{Color}) \rightarrow (\textit{Loc} \rightarrow \textit{Color})$$
$$\textit{cropS} f g = \lambda p \rightarrow \mathbf{if} f p \mathbf{then} g p \mathbf{else} \textit{clear}$$

Generalize and simplify

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

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

transform :: *Transform* \rightarrow *Image a* \rightarrow *Image a*

cond :: *Image Bool* \rightarrow *Image a* \rightarrow *Image a* \rightarrow *Image a*

Generalize and simplify

$transform :: Transform \rightarrow Image\ a \rightarrow Image\ a$

$cond \quad \quad :: Image\ Bool \rightarrow Image\ a \rightarrow Image\ a \rightarrow Image\ a$

$lift_0 :: a \rightarrow Image\ a$

$lift_1 :: (a \rightarrow b) \rightarrow (Image\ a \rightarrow Image\ b)$

$lift_2 :: (a \rightarrow b \rightarrow c) \rightarrow (Image\ a \rightarrow Image\ b \rightarrow Image\ c)$

...

Generalize and simplify

$transform :: Transform \rightarrow Image\ a \rightarrow Image\ a$

$cond \quad \quad :: Image\ Bool \rightarrow Image\ a \rightarrow Image\ a \rightarrow Image\ a$

$lift_0 :: a \rightarrow Image\ a$

$lift_1 :: (a \rightarrow b) \rightarrow (Image\ a \rightarrow Image\ b)$

$lift_2 :: (a \rightarrow b \rightarrow c) \rightarrow (Image\ a \rightarrow Image\ b \rightarrow Image\ c)$

...

Specializing,

$monochrome = lift_0$

$over \quad \quad = lift_2\ overC$

$crop\ r\ im \quad = cond\ r\ im\ emptyIm$

$cond \quad \quad = lift_3\ ifThenElse$

Spatial transformation

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

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

Spatial transformation

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

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

where

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

Spatial transformation

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

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

where

$\textit{transformS} :: (\textit{Loc} \rightarrow \textit{Loc}) \rightarrow (\textit{Loc} \rightarrow \textit{Color}) \rightarrow (\textit{Loc} \rightarrow \textit{Color})$

Spatial transformation

$$\mu :: \textit{Transform} \rightarrow (\textit{Loc} \rightarrow \textit{Loc})$$
$$\mu (\textit{transform} \textit{tr} \textit{im}) \equiv \textit{transformS} (\mu \textit{tr}) (\mu \textit{im})$$

where

$$\textit{transformS} :: (\textit{Loc} \rightarrow \textit{Loc}) \rightarrow (\textit{Loc} \rightarrow \textit{Color}) \rightarrow (\textit{Loc} \rightarrow \textit{Color})$$
$$\textit{transformS} \textit{h} \textit{f} = \lambda p \rightarrow \textit{f} (\textit{h} \textit{p})$$

Subtle implications.

Spatial transformation

$$\mu :: \textit{Transform} \rightarrow (\textit{Loc} \rightarrow \textit{Loc})$$
$$\mu (\textit{transform} \textit{tr} \textit{im}) \equiv \textit{transformS} (\mu \textit{tr}) (\mu \textit{im})$$

where

$$\textit{transformS} :: (\textit{Loc} \rightarrow \textit{Loc}) \rightarrow (\textit{Loc} \rightarrow \textit{Color}) \rightarrow (\textit{Loc} \rightarrow \textit{Color})$$
$$\textit{transformS} \textit{h} \textit{f} = \lambda p \rightarrow \textit{f} (\textit{h} \textit{p})$$

Subtle implications.

What is *Loc*?

Spatial transformation

$$\mu :: \textit{Transform} \rightarrow (\textit{Loc} \rightarrow \textit{Loc})$$
$$\mu (\textit{transform} \textit{tr} \textit{im}) \equiv \textit{transformS} (\mu \textit{tr}) (\mu \textit{im})$$

where

$$\textit{transformS} :: (\textit{Loc} \rightarrow \textit{Loc}) \rightarrow (\textit{Loc} \rightarrow \textit{Color}) \rightarrow (\textit{Loc} \rightarrow \textit{Color})$$
$$\textit{transformS} \textit{h} \textit{f} = \lambda p \rightarrow \textit{f} (\textit{h} \textit{p})$$

Subtle implications.

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

type *Loc* = \mathbb{R}^2

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

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

- Flexible transformation with simple & precise semantics

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

- Flexible transformation with simple & precise semantics
- Efficiency (adaptive)

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

- Flexible transformation with simple & precise semantics
- Efficiency (adaptive)
- Quality/accuracy

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

- Flexible transformation with simple & precise semantics
- Efficiency (adaptive)
- Quality/accuracy
- Modularity/composability

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

- Flexible transformation with simple & precise semantics
- Efficiency (adaptive)
- 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*.

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

- Flexible transformation with simple & precise semantics
- Efficiency (adaptive)
- 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.

Pan gallery

Using standard vocabulary

Using standard vocabulary

- We've created a domain-specific vocabulary.
- Can we reuse standard vocabularies instead?
- Why would we want to?

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.

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:

class *Monoid* *m* **where**

ε :: *m* -- “mempty”

(\oplus) :: *m* → *m* → *m* -- “mappend”

Monoid

Interface:

class *Monoid* *m* **where**

ε **::** *m* -- “mempty”

(\oplus) **::** *m* \rightarrow *m* \rightarrow *m* -- “mappend”

Laws:

$$a \oplus \varepsilon \equiv a$$

$$\varepsilon \oplus b \equiv b$$

$$a \oplus (b \oplus c) \equiv (a \oplus b) \oplus c$$

Monoid

Interface:

class *Monoid* *m* **where**

ε :: *m* -- “mempty”

(\oplus) :: *m* → *m* → *m* -- “mappend”

Laws:

$$a \oplus \varepsilon \equiv a$$

$$\varepsilon \oplus b \equiv b$$

$$a \oplus (b \oplus c) \equiv (a \oplus b) \oplus c$$

Why do laws *matter*?

Monoid

Interface:

class *Monoid* *m* **where**

ε **::** *m* -- “mempty”

(\oplus) **::** *m* \rightarrow *m* \rightarrow *m* -- “mappend”

Laws:

$$a \oplus \varepsilon \quad \equiv \quad a$$

$$\varepsilon \oplus b \quad \equiv \quad b$$

$$a \oplus (b \oplus c) \equiv (a \oplus b) \oplus c$$

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

Monoid

Interface:

class *Monoid* *m* **where**

ε :: *m* -- “mempty”

(\oplus) :: *m* → *m* → *m* -- “mappend”

Laws:

$$a \oplus \varepsilon \equiv a$$

$$\varepsilon \oplus b \equiv b$$

$$a \oplus (b \oplus c) \equiv (a \oplus b) \oplus c$$

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

What monoids have we seen today?

Image monoid

Image monoid

instance *Monoid ImageC* **where**

ε = *lift₀ clear*

(\oplus) = *over*

Image monoid

instance *Monoid ImageC* **where**

ε = *lift₀ clear*

(\oplus) = *over*

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

Image monoid

instance *Monoid ImageC* **where**

ε = *lift₀ clear*

(\oplus) = *over*

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

instance *Monoid a* \Rightarrow *Monoid (Image a)* **where**

ε = *lift₀ ε*

(\oplus) = *lift₂ (⊕)*

Image monoid

instance *Monoid ImageC* **where**

ε = *lift₀ clear*

(\oplus) = *over*

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

instance *Monoid a* \Rightarrow *Monoid (Image a)* **where**

ε = *lift₀ ε*

(\oplus) = *lift₂ (⊕)*

Do these instances satisfy the *Monoid* laws?

Functor

class *Functor* *f* **where**

$(\diamond) :: (a \rightarrow b) \rightarrow (f\ a \rightarrow f\ b)$

Functor

class *Functor* *f* **where**

$(\langle \$ \rangle) :: (a \rightarrow b) \rightarrow (f\ a \rightarrow f\ b)$

For images?

Functor

class *Functor* *f* **where**

$(\llbracket \$ \rrbracket) :: (a \rightarrow b) \rightarrow (f\ a \rightarrow f\ b)$

For images?

instance *Functor* *Image* **where**

$(\llbracket \$ \rrbracket) = \text{lift}_1$

Functor

class *Functor* *f* **where**

$(\langle \$ \rangle) :: (a \rightarrow b) \rightarrow (f\ a \rightarrow f\ b)$

For images?

instance *Functor* *Image* **where**

$(\langle \$ \rangle) = \mathit{lift}_1$

Laws?

Applicative

class *Functor* $f \Rightarrow$ *Applicative* f **where**

pure $:: a \rightarrow f\ a$

$(\langle * \rangle) :: f\ (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$

Applicative

class *Functor* $f \Rightarrow$ *Applicative* f **where**

pure $:: a \rightarrow f\ a$

$(\langle * \rangle) :: f\ (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$

For images?

Applicative

class *Functor* $f \Rightarrow$ *Applicative* f **where**

pure $:: a \rightarrow f\ a$

$(\langle * \rangle) :: f\ (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$

For images?

instance *Applicative* *Image* **where**

pure = *lift*₀

$(\langle * \rangle) = \text{lift}_2\ (\$)$

From *Applicative*,

$\text{lift}A_2\ f\ p\ q = f\ \langle \$ \rangle\ p\ \langle * \rangle\ q$

$\text{lift}A_3\ f\ p\ q\ r = f\ \langle \$ \rangle\ p\ \langle * \rangle\ q\ \langle * \rangle\ r$

-- etc

Applicative

class *Functor* $f \Rightarrow$ *Applicative* f **where**

pure $:: a \rightarrow f\ a$

$(\langle * \rangle) :: f\ (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$

For images?

instance *Applicative* *Image* **where**

pure = *lift*₀

$(\langle * \rangle) = \text{lift}_2\ (\$)$

From *Applicative*,

$\text{lift}A_2\ f\ p\ q = f\ \langle \$ \rangle\ p\ \langle * \rangle\ q$

$\text{lift}A_3\ f\ p\ q\ r = f\ \langle \$ \rangle\ p\ \langle * \rangle\ q\ \langle * \rangle\ r$

-- etc

Laws?

Instance semantics

Instance semantics

Monoid:

$$\mu \varepsilon \quad \equiv \quad \lambda p \rightarrow \varepsilon$$

$$\mu (top \oplus bot) \equiv \lambda p \rightarrow \mu top p \oplus \mu bot p$$

Instance semantics

Monoid:

$$\mu \varepsilon \quad \equiv \lambda p \rightarrow \varepsilon$$

$$\mu (top \oplus bot) \equiv \lambda p \rightarrow \mu top p \oplus \mu bot p$$

Functor:

$$\mu (f \langle \$ \rangle im) \equiv \lambda p \rightarrow f (im p)$$

$$\equiv f \circ im$$

Instance semantics

Monoid:

$$\mu \varepsilon \quad \equiv \lambda p \rightarrow \varepsilon$$

$$\mu (top \oplus bot) \equiv \lambda p \rightarrow \mu top p \oplus \mu bot p$$

Functor:

$$\mu (f \langle \$ \rangle im) \equiv \lambda p \rightarrow f (im p)$$

$$\equiv f \circ im$$

Applicative:

$$\mu (pure a) \quad \equiv \lambda p \rightarrow a$$

$$\mu (imf \langle * \rangle imx) \equiv \lambda p \rightarrow (imf p) (imx p)$$

Monad and Comonad

class *Monad* *f* **where**

return :: $a \rightarrow f\ a$

join :: $f\ (f\ a) \rightarrow f\ a$

class *Functor* *f* \Rightarrow *Comonad* *f* **where**

coreturn :: $f\ a \rightarrow a$

cojoin :: $f\ a \rightarrow f\ (f\ a)$

Monoid specification, revisited

Image monoid specification:

$$\mu \varepsilon \quad \equiv \quad \lambda p \rightarrow \varepsilon$$

$$\mu (top \oplus bot) \equiv \lambda p \rightarrow \mu top p \oplus \mu bot p$$

Monoid specification, revisited

Image monoid specification:

$$\mu \varepsilon \quad \equiv \lambda p \rightarrow \varepsilon$$

$$\mu (top \oplus bot) \equiv \lambda p \rightarrow \mu top p \oplus \mu bot p$$

Instance for the semantic model:

instance *Monoid* $v \Rightarrow$ *Monoid* $(u \rightarrow v)$ **where**

$$\varepsilon \quad = \lambda u \rightarrow \varepsilon$$

$$f \oplus g = \lambda u \rightarrow f u \oplus g u$$

Monoid specification, revisited

Image monoid specification:

$$\mu \varepsilon \quad \equiv \lambda p \rightarrow \varepsilon$$

$$\mu (top \oplus bot) \equiv \lambda p \rightarrow \mu top p \oplus \mu bot p$$

Instance for the semantic model:

instance *Monoid* $v \Rightarrow$ *Monoid* $(u \rightarrow v)$ **where**

$$\varepsilon \quad = \lambda u \rightarrow \varepsilon$$

$$f \oplus g = \lambda u \rightarrow f u \oplus g u$$

Refactoring,

$$\mu \varepsilon \quad \equiv \varepsilon$$

$$\mu (top \oplus bot) \equiv \mu top \oplus \mu bot$$

Monoid specification, revisited

Image monoid specification:

$$\mu \varepsilon \quad \equiv \lambda p \rightarrow \varepsilon$$

$$\mu (top \oplus bot) \equiv \lambda p \rightarrow \mu top p \oplus \mu bot p$$

Instance for the semantic model:

instance *Monoid* $v \Rightarrow$ *Monoid* $(u \rightarrow v)$ **where**

$$\varepsilon \quad = \lambda u \rightarrow \varepsilon$$

$$f \oplus g = \lambda u \rightarrow f u \oplus g u$$

Refactoring,

$$\mu \varepsilon \quad \equiv \varepsilon$$

$$\mu (top \oplus bot) \equiv \mu top \oplus \mu bot$$

So μ *distributes* over monoid operations

Monoid specification, revisited

Image monoid specification:

$$\mu \varepsilon \quad \equiv \lambda p \rightarrow \varepsilon$$

$$\mu (top \oplus bot) \equiv \lambda p \rightarrow \mu top p \oplus \mu bot p$$

Instance for the semantic model:

instance *Monoid* $v \Rightarrow$ *Monoid* $(u \rightarrow v)$ **where**

$$\varepsilon \quad = \lambda u \rightarrow \varepsilon$$

$$f \oplus g = \lambda u \rightarrow f u \oplus g u$$

Refactoring,

$$\mu \varepsilon \quad \equiv \varepsilon$$

$$\mu (top \oplus bot) \equiv \mu top \oplus \mu bot$$

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

Functor specification, revisited

Functor specification:

$$\mu (f \langle \$ \rangle im) \equiv f \circ \mu im$$

Functor specification, revisited

Functor specification:

$$\mu (f \langle \$ \rangle im) \equiv f \circ \mu im$$

Instance for the semantic model:

instance *Functor* ((\rightarrow) *u*) **where**

$$f \langle \$ \rangle h = f \circ h$$

Refactoring,

$$\mu (f \langle \$ \rangle im) \equiv f \langle \$ \rangle \mu im$$

So μ is a *functor* homomorphism.

Applicative specification, revisited

Applicative specification:

$$\mu (\text{pure } a) \quad \equiv \lambda p \rightarrow a$$

$$\mu (\text{imf } \langle * \rangle \text{ imx}) \equiv \lambda p \rightarrow (\mu \text{ imf } p) (\mu \text{ imx } p)$$

Applicative specification, revisited

Applicative specification:

$$\mu (\text{pure } a) \quad \equiv \lambda p \rightarrow a$$

$$\mu (\text{imf } \langle * \rangle \text{ imx}) \equiv \lambda p \rightarrow (\mu \text{ imf } p) (\mu \text{ imx } p)$$

Instance for the semantic model:

instance *Applicative* ((\rightarrow) *u*) **where**

$$\text{pure } a \quad = \lambda u \rightarrow a$$

$$\text{fs } \langle * \rangle \text{ xs} = \lambda u \rightarrow (\text{fs } u) (\text{xs } u)$$

Refactoring,

$$\mu (\text{pure } a) \quad \equiv \text{pure } a$$

$$\mu (\text{imf } \langle * \rangle \text{ imx}) \equiv \mu \text{ imf } \langle * \rangle \mu \text{ imx}$$

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

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

where equality is *semantic*.

Laws for free

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

where equality is *semantic*. Proofs:

$\begin{array}{l} \mu (a \oplus \varepsilon) \\ \equiv \mu a \oplus \mu \varepsilon \\ \equiv \mu a \oplus \varepsilon \\ \equiv \mu a \end{array}$	$\begin{array}{l} \mu (\varepsilon \oplus b) \\ \equiv \mu \varepsilon \oplus \mu b \\ \equiv \varepsilon \oplus \mu b \\ \equiv \mu b \end{array}$	$\begin{array}{l} \mu (a \oplus (b \oplus c)) \\ \equiv \mu a \oplus (\mu b \oplus \mu c) \\ \equiv (\mu a \oplus \mu b) \oplus \mu c \\ \equiv \mu ((a \oplus b) \oplus c) \end{array}$
---	---	--

Works for other classes as well.

Example – linear transformations

Assignment:

- Represent linear transformations
- Implement identity and composition

Example – linear transformations

Assignment:

- Represent linear transformations
- Implement identity and composition

Plan:

- Interface
- Denotation
- Representation
- Calculation (implementation)

Interface and denotation

type $(:-\circ) :: * \rightarrow * \rightarrow *$

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

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

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

...

Interface:

Interface and denotation

Interface:

```
type ( $\text{:-}\circ$ )  $:: * \rightarrow * \rightarrow *$   
 $scale :: Num\ s \Rightarrow (s\ \text{:-}\circ\ s)$   
 $\hat{id} \quad :: a\ \text{:-}\circ\ a$   
 $(\hat{\circ}) \quad :: (b\ \text{:-}\circ\ c) \rightarrow (a\ \text{:-}\circ\ b) \rightarrow (a\ \text{:-}\circ\ c)$   
...
```

Model:

```
type  $a\ \text{-}\circ\ b$  -- Linear subset of  $a \rightarrow b$   
 $\mu :: (a\ \text{:-}\circ\ b) \rightarrow (a\ \text{-}\circ\ b)$ 
```

Interface and denotation

Interface:

$$\begin{aligned} \mathbf{type} \quad (& \multimap) :: * \rightarrow * \rightarrow * \\ \mathit{scale} &:: \mathit{Num} \, s \Rightarrow (s \multimap s) \\ \widehat{\mathit{id}} &:: a \multimap a \\ (\widehat{\circ}) &:: (b \multimap c) \rightarrow (a \multimap b) \rightarrow (a \multimap c) \\ &\dots \end{aligned}$$

Model:

$$\begin{aligned} \mathbf{type} \quad a \multimap b & \text{ -- Linear subset of } a \rightarrow b \\ \mu &:: (a \multimap b) \rightarrow (a \rightarrow b) \end{aligned}$$

Specification:

$$\begin{aligned} \mu (\mathit{scale} \, s) &\equiv \lambda x \rightarrow s \times x \\ \mu \widehat{\mathit{id}} &\equiv \mathit{id} \\ \mu (g \widehat{\circ} f) &\equiv \mu g \circ \mu f \\ &\dots \end{aligned}$$

Representation

Start with 1D. Recall partial specification:

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

Try a direct data type representation:

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

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

Others are more interesting.

Calculate an implementation

Specification:

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

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

Calculation:

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

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

Sufficient definitions:

$$\hat{id} = \text{Scale } 1$$

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

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:

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

Laws:

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

Linear transformation category

Linear map semantics:

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

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

Specification as homomorphism (no abstraction leak):

$$\mu \text{ id} \quad \equiv \text{ id}$$

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

Correct-by-construction implementation:

instance *Category* (\multimap) **where**

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

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

Laws for free

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

where equality is *semantic*.

Laws for free

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

where equality is *semantic*. Proofs:

$\begin{array}{l} \mu (id \circ f) \\ \equiv \mu id \circ \mu f \\ \equiv id \circ \mu f \\ \equiv \mu f \end{array}$	$\begin{array}{l} \mu (g \circ id) \\ \equiv \mu g \circ \mu id \\ \equiv \mu g \circ id \\ \equiv \mu g \end{array}$	$\begin{array}{l} \mu ((h \circ g) \circ f) \\ \equiv (\mu h \circ \mu g) \circ \mu f \\ \equiv \mu h \circ (\mu g \circ \mu f) \\ \equiv \mu (h \circ (g \circ f)) \end{array}$
---	---	--

Works for other classes as well.

Higher dimensions

Interface:

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

$$(\nabla) :: (a : \multimap c) \rightarrow (b : \multimap c) \rightarrow (a \times b : \multimap c)$$

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

class *Category* $k \Rightarrow$ *ProductCat* k **where**

type $a \times_k b$

$exl :: k (a \times_k b) a$

$exr :: k (a \times_k b) b$

$(\Delta) :: k a c \rightarrow k a d \rightarrow k a (c \times_k d)$

class *Category* $k \Rightarrow$ *CoproductCat* k **where**

type $a +_k b$

$inl :: k a (a +_k b)$

$inr :: k b (a +_k b)$

$(\nabla) :: k a c \rightarrow k b c \rightarrow k (a +_k b) c$

Similar to *Arrow* and *ArrowChoice* classes.

Semantic morphisms

$$\mu \text{ exl} \quad \equiv \text{exl}$$

$$\mu \text{ exr} \quad \equiv \text{exr}$$

$$\mu (f \triangle g) \equiv \mu f \triangle \mu g$$

$$\mu \text{ inl} \quad \equiv \text{inl}$$

$$\mu \text{ inr} \quad \equiv \text{inr}$$

$$\mu (f \nabla g) \equiv \mu f \nabla \mu g$$

For $a \multimap b$,

$$\mathbf{type} \ a \times_{(\multimap)} b = a \times b$$

$$\text{exl} \ (a, b) = a$$

$$\text{exr} \ (a, b) = b$$

$$f \triangle g = \lambda a \rightarrow (f \ a, g \ a)$$

$$\mathbf{type} \ a +_{(\multimap)} b = a \times b$$

$$\text{inl} \ a = (a, 0)$$

$$\text{inr} \ b = (0, b)$$

$$f \nabla g = \lambda(a, b) \rightarrow f \ a + g \ b$$

For calculation, see blog post *Reimagining matrices*.

Full representation and denotation

data $(:-\circ) :: * \rightarrow * \rightarrow *$ **where**

Scale $:: Num\ s \Rightarrow s \rightarrow (s :-\circ s)$

$(:\triangle) :: (a :-\circ c) \rightarrow (a :-\circ d) \rightarrow (a :-\circ c \times d)$

$(:\nabla) :: (a :-\circ c) \rightarrow (b :-\circ c) \rightarrow (a \times b :-\circ c)$

$\mu :: (a :-\circ b) \rightarrow (a \text{ } \text{ } b)$

$\mu (\textit{Scale}\ s) = \lambda x \rightarrow s \times x$

$\mu (f : \triangle g) = \lambda a \rightarrow (f\ a, g\ a)$

$\mu (f : \nabla g) = \lambda(a, b) \rightarrow f\ a + g\ b$

Functional reactive programming

Functional reactive programming

Two essential properties:

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

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:

type *Behavior a*

Model:

$\mu :: \textit{Behavior a} \rightarrow (\mathbb{R} \rightarrow a)$

Denotational design

Central type:

type *Behavior a*

Model:

$\mu :: \textit{Behavior a} \rightarrow (\mathbb{R} \rightarrow a)$

Suggests API and semantics (via morphisms).

What standard algebraic abstractions does the model inhabit?

Denotational design

Central type:

type *Behavior a*

Model:

$\mu :: \textit{Behavior } a \rightarrow (\mathbb{R} \rightarrow a)$

Suggests API and semantics (via morphisms).

What standard algebraic abstractions does the model inhabit?

Monoid, Functor, Applicative, Monad, Comonad.

instance *Functor* ((\rightarrow) *t*) **where**

$$f \langle \$ \rangle h = f \circ h$$

Morphism:

$$\begin{aligned} & \mu (f \langle \$ \rangle b) \\ \equiv & f \langle \$ \rangle \mu b \\ \\ \equiv & f \circ \mu b \end{aligned}$$

Applicative

instance *Applicative* ((\rightarrow) *t*) **where**

$$\text{pure } a = \lambda t \rightarrow a$$

$$g \langle * \rangle h = \lambda t \rightarrow (g \ t) \ (h \ t)$$

Morphisms:

$$\mu \ (\text{pure } a)$$

$$\equiv \text{pure } a$$

$$\equiv \lambda t \rightarrow a$$

$$\mu \ (fs \ \langle * \rangle \ xs)$$

$$\equiv \mu \ fs \ \langle * \rangle \ \mu \ xs$$

$$\equiv \lambda t \rightarrow (\mu \ fs \ t) \ (\mu \ xs \ t)$$

Corresponds exactly to the original FRP denotation.

Monad

instance *Monad* ((\rightarrow) *t*) **where**

$$\text{join } ff = \lambda t \rightarrow ff\ t\ t$$

Morphism:

$$\begin{aligned} & \mu (\text{join } bb) \\ \equiv & \text{join } (\mu \langle \$ \rangle \mu\ bb) \\ \\ \equiv & \text{join } (\mu \circ \mu\ bb) \\ \equiv & \lambda t \rightarrow (\mu \circ \mu\ bb)\ t\ t \\ \equiv & \lambda t \rightarrow \mu (\mu\ bb\ t)\ t \end{aligned}$$

Comonad

class *Comonad* *w* **where**

coreturn :: $w\ a \rightarrow a$

cojoin :: $w\ a \rightarrow w\ (w\ a)$

Functions:

instance *Monoid* *t* \Rightarrow *Comonad* $((\rightarrow)\ t)$ **where**

coreturn :: $(t \rightarrow a) \rightarrow a$

coreturn *f* = *f* ε

cojoin *f* = $\lambda t\ t' \rightarrow f\ (t \oplus t')$

Suggest a relative time model.

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

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*.

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.

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.

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

type $a \rightarrow b$

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

This time, μ has an inverse.

Exploit inverses to calculate instances. Example:

$$\begin{aligned} \mu \text{ id} &\equiv \text{id} \\ \Leftarrow \text{id} &\equiv \mu^{-1} \text{ id} \end{aligned}$$

$$\begin{aligned} \mu (g \circ f) &\equiv \mu g \circ \mu f \\ \Leftarrow g \circ f &\equiv \mu^{-1} (\mu g \circ \mu f) \end{aligned}$$

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
- This talk