# Denotational Design

## from meanings to programs

Conal Elliott

Tabula

May, 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

# 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

# Library design

Goal: precise, elegant, reusable abstractions.

# Library design

Goal: precise, elegant, reusable abstractions.

Where have such things been developed?

# Library design

Goal: precise, elegant, reusable abstractions.

Where have such things been developed? *Math — abstract algebra.*

# Library design

Goal: precise, elegant, reusable abstractions.

Where have such things been developed? *Math — abstract algebra.*

Non-leaky abstraction ≡ *homomorphism.*

# Library design

Goal: precise, elegant, reusable abstractions.

Where have such things been developed? *Math — abstract algebra.*

Non-leaky abstraction ≡ *homomorphism.*

In Haskell,

# Library design

Goal: precise, elegant, reusable abstractions.

Where have such things been developed? *Math — abstract algebra.*

Non-leaky abstraction ≡ *homomorphism.*

In Haskell,

- Standard type classes
- Laws
- Semantic type class morphisms (TCMs)

# 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*)

# 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.

# 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

Interface:

$$
\begin{aligned}
&\textbf{type}\ (:\!\multimap) :: * \to * \to * \\
&scale :: Num\ s \Rightarrow (s :\!\multimap s) \\
&\widehat{id} \quad :: a :\!\multimap a \\
&(\hat{\circ}) \quad :: (b :\!\multimap c) \to (a :\!\multimap b) \to (a :\!\multimap c) \\
&\ldots
\end{aligned}
$$

# Interface and denotation

Interface:

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

Model:

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

# Interface and denotation

Interface:

$$
\begin{aligned}
&\textbf{type}\ (:\!\multimap) :: * \to * \to *\\
&scale :: Num\ s \Rightarrow (s :\!\multimap s)\\
&\widehat{id} \quad :: a :\!\multimap a\\
&(\hat{\circ}) \quad :: (b :\!\multimap c) \to (a :\!\multimap b) \to (a :\!\multimap c)\\
&\ldots
\end{aligned}
$$

Model:

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

Specification:

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

# Representation

Start with 1D. Recall partial specification:

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

Try a direct data type representation:

$$\begin{aligned}
&\textbf{data}\ (:\!\multimap) :: * \to * \to *\ \ \textbf{where}\\
&\quad Scale :: Num\ s \Rightarrow s \to (s :\!\multimap s) \quad \text{-- ...}\\
&\mu :: (a :\!\multimap b) \to (a \multimap b)\\
&\mu\ (Scale\ s) = \lambda x \to s \times x
\end{aligned}$$

Spec trivially satisfied by $scale = Scale$.

Others are more interesting.

# Calculate an implementation

Specification:

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

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

Calculation:

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

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

Sufficient definitions:

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

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

# 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 $(:\multimap)$?

# Category

Interface:

$$\begin{aligned}&\textbf{class } \mathit{Category}\ k \textbf{ where}\\ &\quad \mathit{id} \ :: k\ a\ a\\ &\quad (\circ) :: k\ b\ c \to k\ a\ b \to k\ a\ c\end{aligned}$$

Laws:

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

# Linear transformation category

Linear map semantics:

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

Specification as homomorphism (no abstraction leak):

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

Correct-by-construction implementation:

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

$$id = Scale\ 1$$
$$Scale\ s \circ Scale\ s' = Scale\ (s \times s')$$

# Laws for free

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

where equality is *semantic*.

# Laws for free

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

where equality is *semantic*. Proofs:

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

Works for other classes as well.

# Higher dimensions

Interface:

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

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

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

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

Similar to *Arrow* and *ArrowChoice* classes.

# Semantic morphisms

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

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

For $a \multimap b$,

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

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

For calculation, see blog post *Reimagining matrices*.

# Full representation and denotation

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

$$Scale :: Num\ s \Rightarrow s \to (s :\!\multimap s)$$

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

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

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

$$\mu\ (Scale\ s) = \lambda x \to s \times x$$

$$\mu\ (f :\!\vartriangle g) = \lambda a \to (f\ a, g\ a)$$

$$\mu\ (f :\!\triangledown g) = \lambda(a, b) \to f\ a + g\ b$$

# 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:

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

Model:

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

# 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?

# 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

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

Morphism:

$$\begin{aligned} & \mu\ (\mathit{fmap}\ f\ b) \\ \equiv\ & \mathit{fmap}\ f\ (\mu\ b) \\ \\ \equiv\ & f \circ \mu\ b \end{aligned}$$

# Applicative

$$\begin{aligned}&\textbf{instance } \mathit{Applicative}\ ((\rightarrow)\ t)\ \textbf{where}\\ &\quad \mathit{pure}\ a\ \ = \lambda t \rightarrow a\\ &\quad g \mathbin{<\!\!*\!\!>} h = \lambda t \rightarrow (g\ t)\ (h\ t)\end{aligned}$$

Morphisms:

$$\begin{aligned}&\mu\ (\mathit{pure}\ a)\\ &\equiv \mathit{pure}\ a\\ \\ &\equiv \lambda t \rightarrow a\end{aligned}$$

$$\begin{aligned}&\mu\ (fs \mathbin{<\!\!*\!\!>} xs)\\ &\equiv \mu\ fs \mathbin{<\!\!*\!\!>} \mu\ xs\\ \\ &\equiv \lambda t \rightarrow (\mu\ fs\ t)\ (\mu\ xs\ t)\end{aligned}$$

Corresponds exactly to the original FRP denotation.

# Monad

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

$$join\ ff = \lambda t \rightarrow ff\ t\ t$$

Morphism:

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

# Comonad

$$\textbf{class } \mathit{Comonad}\ w \textbf{ where}$$
$$\quad \mathit{coreturn} :: w\ a \to a$$
$$\quad \mathit{cojoin} :: w\ a \to w\ (w\ a)$$

Functions:

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

Suggest a relative time model.

# Image manipulation

# Image manipulation

Central type:

**type** *Image a*

# Image manipulation

Central type:

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

Model:

$$\mu :: \mathit{Image}\ a \to (\mathbb{R}^2 \to a)$$

# Image manipulation

Central type:

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

Model:

$$\mu :: \mathit{Image}\ a \to (\mathbb{R}^2 \to a)$$

As with behaviors,

- Suggests API and semantics (via morphisms).
- Classes: *Monoid*, *Functor*, *Applicative*, *Monad*, *Comonad*.

See Pan page for pictures & papers.

# Memo tries

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

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

This time, $\mu$ has an inverse.

Exploit inverses to calculate instances. Example:

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

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

# Denotational design

# 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*
- Posts on type class morphisms
- This talk