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 $\equiv$ *homomorphism*.

Library design

Goal: precise, elegant, reusable abstractions.

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

Non-leaky abstraction $\equiv$ *homomorphism*.

In Haskell,

Library design

Goal: precise, elegant, reusable abstractions.

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

Non-leaky abstraction $\equiv$ *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:

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

Interface and denotation

Interface:

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

$$\begin{array}{ll} \mathit{scale} :: \mathit{Num}\ s \Rightarrow (s :\!\multimap s) & \\ \widehat{id} & :: a :\!\multimap a \\ (\hat{\circ}) & :: (b :\!\multimap c) \to (a :\!\multimap b) \to (a :\!\multimap c) \\ \ldots & \end{array}$$

Model:

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

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

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:

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

Representation

Start with 1D. Recall partial specification:

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

Try a direct data type representation:

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

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:

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

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

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

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:

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

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

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

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

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

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

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

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:

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

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:

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

Model:

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

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

$$\begin{array}{l}\textbf{instance}\ \mathit{Applicative}\ ((\rightarrow)\ t)\ \textbf{where} \\ \quad \mathit{pure}\ a \ \ = \lambda t \rightarrow a \\ \quad g \circledast h = \lambda t \rightarrow (g\ t)\ (h\ t)\end{array}$$

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

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

Morphism:

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

Comonad

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

Functions:

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

Suggest a relative time model.

Image manipulation

Image manipulation

Central type:

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

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:

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

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, μ has an inverse.

Exploit inverses to calculate instances. Example:

$$\boxed{\begin{array}{c} \mu\ id \equiv id \\ \Leftarrow id \equiv \mu^{-1}\ id \end{array}} \qquad \boxed{\begin{array}{c} \mu\ (g \circ f) \equiv \mu\ g \circ \mu\ f \\ \Leftarrow g \circ f \equiv \mu^{-1}\ (\mu\ g \circ \mu\ f) \end{array}}$$

Denotational design

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