Monads and other abstractions

John Wiegley

22 Jul 2014

Workshop overview

1. Basic math definitions
2. Algebras and laws
3. Working with proofs
4. Category theory & Functors
5. Monads

Mathematics

Meaning

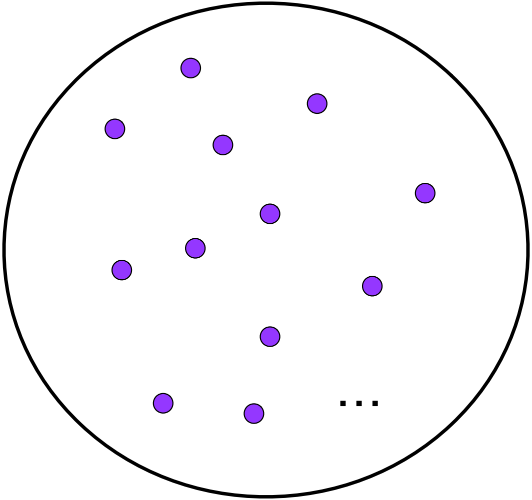
There isn’t any.

Abstraction

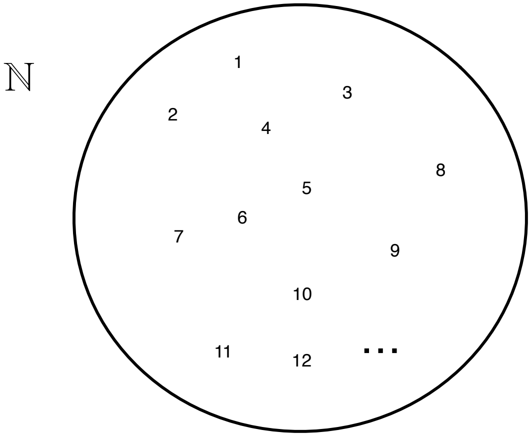
Structures, and relationships between structures.

Sets

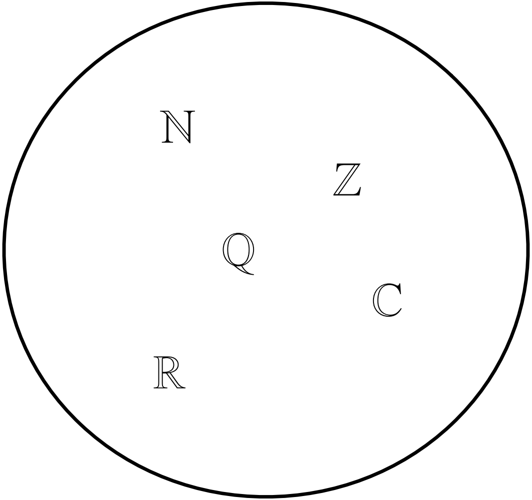
“Stuff”



"Stuff"



Sets of sets



Extensional

Can be defined by stating its elements.

$\{$ *True*, *False* $\}$

Intensional

Or by describing them.

$$\{\ x \mid x \in \mathbb{N}, \mathit{even}(x)\ \}$$

Programmatic

Can be modeled programmatically.

```
type Set a = a -> Bool

import Data.Set as S
type Set a = S.Set a
```

Exercise

Using the functional definition of sets, define union and intersection.

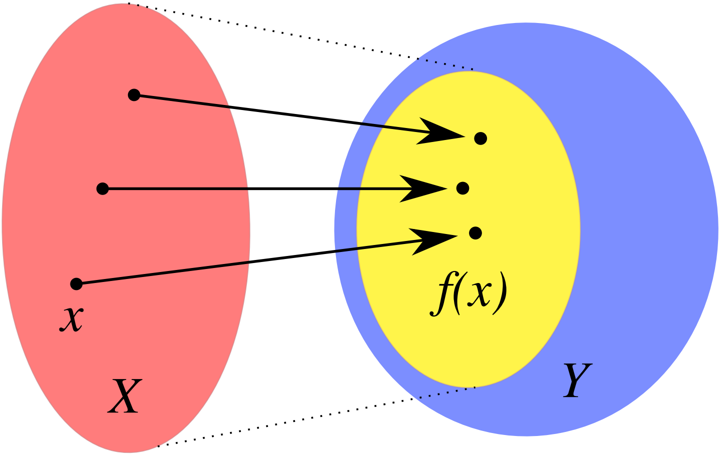
```
type Set a = a -> Bool

union :: Set a -> Set a -> Set a
inter :: Set a -> Set a -> Set a
```

Deceptively simple

With a basic definition and seven axioms (we've seen two!), you can generate a good deal of mathematics.

Functions

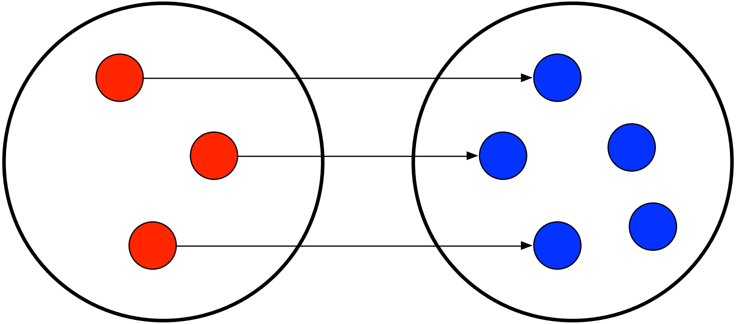
Domain, co-domain, range



$$f : X \to Y$$

Injective

"one to one"



Every **x** maps to a distinct **y**

Injective

$$f : A \to B$$

$$\forall x, y \in A$$

$$f\ x = f\ y \to x = y$$

Injective

Examples of injective things:

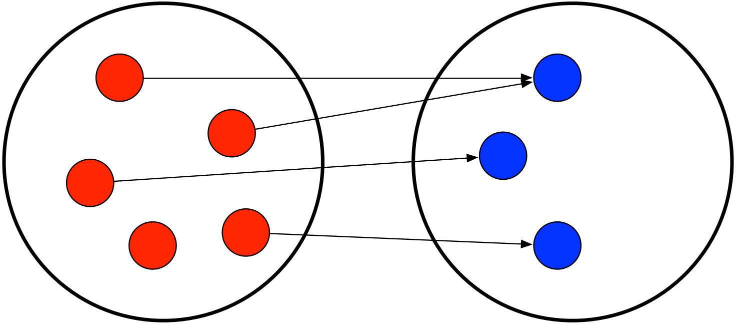
- Data constructors
- Type constructors
- But not type synonyms…

Exercise

1. Write an injective function on `Integer`, and one that is not injective.
2. How do you test it in both cases?

Surjective

"onto"



At least one **x** maps to every **y**

Surjective

$$f : A \to B$$

$$\forall y \in B, \exists x \in A$$

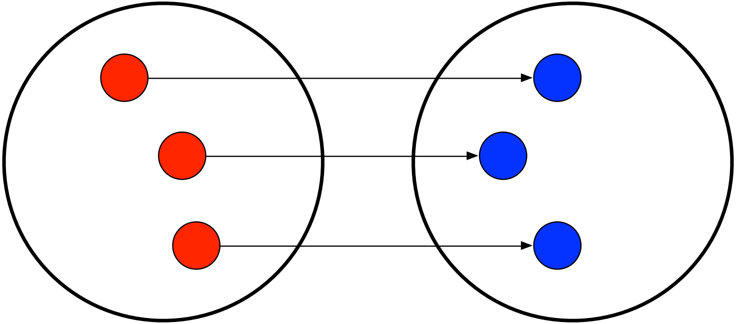
$$f\ x = y$$

Surjective

A function is surjective if the codomain and range are equal. Example:

- `even` is surjective
- `times2` is not

Bijective



One **x** for every **y**

Higher-order functions

Definition (Identity)

$id\ x = x$

Higher-order functions

Definition (Identity)

$$id\ x = x$$

Definition (Composition)

$$(f \circ g)\ x = f(g(x))$$

Properties of functions

$$f : dom \to cod$$

Properties of functions

$$f : dom \rightarrow cod$$

Definition (Idempotent)

$f \circ f = f$

Properties of functions

$$f : dom \to cod$$

Definition (Idempotent)

$f \circ f = f$

Definition (Involutive)

$f \circ f = id$

More properties

Definition (Section)

$f \circ s = id$

Definition (Retract)

$r \circ f = id$

Exercise

For the set of integers, show examples of:

1. idempotency
2. involution
3. section
4. retraction

Isomorphism

An isomorphism is a pair of functions satisfying two equations:

$$f \circ g = id_{cod(f)}$$

$$g \circ f = id_{cod(g)}$$

Isomorphism

In terms of the types involved:

$$A \cong B$$

$$g : A \to B$$

$$f : B \to A$$

Exercise

```haskell
data Unit = Unit
data Maybe a = Nothing | Just a
```

Exercise

```
data Unit = Unit
data Maybe a = Nothing | Just a
```

Write two functions

$$\texttt{toMaybe} :: \texttt{Integer} \to \texttt{Maybe Unit}$$
$$\texttt{fromMaybe} :: \texttt{Maybe Unit} \to \texttt{Integer}$$

Laws

Imposed structure

In the absence of meaning, laws create structure.

Principled restriction

Laws restrict how functions and values relate to each other.

Principled restriction

Laws restrict how functions and values relate to each other.

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
```

Associativity

$$x \bullet (y \bullet z) = (x \bullet y) \bullet z$$

Commutativity

$$x \bullet y = y \bullet x$$

Transitivity

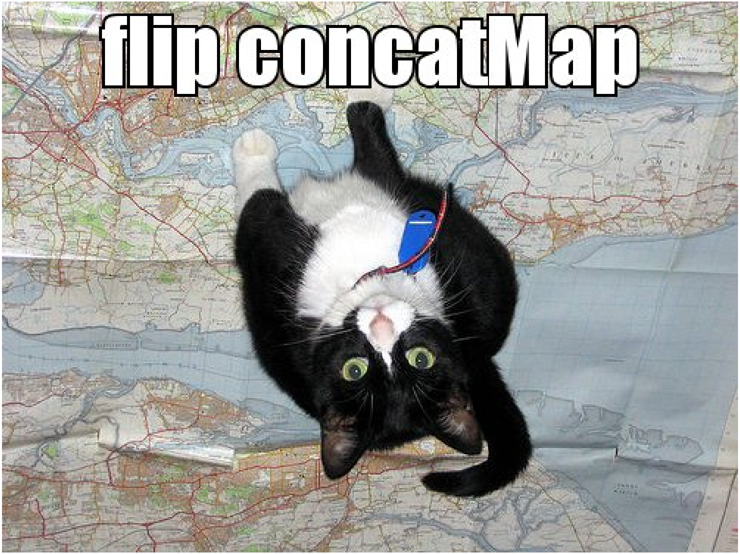
$$x \bullet y \to y \bullet z \to x \bullet z$$

Lawless!

Behold, the face of evil:

```haskell
class Pointed a where
  point :: a
```

[Questions?]



Algebras

Sets with structure

Algebras are basically:

- a set (called the *carrier*)
- functions closed over the set
- laws to govern these functions

Named structures

Some structures recur often enough that it's useful to name them, but the names are arbitrary.

Magma

$$(S, s \to s \to s)$$

The set of laws is empty!

Magma

```haskell
class Magma a where
  binop :: a -> a -> a

instance Magma Integer where
  binop = (+)
```

Semigroup

$$(S, s \to s \to s)$$

Laws:

1. associativity

Semigroup

```
class Semigroup a where
  (<>) :: a -> a -> a
```

Semigroup

```haskell
class Semigroup a where
  (<>) :: a -> a -> a
```

Semigroup law

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

Monoid

$$(S, \varepsilon, s \to s \to s)$$

Laws:

1. left identity
2. right identity
3. associativity

Monoid

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
```

Monoid

```haskell
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
```

Monoid laws

$$\varepsilon \oplus a = a$$

$$a \oplus \varepsilon = a$$

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

Group

$$(S, \varepsilon, s \to s \to s, s \to s)$$

Laws:

1. left identity
2. right identity
3. associativity
4. inverse elements

Group

Group laws

$$\varepsilon \oplus a = a$$
$$a \oplus \varepsilon = a$$
$$a \oplus (b \oplus c) = (a \oplus b) \oplus c$$
$$a \oplus a^{-1} = \varepsilon$$

Homomorphism

“Structure preserving.”

```
floor :: Float -> Int
```

Free objects

What if our monoid, instead of *doing something*, only constructed values?

Free objects

```haskell
data MV a = MEmpty
          | Var a
          | MAppend (MV a) (MV a)

instance Monoid (MV a) where
  mempty  = MEmpty
  mappend = MAppend
```

Building trees

$$(a \oplus b) \oplus c \oplus (d \oplus e)$$

```
(Var a `MAppend` Var b)
   `MAppend`
Var c
   `MAppend`
(Var d `MAppend` Var e)
```

Using the laws

Due to the law of associativity, calls to `mappend` can always be re-associated:

$$a \oplus (b \oplus (c \oplus (d \oplus e)))$$

Using the laws

This changes the expression into something linear, rather than a tree:

```
Var a `MAppend`
 (Var b `MAppend`
  (Var c `MAppend`
   (Var d `MAppend`
    (Var e `MAppend` MEmpty))))
```

Using the laws

Relying on this law, we can simplify the data type:

```
data MV a = MEmpty
          | MAppend a (MV a)
```

Using the laws

Let's rename the constructors to something more familiar:

```
data List a = Nil
            | Cons a (List a)
```

Data structures

Free objects of an algebra become data structures in programming.

Folding

Choosing operations for an algebra is equivalent to folding over its free object.

Evaluators

The two essential aspects of an algebra are:

- Forming expressions
- Evaluating these expressions

Free functors

Every free object is trivially a functor, called a free functor.

```
instance Functor List where
  fmap _ Nil = Nil
  fmap f (Cons x xs)
    = Cons (f x) (fmap f xs)
```

F-algebras

We can encode other algebras using functions and free functors:

```haskell
type Algebra f a = f a -> a

sum :: Algebra List Int
sum Nil = 0
sum (Cons x xs) = x + sum xs
```

Recursion schemes

We won't cover it, but the recursion can be abstracted away for even more generality:

`https://www.fpcomplete.com/user/bartosz/understanding-algebras`

Computational structures

Every free functor can be modeled as a computation rather than a data type:

$$\texttt{List}\ a \cong \forall r, r \to (a \to r \to r) \to r$$

Proving isomorphism

Proof of an isomorphism requires four things:

1. Write a `to` function.
2. Write a `from` function.
3. Show: $\forall x, \mathit{to}\ (\mathit{from}\ x) = x$.
4. Show: $\forall y, \mathit{from}\ (\mathit{to}\ y) = y$.

QuickCheck

In lieu of real proofs, we can sometimes just pick types and use QuickCheck.

Exercise

Prove the following isomorphisms:

$$
\begin{aligned}
\texttt{Identity}\ a &\cong \forall r, (a \to r) \to r \\
\texttt{Maybe}\ a &\cong \forall r, r \to (a \to r) \to r \\
\texttt{Either}\ a\ b &\cong \forall r, (a \to r) \to (b \to r) \to r \\
(a, b) &\cong \forall r, (a \to b \to r) \to r \\
\texttt{List}\ a &\cong \forall r, r \to (a \to r \to r) \to r
\end{aligned}
$$

Exercise

Easy:

- Write `head` for both forms of list.

```
head :: List a -> a
head :: [a] -> a
```

Hard:

- Write `tail` for both forms of list.

```
tail :: List a -> List a
tail :: [a] -> [a]
```

Types are algebras too

$$a + b = \texttt{Either}\ a\ b$$
$$= \texttt{Foo}\ a \mid \texttt{Bar}\ b$$

$$a * b = (a, b)$$
$$= \texttt{Foo}\ a\ b$$

$$b^a = a \to b$$
$$1 = \texttt{Foo}$$
$$0 = \texttt{Void}$$

Which algebra is it?

A near-semiring structure over the set S of types.

1. $(S, +, 0)$ is a monoid
2. $(S, *)$ is a semigroup
3. $\forall a, b, c \in S, (a + b) * c = a * c + b * c$
4. $\forall a \in S, 0 * a = 0$

Example: currying

$$(c^b)^a = c^{ba}$$

```
a -> b -> c   <>   (a, b) -> c
```

Example: lists

$$\begin{aligned}
L(a) &= 1 + a \bullet L(a) \\
&= 1 + a \bullet (1 + a \bullet L(a)) \\
&= 1 + a + a^2 \bullet (1 + a \bullet L(a)) \\
&= 1 + a + a^2 + a^3 \bullet (1 + a \bullet L(a)) \\
&= \cdots \\
&= 1 + a + a^2 + a^3 + a^4 + a^5 + \cdots
\end{aligned}$$

Example: lists

$$
\begin{aligned}
CL(a) &= \forall r, r \to (a \to r \to r) \to r \\
&= \forall r, (r^{(a \to r \to r)})^r \\
&= \forall r, (r^{((r^r)^a)})^r \\
&= \forall r, r^{(r \bullet r^{(r \bullet a)})} \\
&= \forall r, r^{r^{(1 + a \bullet r)}}
\end{aligned}
$$

[Break]

Equational Reasoning

Working with proofs

Equational reasoning gives us a way to reason about pure computations.

Basic format

$$
\begin{aligned}
x &= y \\
&= y' && \text{reason} \\
&= y'' && \text{reason} \\
&= x && \text{reason}
\end{aligned}
$$

Example

$$
\begin{aligned}
f \circ (g \circ h) &= (f \circ g) \circ h \\
&= (\lambda x \to f\ (g\ x)) \circ h && \textbf{unfold } \circ \\
&= \lambda y \to (\lambda x \to f\ (g\ x))\ (h\ y) \\
& && \textbf{unfold } \circ \\
&= \lambda y \to f\ (g\ (h\ y)) && \beta\textbf{-reduction} \\
&= \lambda y \to f\ ((g \circ h)\ y) && \textbf{fold } \circ \\
&= \lambda y \to (f \circ (g \circ h))\ y && \textbf{fold } \circ \\
&= f \circ (g \circ h) && \eta\textbf{-contraction}
\end{aligned}
$$

□

Quantification

Existential

$$\exists x, P\ x$$

Universal

$$\forall x, P\ x$$

Universal

True?

$$\forall x, \exists y \rightarrow x = y$$

Universal

True?

$\forall x, \exists y \rightarrow x \neq y$

Existential

True?

$$\exists y \to \forall x, x = y$$

True?

$$\exists y \to \forall x, x \neq y$$

Relationship

$$\exists x, \varphi(x) \quad \equiv \quad \neg\forall x, \neg\varphi(x)$$

As a game

You can think of quantification like a game between two players, the caller and the callee:

- $\forall$ means the caller gets to decide the object
- $\exists$ means the callee gets to decide

As a game

When we prove, or a write a function, we are the callee. When we call a function or apply a lemma, we are the caller.

Switching roles

It's possible to switch roles inside a function:

$$\forall x, (\forall y, y \to r) \to x \to r$$

Parametricity

Theorems for free!

What does the following type imply (assuming no $\perp$)?

Example (filter)

```
filter :: (a -> Bool) -> [a] -> [a]
filter f xs = _
```

Theorems for free!

Theorem (filter)

$\forall g : a \to b$

$\forall p : a \to \texttt{Bool}$
$\forall q : b \to \texttt{Bool}$

$$\forall x, p\ x = q\ (g\ x) \longrightarrow$$
$$map\ g\ (\textbf{\textit{filter}}\ p\ xs) = \textbf{\textit{filter}}\ q\ (map\ g\ xs)$$

Why is it free?

It’s not just that the type implies the theorem: Writing such a function is a proof of that theorem.

Free theorem generator

Automatically generate free theorems for sub-languages of Haskell:

`http://www-ps.iai.uni-bonn.de/cgi-bin/free-theorems-webui.cgi`

Further reading

"Theorems for free!"

`http://ttic.uchicago.edu/~dreyer/`

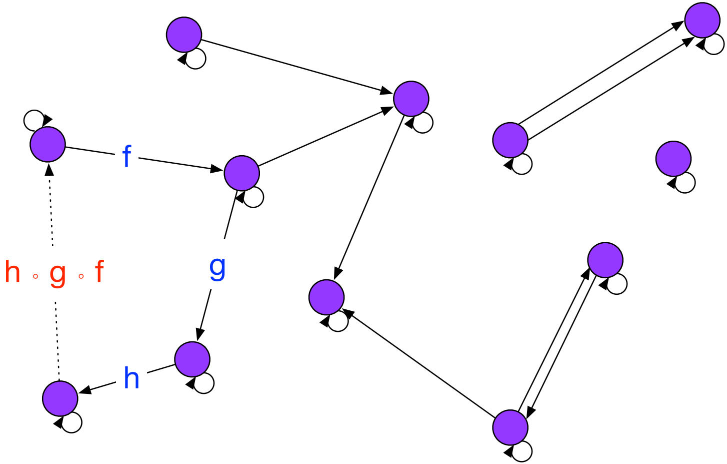
`course/papers/wadler.pdf`

Curry-Howard Isomorphism

Types	Theorems
Values	Proofs

Category Theory

Category



Not all sets

Instead of sets with elements and functions, we have categories with objects and morphisms.

All sets are categories, but not vice-versa.

Example: Any set

Objects	Set elements
Morphisms	Just the identities (a **discrete** category)

Example: Posets

Objects	Set elements
Morphisms	Identities and $\leq$ between some elements
Composition	$(y \leq z) \circ (x \leq y) = (x \leq z)$

Example: Graphs

Objects	Vertices
Morphisms	Edges and self-edges (bidirectional if undirected)
Composition	In the “obvious” way

Example: Set

Objects	Sets
Morphisms	Functions
Composition	As functions do

Example: Mon

Objects	Sets with monoid structure
Morphisms	Monoid homomorphisms
Composition	As functions do

Example: Cat

Objects	Categories
Morphisms	Functors
Composition	As functions do

Example: Fun(C,D)

Objects	Functors $C \to D$
Morphisms	Natural transformations
Composition	As polymorphic functions

Many categories

Any book on category theory will have many more examples of categories than these few.

Some books

- Lawvere, *Conceptual Mathematics: A First Introduction to Categories*
- Awodey, *Category Theory*
- Mac Lane, *Categories for the Working Mathematician*

Other resources

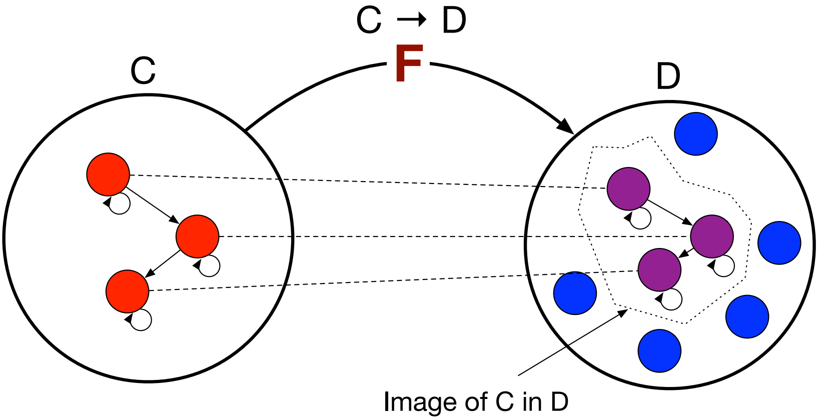
- Catster's videos:
  `http://byorgey.wordpress.com/catsters-guide-2`
- Awodey's presentation on Category Theory at OPLSS 2013
- ##categorytheory on IRC

Concepts transfer

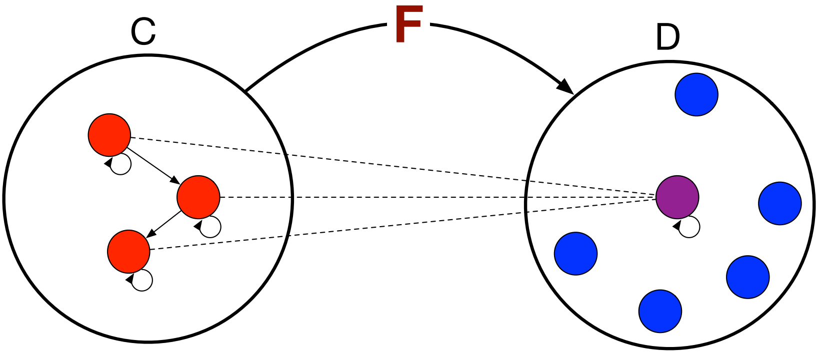
One of the profound concepts in category theory is how ideas transfer.

Functors

Categorical model



Unit mapping



Definition in code

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```

Functor laws

Definition (1. Identity law)

$fmap\ id = id$

Functor laws

Definition (1. Identity law)

$$fmap\ id = id$$

Definition (2. Composition law)

$$fmap\ (f \circ g) = fmap\ f \circ fmap\ g$$

Not containers!

A Functor can sometimes map to:

- a container
- a computation

. . . but a Functor *per se* is neither.

As Context

F `a`

Don’t be fooled

Functors are humble, but powerful.

Origins

Their [Eilenberg and Mac Lane's] goal was to understand natural transformations; in order to do that, functors had to be defined, which required categories.

– Wikipedia

Identity

```
data Identity a = Identity a

instance Functor Identity where
  fmap f (Identity x) = ?
```

Identity

```haskell
data Identity a = Identity a

instance Functor Identity where
  fmap f (Identity x)
    = Identity (f x)
```

Proving Identity Law

$$\mathit{id}\ \mathtt{x} = \mathit{fmap}\ \mathit{id}\ \mathtt{x}$$

$$\begin{aligned}
\mathit{id}\ (\mathtt{Identity}\ \mathtt{x}') &= \mathit{fmap}\ \mathit{id}\ (\mathtt{Identity}\ \mathtt{x}') && \textbf{unfold } \mathtt{x} \\
&= \mathtt{Identity}\ (\mathit{id}\ \mathtt{x}') && \textbf{defn. } \mathtt{fmap} \\
\mathtt{Identity}\ \mathtt{x}' &= \mathtt{Identity}\ \mathtt{x}' && \textbf{defn. } \mathtt{id}
\end{aligned}$$

Proving Composition

$$
\begin{aligned}
& \mathit{fmap}\ (f \circ g)\ \mathtt{x} & \\
=\ & \mathit{fmap}\ (f \circ g)\ (\mathtt{Identity}\ \mathtt{x}') & \textbf{unfold}\ \mathtt{x} \\
=\ & \mathtt{Identity}\ ((f \circ g)\ \mathtt{x}') & \textbf{defn.}\ \mathtt{fmap} \\
=\ & \mathtt{Identity}\ (f(g(\mathtt{x}'))) & \textbf{defn.}\ \circ \\
=\ & \mathit{fmap}\ f\ (\mathtt{Identity}(g(\mathtt{x}'))) & \textbf{defn.}\ \mathtt{fmap} \\
=\ & \mathit{fmap}\ f\ (\mathit{fmap}\ g\ (\mathtt{Identity}\ \mathtt{x}')) & \textbf{defn.}\ \mathtt{fmap} \\
=\ & \mathit{fmap}\ f\ (\mathit{fmap}\ g\ \mathtt{x}) & \textbf{fold}\ \mathtt{x}
\end{aligned}
$$

Maybe

```
data Maybe a = Nothing | Just a

instance Functor Maybe where
  fmap f Nothing  = ?
  fmap f (Just x) = ?
```

Maybe

```
data Maybe a = Nothing | Just a

instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

Either

```
data Left e a = Left e | Right a
```

Tuple

```
data Pair p a = Pair p a
```

Const

```
data Const c a = Const c
```

Exercise

Remember how free functors were encoded as functions? This extends to any functor:

$$f\ a \ \cong\ \forall r, (a \to r) \to f\ r$$

Exercise

$$\texttt{lower} :: (\forall r.\, (a \to r) \to f\ r) \;\to\; f\ a$$

$$\texttt{lift} :: \texttt{Functor}\, f \;\Rightarrow\; f\ a \;\to\; (a \to r) \to f\ r$$

Yoneda lemma

You just proved the Yoneda lemma in a functional language!

Exercise

A Yoneda embedding is itself a Functor:

```
data Yoneda f a
  = Yoneda (forall r. (a -> r) -> f r)

instance Functor (Yoneda f) where
  fmap g (Yoneda k) = ?
```

fmap optimization

One thing Yoneda buys us (among others) is optimization of repeated calls to `fmap`.

Concepts lift

A lot of what we know about values can be lifted to functors.

Lifted Identity

```
data Identity a
  = Identity a

data IdentityF f a
  = IdentityF (f a)
```

Lifted Maybe

```
data Maybe a
  = Nothing | Just a

data MaybeF f a
  = NothingF a | Just (f a)
```

Lifted Either

```haskell
data Either e a
  = Left e | Right a

data EitherF f g a
  = LeftF (f a) | RightF (g a)
```

Lifted Tuple

```
data Tuple a b
  = Tuple a b

data TupleF f g a
  = TupleF (f (g a))
```

Lifted List

```
data List a
  = Nil
  | Cons a (List a)

data ListF f a
  = NilF a
  | ConsF (f (ListF f a))
```

Free Monad

We could rename the constructors of our lifted list:

```
data FreeMonad m a
  = Return a
  | Join (m (FreeMonad m a))
```

As Context

F `a`

As Context

[F] `a`

As Context

[] a

```
Return a
```

As Context

[F, F, F] `a`

```
Join
  (f (Join
      (f (Join
          (f (Return a))))))
```

Famous quote

"A monad is just a monoid in the category of endofunctors, what's the problem?"
– Saunders Mac Lane

Likewise, our free monad is just a free monoid over functors.

[Break]

Applicatives

Definition in code

```
class Functor f
    => Applicative f where
  pure  :: f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Important

One aspect of `Applicative` gives a clue to its power:

The `<*>` operator takes two functorial arguments.

Applicative laws

Definition (1. Identity law)

$pure\ id \otimes v = v$

Applicative laws

Definition (1. Identity law)

$pure\ id \otimes v = v$

Definition (2. Composition law)

$pure\ (\circ) \otimes u \otimes v \otimes w = u \otimes (v \otimes w)$

Applicative laws

Definition (1. Identity law)

$$pure\ id \otimes v = v$$

Definition (2. Composition law)

$$pure\ (\circ) \otimes u \otimes v \otimes w = u \otimes (v \otimes w)$$

Definition (3. Homomorphism law)

$$pure\ f \otimes pure\ x = pure\ (f(x))$$

Applicative laws

Definition (4. Interchange law)

$$u \otimes pure\ y = pure\ (\$\ y) \otimes u$$

Applicative laws

Definition (4. Interchange law)

$$u \otimes \mathit{pure}\ y = \mathit{pure}\ (\$\ y) \otimes u$$

Definition (5. Functor relation law)

$$\mathit{pure}\ f \otimes x = \mathit{fmap}\ f\ x$$

Identity

```
data Identity a = Identity a

instance Applicative Identity where
  pure x = Identity x
  Identity f <*> Identity x = ?
```

Identity

```
data Identity a = Identity a

instance Applicative Identity where
  pure x = Identity x
  Identity f <*> Identity x
    = Identity (f x)
```

Proving Identity

$$
\begin{aligned}
& \textit{pure id} \otimes \texttt{v} & \\
={} & \textit{pure id} \otimes \texttt{Identity v} & \textbf{unfold } \texttt{v} \\
={} & \texttt{Identity } \textit{id} \otimes \texttt{Identity v} & \textbf{defn. } \texttt{pure} \\
={} & \texttt{Identity } (\textit{id } \texttt{v}) & \textbf{defn. } \otimes \\
={} & \texttt{Identity v} & \textbf{defn. } \texttt{id} \\
={} & \texttt{v} & \textbf{fold } \texttt{v}
\end{aligned}
$$

Proving Homomorphism

$$
\begin{aligned}
& \mathit{pure}\ f \otimes \mathit{pure}\ x & \\
= {} & \texttt{Identity}\ f \otimes \texttt{Identity}\ x & \textbf{defn. } \texttt{pure} \\
= {} & \texttt{Identity}\ (f(x)) & \textbf{defn. } \otimes \\
= {} & \mathit{pure}\ (f(x)) & \textbf{defn. } \texttt{pure}
\end{aligned}
$$

Maybe

```
data Maybe a = Nothing | Just a

instance Applicative Maybe where
  pure x = ?

  Nothing <*> Nothing = ?
  Just f  <*> Nothing = ?
  Nothing <*> Just x  = ?
  Just f  <*> Just x  = ?
```

Maybe

```
data Maybe a = Nothing | Just a

instance Applicative Maybe where
  pure x = ?

  Just f <*> Just x = Just (f x)
  _      <*> _      = Nothing
```

Either

```
data Left e a = Left e | Right a
```

Tuple

```
data Pair p a = Pair p a
```

Const

Const requires a trickier instance.

```
data Const c a = Const c

instance Monoid c
    => Applicative (Const c) where
  pure x = ?
  Const a <*> Const b = ?
```

Analysis

Applicatives allow for expression analysis.

Example: Minimizing expensive key lookups.

Composition

Another useful trait is that applicatives compose well.

`http://comonad.com/reader/2012/abstracting-with-applicatives`

Monads

Definition in code

```
class Monad m where
  return :: m a
  (>>=)  :: m a -> (a -> m b) ->
            m b
```

Definition in code (join)

```
class Monad m where
  return :: m a
  join   :: m (m a) -> m a
```

Bind in terms of join

```
m >>= f  =  join (fmap f m)
```

How bind works

```
              m   :: m a
            f     ::   a -> m b
      fmap f      :: m a -> m (m b)
      fmap f m    :: m (m b)
join (fmap f m)   :: m b
```

Not fmap

Bind differs from fmap in that a new `m` was created.

Monad laws

Definition (1. Left identity law)

$\texttt{return}\ a \gg= f = f\ a$

Monad laws

Definition (1. Left identity law)

$$\texttt{return}\ a \gg= f = f\ a$$

Definition (2. Right identity Law)

$$m \gg= \texttt{return} = m$$

Monad laws

Definition (1. Left identity law)

$$\texttt{return}\ a \gg\!= f = f\ a$$

Definition (2. Right identity Law)

$$m \gg\!= \texttt{return} = m$$

Definition (3. Associativity Law)

$$(m \gg\!= f) \gg\!= g = m \gg\!= \lambda x \to f\ x \gg\!= g$$

Monadic composition

```
f >=> g  =  \x -> f x >>= g

g <=< f  =  \x -> g =<< f x
```

Monad laws (form 2)

Definition (1. Left identity law (alt))

`return` `>=>` $f = f$

Monad laws (form 2)

Definition (1. Left identity law (alt))

`return >=>` $f = f$

Definition (2. Right identity Law (alt))

f `>=> return` $= f$

Monad laws (form 2)

Definition (1. Left identity law (alt))

`return >=>` $f = f$

Definition (2. Right identity Law (alt))

f `>=> return` $= f$

Definition (3. Associativity Law (alt))

$(f \mathbin{>=>} g) \mathbin{>=>} h = f \mathbin{>=>} (g \mathbin{>=>} h)$

Chaining

Monads allow us to chain computations.

```
x >>= f >>= g >>= >>= h
```

Chaining

Haskell has a special notation for this:

```
do a <- x
   b <- f a
   c <- g b
   h c
```

Desugared

Which is just sugar for:

```
(x     >>= \a ->
 f  a  >>= \b ->
 g  b  >>= \c ->
 h  c)
```

Thinking about join

When implementing a new Monad, ask yourself: What does it mean to "multiply" two contexts?

Identity

```
data Identity a = Identity a

instance Functor Identity where
  Identity m >>= f = ?
```

Identity

```
data Identity a = Identity a

instance Functor Identity where
  Identity m >>= f = f m
```

Proving Left Identity

$$
\begin{aligned}
& \textit{return}\ \texttt{a} >\!\!>\!= f & \\
=\ & \texttt{Identity}\ \texttt{a} >\!\!>\!= f & \textbf{defn. } \texttt{return} \\
=\ & f\ \texttt{a} & \textbf{defn. } \text{»=}
\end{aligned}
$$

Proving Right Identity

$$
\begin{aligned}
& \mathtt{m} >>= \textit{return} \\
= \; & \mathtt{Identity}\, \mathtt{m}' >>= \textit{return} && \textbf{unfold } \mathtt{m} \\
= \; & \textit{return}\, \mathtt{m}' && \textbf{defn. } \text{»=} \\
= \; & \mathtt{Identity}\, \mathtt{m}' && \textbf{defn. } \mathtt{return} \\
= \; & \mathtt{m} && \textbf{fold } \mathtt{m}
\end{aligned}
$$

Proving Associativity

$$
\begin{aligned}
& (m >>= f) >>= g \\
= {} & (\texttt{Identity}\,\texttt{m}' >>= f) >>= g && \textbf{unfold } \texttt{m} \\
= {} & f\ \texttt{m}' >>= g && \textbf{defn. } \text{»=} \\
= {} & (\lambda x \to f\ x >>= g)\ \texttt{m}' && \eta\textbf{-expansion} \\
= {} & \texttt{Identity}\,\texttt{m}' >>= (\lambda x \to f\ x >>= g) && \textbf{defn. } \text{»=} \\
= {} & m >>= (\lambda x \to f\ x >>= g) && \textbf{fold } \texttt{m}
\end{aligned}
$$

Maybe

```
data Maybe a = Nothing | Just a

instance Functor Maybe where
  Nothing >>= f = ?
  Just x  >>= f = ?
```

Maybe

```
data Maybe a = Nothing | Just a

instance Functor Maybe where
  Nothing >>= f = Nothing
  Just x  >>= f = Just (f x)
```

Maybe

`Maybe` gives us a way of short-circuiting computations.

Either

```
data Left e a = Left e | Right a
```

Either

`Either` lets us short-circuit with an alternate result.

Tuple

What additional constraint is needed to make this a Monad, and why?

```haskell
data Pair w a = Pair w a
```

Tuple

Tuples form the Writer monad, if we write one more function:

```
tell :: Monoid w => w -> Pair w ()
tell w = ?
```

Exercise: Const

Why can't it be a monad?

```
data Const c a = Const c

instance Monoid c
    => Monad (Const c) where
  return = pure
  Const c >>= f = ?
```

Reader

Functions are functors, applicatives and monads too. As a monad, it's often called `Reader`.

```
instance Monad ((->) a) where
  return x = ?
  k >>= f  = ?
```

State

`State` is another classic monad.

```
newtype State s a
  = State { runS :: s -> (a, s) }

instance Monad (State s) where
  return x = ?
  k >>= f  = ?
```

State

`State` is made useful by two more functions, `get` and `put`.

```
get :: State s s
get = ?

put :: s -> State s ()
put s = ?
```

Silly example

```
let f x = when (even x) $ do
          y <- get
          put (x + y)
in runS (mapM_ f [1,2,3,4,5]) 0
```

Free Monads

With free monads, we can defer the choice of return and bind, allowing us to model computations.


THIS MEETING IS OVER.