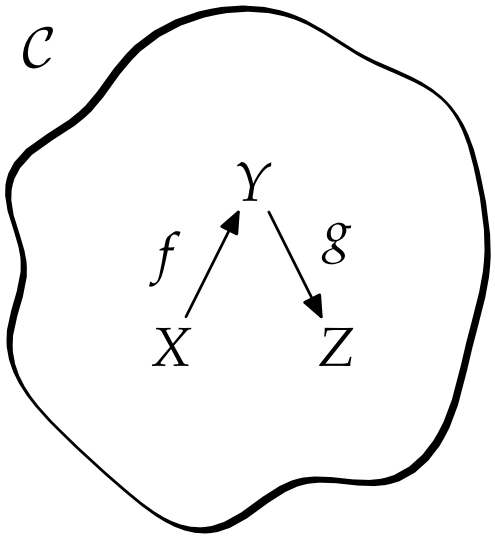
Abstract Nonsense for Functional Programmers

Edsko de Vries

January 9, 2009

Introduction

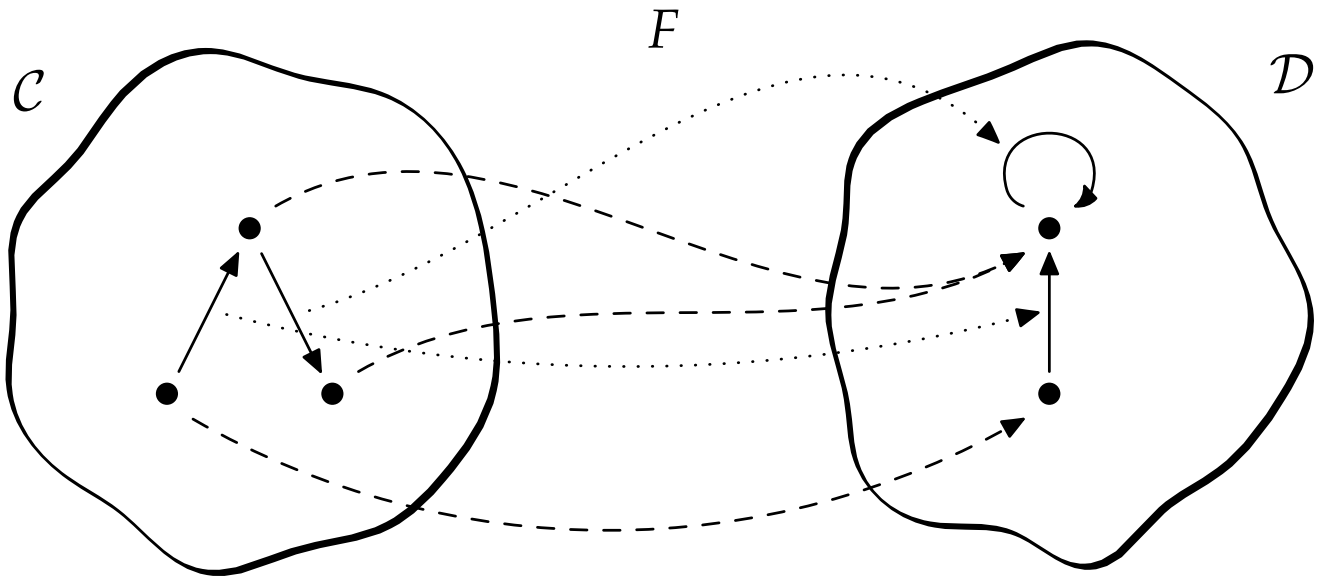
A category $\mathcal{C}$ consists of a set of objects and morphisms between objects.



Example: category **Hask** of Haskell types and programs.

Functors

A functor F is a morphism in the category **Cat** of categories.




```
class Functor f :: * → * where
  fmap :: (a → b) → f a → f b
```

Functor Laws

In addition, a functor must satisfy a number of laws:

- $F(f : A \to B) : F(A) \to F(B)$
- $F(g \circ f) = Fg \circ Ff$
- $F(1_A) = 1_{FA}$

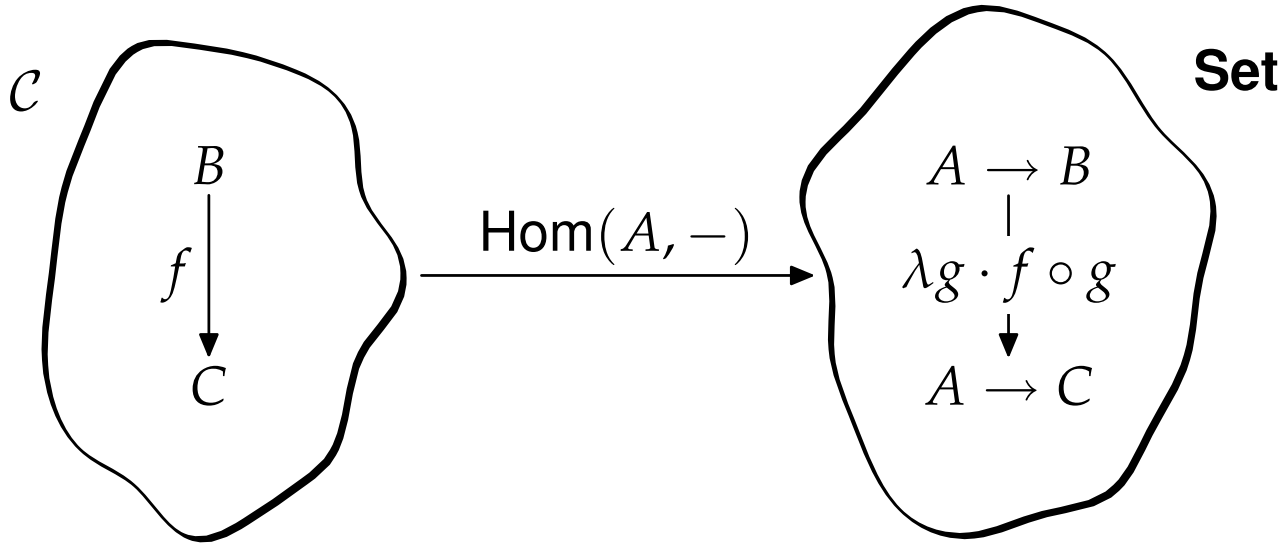
In Haskell:

```
fmap (f :: a → b) :: f a → f b
fmap (g . f) = fmap g . fmap f
fmap id = id
```

Question: the functor laws are *sufficient* to guarantee that **Cat** is a category, but are they *necessary*?

Hom Functors

Pick a category $\mathcal{C}$ and an object $A : \mathcal{C}$.
Then $\mathrm{Hom}(A, -) : \mathcal{C} \to \mathbf{Set}$ is a functor:



A functor F is **represented** by an object $A : \mathcal{C}$ if F is naturally isomorphic to $\mathrm{Hom}(A, -)$.

Algebras

An algebra is a set A together with some operations that return values in that set. Examples:

- $(\mathbb{N}, 0 :: \mathbb{1} \to \mathbb{N}, (+) :: \mathbb{N} \times \mathbb{N} \to \mathbb{N})$
- $(\mathbb{B}, \mathtt{true} :: \mathbb{1} \to \mathbb{B}, (\equiv) :: \mathbb{B} \times \mathbb{B} \to \mathbb{B})$
- $(\mathbb{B}, \mathtt{false} :: \mathbb{1} \to \mathbb{B}, (\vee) :: \mathbb{B} \times \mathbb{B} \to \mathbb{B})$

Alternatively:

- $(\mathbb{N}, \mathtt{in} :: \mathbb{1} + \mathbb{N} \times \mathbb{N} \to \mathbb{N})$
- $(\mathbb{B}, \mathtt{in} :: \mathbb{1} + \mathbb{B} \times \mathbb{B} \to \mathbb{B})$
- $(\mathbb{B}, \mathtt{in} :: \mathbb{1} + \mathbb{B} \times \mathbb{B} \to \mathbb{B})$

Or, letting $F\ X = \mathbb{1} + X \times X$,

- $(\mathbb{N}, \mathtt{in} :: F\ \mathbb{N} \to \mathbb{N})$
- $(\mathbb{B}, \mathtt{in} :: F\ \mathbb{B} \to \mathbb{B})$
- $(\mathbb{B}, \mathtt{in} :: F\ \mathbb{B} \to \mathbb{B})$

Homomorphisms

A homomorphism is a structure preserving map between F algebras:

$$\begin{array}{ccc} FA & \xrightarrow{\mathtt{in}_A} & A \\ {\scriptstyle Fh}\downarrow & & \downarrow{\scriptstyle h} \\ FB & \xrightarrow[\mathtt{in}_B]{} & B \end{array}$$

Take $F\,X = \mathbb{1} + X \times X$. Consider $f : (\mathbb{N}^*, \epsilon, (+\!\!+)) \to (\mathbb{N}, 0, (+))$.

$$\begin{array}{ccc} \mathbb{1} + \mathbb{N}^* \times \mathbb{N}^* & \xrightarrow{\mathtt{in}} & \mathbb{N}^* \\ {\scriptstyle \mathbb{1}+f\times f}\downarrow & & \downarrow{\scriptstyle f} \\ \mathbb{1} + \mathbb{N} \times \mathbb{N} & \xrightarrow[\mathtt{in}]{} & \mathbb{N} \end{array}$$

```
f []             = 0
f (xs ++ ys) = f xs + f ys
```

Underspecified: both `length` and `sum` satisfy the equations.

Initial algebras

However, when we take $F\ X = \mathbb{1} + \mathbb{N} \times X$ and then consider $f : (\mathbb{N}^*, \epsilon, (:)) \to (\mathbb{N}, 0, (+))$, we get

$$\begin{array}{ccc} \mathbb{1} + \mathbb{N} \times \mathbb{N}^* & \xrightarrow{\texttt{in}} & \mathbb{N}^* \\ {\scriptstyle \mathbb{1}+\mathbb{N}\times f}\downarrow & & \downarrow{\scriptstyle f} \\ \mathbb{1} + \mathbb{N} \times \mathbb{N} & \xrightarrow[\texttt{in}]{} & \mathbb{N} \end{array}$$

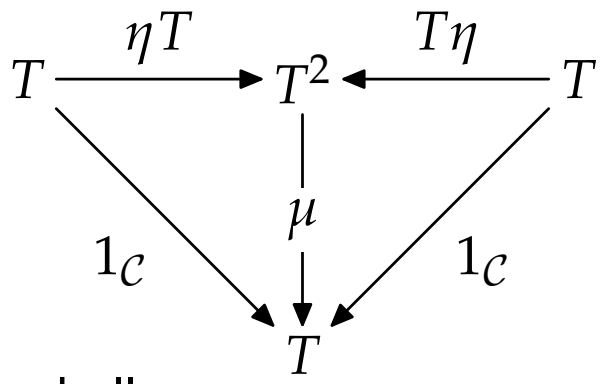
In other words,

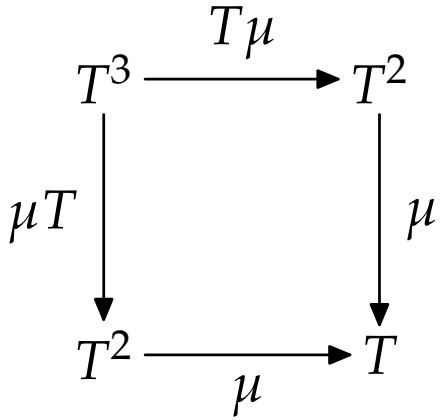
```
f []       = 0
f (n:ns)   = n + f ns
```

(Inductive) datatypes correspond to **initial algebras**; the corresponding homomorphisms are called **catamorphisms** or **folds**.

Monads

A monad on a category $\mathcal{C}$ is defined to be a triple (T, η, μ) of a functor $T : \mathcal{C} \to \mathcal{C}$ and two natural transformation $\eta : 1_{\mathcal{C}} \to \mathcal{C}$ and $\mu : T^2 \to T$:





In Haskell:

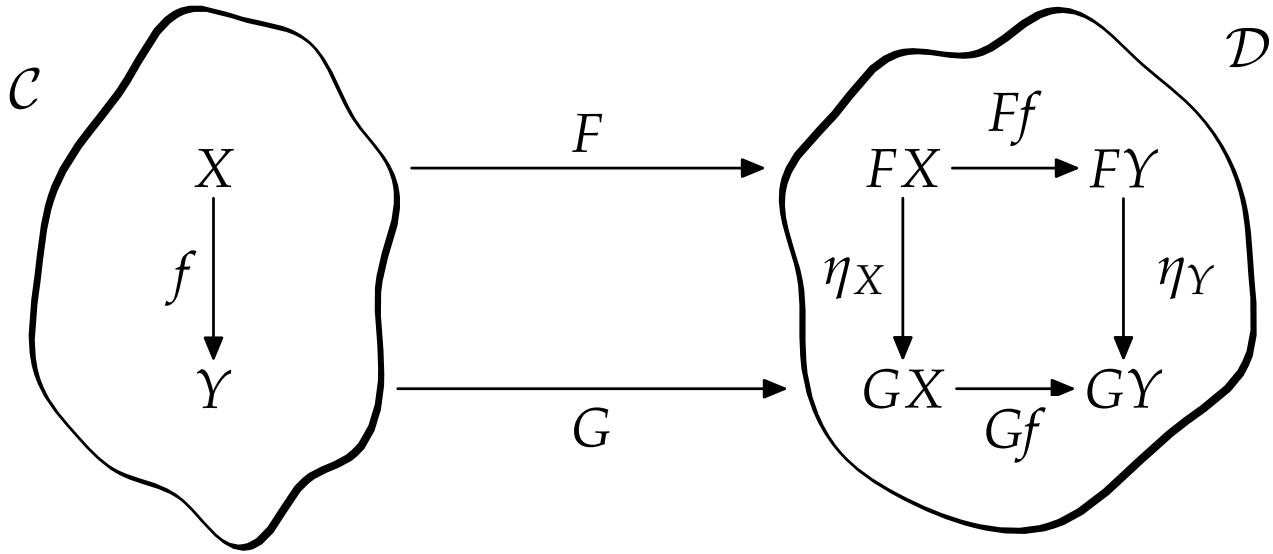
```
class Monad (t :: * → *) where
  return :: ∀ a. a → t a
  join   :: ∀ a. (t (t a)) → t a

join . return = id = join . fmap return
join . join = join . fmap join

(>>=) :: m a → (a → m b) → m b
x >>= f = join (fmap f x)
```

Partiality Monad

(removed)

Natural transformations

A natural transformation η is a morphism in the functor category **Fun**$(\mathcal{C}, \mathcal{D})$ of functors between $\mathcal{C}$ and $\mathcal{D}$.



In Haskell:

```
f      :: X → Y
F, G   :: * → *
```

```
η . fmap f = fmap f . η
```
(Theorem for free!)

Yoneda Lemma

Pick an object A in a category $\mathcal{C}$, and let

$$h_A = \mathrm{Hom}(A, -)$$

Let F be an arbitrary functor from $\mathcal{C}$ to **Set**. Then

Lemma (Yoneda)

$$\mathrm{Nat}(h_A, F) \cong F(A)$$

Naturality square for $\eta : \mathrm{Nat}(h_A, F)$:

$$\begin{array}{ccc} h_A X & \xrightarrow{h_A f} & h_A Y \\ {\scriptstyle \eta_X}\downarrow & & \downarrow{\scriptstyle \eta_Y} \\ FX & \xrightarrow[Ff]{} & FY \end{array}$$

In Haskell:

```
η :: ∀b. (a → b) → f b
```

Witness (in Haskell)

Recall

Lemma (Yoneda)

$\mathrm{Nat}(h_A, F) \cong F(A)$

In Haskell:

```
check   :: Functor f ⇒ f a → (∀ b. (a → b) → f b)
uncheck :: Functor f ⇒ (∀ b. (a → b) → f b) → f a
```

Identity functor

Lemma (Yoneda)

The following functions are mutually inverse.

```
check   :: Functor f ⇒ f a → (∀ b. (a → b) → f b)
uncheck :: Functor f ⇒ (∀ b. (a → b) → f b) → f a
```

Specific instance: take $f = id$.

```
check   :: a → (∀ b. (a → b) → b)
check a f = f a

uncheck :: (∀ b. (a → b) → b) → a
uncheck t = ?
```

List functor

Lemma (Yoneda)

The following functions are mutually inverse.

```
check   :: Functor f ⇒ f a → (∀ b. (a → b) → f b)
uncheck :: Functor f ⇒ (∀ b. (a → b) → f b) → f a
```

Specific instance: take $f = []$.

```
check   :: [a] → (∀ b. (a → b) → [b])
check f as = map f as

uncheck :: (∀ b. (a → b) → [b]) → [a]
uncheck t = ?
```

Hom functor

Lemma (Yoneda)

The following functions are mutually inverse.

```
check   :: Functor f ⇒ f a → (∀ b. (a → b) → f b)
uncheck :: Functor f ⇒ (∀ b. (a → b) → f b) → f a
```

Specific instance: take $f = \mathrm{Hom}(C, -)$.

```
check   :: (c → a) → (∀ b. (a → b) → (c → b))
check a f = f . a

uncheck :: (∀ b. (a → b) → (c → b)) → (c → a)
uncheck t = ?
```

General case

Lemma (Yoneda)

The following functions are mutually inverse.

```
check   :: Functor f ⇒ f a → (∀ b. (a → b) → f b)
check a f = fmap f a

uncheck :: Functor f ⇒ (∀ b. (a → b) → f b) → f a
uncheck t = t id
```

Proof.

```
check (uncheck f) a
   = check (f id) a
   = fmap a (f id)
   = f (fmap a id)
   = f (a . id)
   = f a
```

```
uncheck (check f)
   = (check f) id
   = fmap id f
   = id f
   = f
```

□

Corollary

Lemma (Barr and Wells, 4.5.4)

*If $F : \mathcal{C} \to$ **Set** is represented by both C and D, then $C \cong D$.*

In other words, if the set of functions from C is isomorphic to the set of functions from D, then C and D are isomorphic.

Justifies treating objects in a category as “black boxes”.

References

General category theory

- “Category Theory”, *Steve Awodey*
- “Category Theory for Computing Science” (3rd edition), *Michael Barr and Charles Wells*

Algebra

- “Generic Programming—An Introduction”, *Roland Backhouse et al.*

Yoneda lemma in Haskell

- “Reverse Engineering Machines with the Yoneda Lemma”, *Dan Piponi*