All About Comonads (Part 1)

An incomprehensible guide to the theory and practice of comonadic programming in Haskell

Edward Kmett

http://comonad.com/

Categories

- Categories have objects and arrows
- Every object has an identity arrow
- Arrow composition is associative

Categories

- Categories have objects and arrows
- Every object has an identity arrow
- Arrow composition is associative

- **Hask** is a category with types as objects and functions between those types as arrows.

Categories in Haskell

- In Control.Category (GHC 6.10+):

```
import Prelude hiding (id,(.))

class Category (⇀) where
   id :: a ⇀ a
   (.) :: (b ⇀ c) → (a ⇀ b) → (a ⇀ c)
```

Categories in Haskell

```
import Prelude hiding (id,(.))

class Category (⇀) where
    id :: a ⇀ a
    (.) :: (b ⇀ c) ⟶ (a ⇀ b) ⟶ (a ⇀ c)

instance Category (⟶) where
    id x = x
    (f . g) x = f (g x)
```

Categories in Haskell

```
class Category (⇀) where
  id :: a ⇀ a
  (.) :: (b ⇀ c) → (a ⇀ b) → (a ⇀ c)
```

The dual category C^{op} of a category C has arrows in the opposite direction.

```
data Dual k a b = Dual (k b a)

instance Category (⇀) => Category (Dual (⇀)) where
  id = Dual id
  Dual f . Dual g = Dual (g . f)
```

Functors

- Let C and D be categories
- A functor F from C to D maps
  - objects of C onto objects of D
  - arrows of C onto arrows of D

while preserving the identity morphisms and composition of morphisms:

$F\ (id_X) = id_{F(X)}$

$F\ (g\ .\ F) = F\ g\ .\ F\ f$

Functors in Haskell

class Functor f where

  fmap :: (a ⟶ b) ⟶ f a ⟶ f b

requiring the following two laws:

1.) fmap id = id

2.) fmap (g . f) = fmap g . fmap f

Note that (2) above follows as a free theorem from the type of fmap, so you only need to check (1)!

Functors in Haskell

type (⇀) = (⟶)

class Functor f where

fmap :: (a ⇀ b) ⟶ (f a ⇀ f b)

requiring the following two laws:

1.) fmap id = id

2.) fmap (g . f) = fmap g . fmap f

Note that (2) above follows as a free theorem from the type of fmap, so you only need to check (1)!

Functors in Haskell

type (⇀) = (⟶)

class Functor f where

    fmap :: (a ⇀ b) ⟶ (f a ⇀ f b)

- Ignores the object mapping and focuses on arrows

Cofunctor = Functor

type (⇀) = (⟶)

class Functor f where

  fmap :: (a ⇀ b) ⟶ (f a ⇀ f b)

class Cofunctor f where

  cofmap :: (b ⇀ a) ⟶ (f b ⇀ f a)

It’s the same thing!

Cofunctor /= ContravariantFunctor

```
type (⇀) = (⟶)

class Functor f where

  fmap :: (a ⇀ b) ⟶ (f a ⇀ f b)

class Cofunctor f where

  cofmap :: (b ⇀ a) ⟶ (f b ⇀ f a)

class ContravariantFunctor f where

  contrafmap :: (b ⇀ a) ⟶ (f a ⇀ f b)
```

Nothing said arrows had to point the same way!

Example: Contravariant Functor

```
class ContravariantFunctor f where
    contrafmap :: (b ⟶ a) ⟶ f a ⟶ f b

newtype Test a = Test { runTest :: a -> Bool }
instance ContravariantFunctor Test where
    contrafmap f (Test g) = Test (g . f)

isZero :: Test Int
isZero = Test (==0)

isEmpty :: Test [a]
isEmpty = contrafmap length isZero

result :: Bool
result = runTest isEmpty “Hello”
```

Functors in Haskell Redux

type (⇀) = (⟶)

class Functor f where

  fmap :: (a ⇀ b) ⟶ (f a ⇀ f b)

- Ignores the object mapping and focuses on arrows
- Models only covariant **Hask** endofunctors!

Functors in category-extras

```
class Functor f where
    fmap :: (a ⟶ b) ⟶ f a ⟶ f b
```

```
class (Category (⇀), Category (⇁)) =>
    Functor' f (⇀) (⇁) | f (⇀) ⟶ (⇁), f (⇁) ⟶ (⇀) where
    fmap' :: (a ⇀ b) ⟶ (f a ⇁ f b)
```

- Now contravariant endofunctors from C are just functors from C^{op}.

See Control.Functor.Categorical

Functors in category-extras

- As an aside if you prefer type families…

```
class (Category (Dom f),  Category (Cod f)) => Functor’ f
  where
  type Dom f :: * ⟶ * ⟶ *
  type Cod f :: * ⟶ * ⟶ *
  fmap’ :: Dom f a b ⟶ Cod f (f a) (f b)
```

Monads in Haskell

class Monad m where

    return :: a ⟶ m a

    (>>=) :: m a ⟶ (a ⟶ m b) ⟶ m b

and some laws we'll revisit later:

1.) return a >>= f = f a

2.) m >>= return = m

3.) (m >>= f) >>= g = m >>= (\x -> f x >>= g)

Monads in Haskell

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

Seems rather object-centric!

Monads in Haskell

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

type (⇀) = (⟶)
class Monad' m where
    return :: a ⇀ m a
    (>>=) :: m a ⟶ (a ⇀ m b) ⟶ m b
```

Monads in Haskell

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

type (⇀) = (⟶)
class Monad’ m where
    return :: a ⇀ m a
    (=<<) :: (a ⇀ m b) ⟶ m a ⟶ m b
```

Monads in Haskell

```
class Monad m where

    return :: a ⟶ m a

    (>>=) :: m a ⟶ (a ⟶ m b) ⟶ m b


type (⇀) = (⟶)
class Monad' m where

    return :: a ⇀ m a

    (=<<) :: (a ⇀ m b) ⟶ (m a ⇀ m b)
```

Monads in Haskell

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

class Category (⇀) => Monad’ m (⇀) where
  return :: a ⇀ m a
  (=<<) :: (a ⇀ m b) ⟶ (m a ⇀ m b)
```

Monads in Haskell

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

```
class Category (⇀) => Monad' m (⇀) where
    return :: a ⇀ m a
    bind :: (a ⇀ m b) ⟶ (m a ⇀ m b)
```

Now we're only talking about arrows!

The original Haskell definition required a category with 'Exponentials.' This definition does not.

Monads in Haskell

```
class Functor m => Monad m where
    return :: a ⟶ m a
    bind :: (a ⟶ m b) ⟶ (m a ⟶ m b)

class Functor' m (⇀) (⟶) => Monad' m (⇀) where
    return :: a ⇀ m a
    bind :: (a ⇀ m b) ⟶ (m a ⇀ m b)
```

See Control.Monad.Categorical

Monad laws revisited

```
class Functor m => Monad m where
    return :: a ⟶ m a
    bind :: (a ⟶ m b) ⟶ (m a ⟶ m b)
```

So in this terminology the monad laws are:

1.) bind return = id

2.) bind f . return = f

3.) bind f . bind g = bind (bind g . f)

So Why the Fuss?

- A comonad over C is a monad over C^{op}.
- So we want to be able to turn the arrows around. (>>=) was muddling our thinking by mixing arrows from **Hask** and "exponentials" from the category in question.

Comonads in Haskell

```
type (⇀) = (⟶)
class Functor m => Monad m where
    return :: a ⇀ m a
    bind :: (a ⇀ m b) ⟶ (m a ⇀ m b)

class Functor m => Comonad m where
    coreturn :: m a ⇀ a
    cobind :: (m b ⇀ a) ⟶ (m b ⇀ m a)
```

Comonads in Haskell

```
type (⇀) = (⟶)
class Functor m => Monad m where
    return :: a ⇀ m a
    bind :: (a ⇀ m b) ⟶ (m a ⇀ m b)

class Functor m => Comonad m where
    coreturn :: m a ⇀ a
    cobind :: (m b ⇀ a) ⟶ (m b ⇀ m a)
```

So Functor = Cofunctor, **but** Monad /= Comonad.

Comonads in Haskell

```
type (⇀) = (⟶)
class Functor m => Monad m where

    return :: a ⇀ m a

    bind :: (a ⇀ m b) ⟶ (m a ⇀ m b)


class Functor w => Comonad w where

    coreturn :: w a ⇀ a

    cobind :: (w b ⇀ a) ⟶ (w b ⇀ w a)
```

Comonads in Haskell

```
type (⇀) = (⟶)
class Functor m => Monad m where
    return :: a ⇀ m a
    bind :: (a ⇀ m b) ⟶ (m a ⇀ m b)

class Functor w => Comonad w where
    coreturn :: w a ⇀ a
    cobind :: (w a ⇀ b) ⟶ (w a ⇀ w b)
```

Comonads in Haskell

```
type (⇀) = (⟶)
class Functor m => Monad m where
    return :: a ⇀ m a
    bind :: (a ⇀ m b) ⟶ (m a ⇀ m b)

class Functor w => Comonad w where
    extract :: w a ⇀ a
    extend :: (w a ⇀ b) ⟶ (w a ⇀ w b)
```

Comonads in Haskell

```
class Functor w => Comonad w where
    extract :: w a ⟶ a
    extend :: (w a ⟶ b) ⟶ (w a ⟶ w b)
```

With 3 laws

1.) extend extract = id

2.) extract . extend f = f

3.) extend f . extend g = extend (f . extend g)

Monad Join and Bind

join :: Monad m => m (m a) ⟶ m a

join = bind id

bind :: Monad m => (a ⟶ m b) ⟶ (m a ⟶ m b)

bind f = join . fmap f

So, we can define a monad with either

1.) return, join and fmap

2.) return and bind.

Comonad Duplicate and Extend

```
duplicate :: Comonad w => w a → w (w a)
duplicate = extend id
```

```
extend :: Comonad w => (w a → b) → (w a → w b)
extend f = fmap f . duplicate
```

We can define a comonad with either

1.) extract, duplicate and fmap

2.) extract and extend

Exercise: The Product Comonad

Given:

```
data Product e a = Product e a
class Functor w => Comonad w where
  extract :: w a -> a
  extend :: (w a -> b) -> w a -> w b
  extend f = fmap f . duplicate

  duplicate :: w a -> w (w a)
  duplicate = extend id
```

Derive:

instance Functor (Product e) – or instance Functor ((,)e)

instance Comonad (Product e) – or instance Comonad ((,)e)