Comonads: what are they and what can you do with them?

Melbourne Haskell Users Group

David Overton

29 May 2014

Table of Contents

Motivation

- Monads are an abstract concept from category theory, have turned out to be surprisingly useful in functional programming.
- Category theory also says that there exists a dual concept called *comonads*. Can they be useful too?
- Intuition:
    - Monads abstract the notion of *effectful computation of a value*.
    - Comonads abstract the notion of *a value in a context*.
    - "Whenever you see large datastructures pieced together from lots of small but similar computations there's a good chance that we're dealing with a comonad."
    —Dan Piponi

Table of Contents

What is a Comonad?

- A comonad is just a comonoid in the category of endofunctors...
- A comonad is the category theoretic *dual* of a monad.
- A comonad is a monad with the "arrows" reversed.

Both monads and comonads are functors. (Functor is its own dual.)

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

```
class Functor m ⇒ Monad m where
  return :: a → m a
  bind :: (a → m b) → (m a → m b)
  join :: m (m a) → m a

  join = bind id
  bind f = fmap f ∘ join

(>>=) :: m a → (a → m b) → m b
(>>=) = flip bind
```

```
class Functor w ⇒ Comonad w where
  extract :: w a → a
  extend :: (w b → a) → (w b → w a)
  duplicate :: w a → w (w a)

  duplicate = extend id
  extend f = fmap f ∘ duplicate

(=>>) :: w b → (w b → a) → w a
(=>>) = flip extend
```

Intuition

- Monadic values are typically *produced* in effectful computations:

```
a → m b
```

- Comonadic values are typically *consumed* in context-sensitive computations:

```
w a → b
```

Monad laws

Left identity	$\texttt{return} \circ \texttt{bind f} \quad = \texttt{f}$
Right identify	$\texttt{bind return} \quad = \texttt{id}$
Associativity	$\texttt{bind f} \circ \texttt{bind g} \quad = \texttt{bind (f} \circ \texttt{bind g)}$

Comonad laws

Left identity	$\texttt{extract} \circ \texttt{extend f} \quad = \texttt{f}$
Right identity	$\texttt{extend extract} \quad = \texttt{id}$
Associativity	$\texttt{extend f} \circ \texttt{extend g} \quad = \texttt{extend (f} \circ \texttt{extend g)}$

Table of Contents

Example: reader/writer duality

```
-- Reader monad
instance Monad ((→) e) where
    return = const
    bind f r = λc → f (r c) c
```

```
-- CoReader (a.k.a. Env) comonad
instance Comonad ((,) e) where
    extract = snd
    extend f w = (fst w, f w)
```

```
-- Writer monad
instance Monoid e ⇒ Monad ((,) e)
  where
    return = ((,) mempty)
    bind f (c, a) = (c ◊ c', a')
        where (c', a') = f a
```

```
-- CoWriter (a.k.a. Traced) comonad
instance Monoid e ⇒ Comonad ((→) e)
  where
    extract m = m mempty
    extend f m = λc →
      f (λc' → m (c ◊ c'))
```

```
newtype State s a = State  { runState :: s → (a, s) }
instance Monad (State s) where
    return a = State $ λs → (a, s)
    bind f (State g) = State $ λs →
        let (a, s') = g s
        in runState (f a) s'


data Store s a = Store (s → a) s -- a.k.a. ``Costate''
instance Comonad (Store s) where
    extract (Store f s) = f s
    extend f (Store g s) = Store (f ∘ Store g) s
```

One definition of Lens:

```
type Lens s a = a → Store s a
```

Hence the statement that lenses are "the coalgebras of the costate comonad".

Example: stream comonad

```haskell
data Stream a = Cons a (Stream a)

instance Functor Stream where
    fmap f (Cons x xs) = Cons (f x) (fmap f xs)

instance Comonad Stream where
    extract (Cons x _ ) = x
    duplicate xs@(Cons _ xs') = Cons xs (duplicate xs')
    extend f xs@(Cons _ xs') = Cons (f xs) (extend f xs')
```

- `extract = head`, `duplicate = tails`.
- `extend` *extends* the function `f :: Stream a → b` by applying it to all tails of stream to get a new `Stream b`.
- `extend` is kind of like `fmap`, but instead of each call to `f` having access only to a single element, it has access to that element and the whole tail of the list from that element onwards, i.e. it has access to the element and a *context*.

Example: list zipper

```
data Z a = Z [a] a [a]

left, right :: Z a → Z a
left (Z (l:ls) a rs) = Z ls l (a:rs)
right (Z ls a (r:rs)) = Z (a:ls) r rs

instance Functor Z where
  fmap f (Z l a r) = Z (fmap f l) (f a) (fmap f r)

iterate1 :: (a → a) → a → [a]
iterate1 f = tail ∘ iterate f

instance Comonad Z where
  extract (Z _ a _) = a
  duplicate z = Z (iterate1 left z) z (iterate1 right z)
  extend f z = Z (fmap f $ iterate1 left z) (f z)
                 (fmap f $ iterate1 right z)
```

Example: list zipper (cont.)

- A *zipper* for a data structure is a transformed structure which gives you a *focus* element and a means of stepping around the structure.
- `extract` returns the focused element.
- `duplicate` returns a zipper where each element is itself a zipper focused on the corresponding element in the original zipper.
- `extend` is kind of like `fmap`, but instead of having access to just one element, each call to `f` has access to the entire zipper focused at that element. I.e. it has the whole zipper for context.
- Compare this to the Stream comonad where the context was not the whole stream, but only the tail from the focused element onwards.
- It turns out that *every zipper is a comonad*.

Example: array with context

```
data CArray i a = CA (Array i a) i

instance Ix i ⇒ Functor (CArray i) where
  fmap f (CA a i) = CA (fmap f a) i

instance Ix i ⇒ Comonad (CArray i) where
  extract (CA a i) = a ! i
  extend f (CA a i) =
    let es' = map (λj → (j, f (CA a j))) (indices a)
    in CA (array (bounds a) es') i
```

- `CArray` is basically a zipper for `Array`s.
- `extract` returns the focused element.
- `extend` provides the entire array as a context.

Table of Contents

Application: 1-D cellular automata – Wolfram's rules

```haskell
rule :: Word8 → Z Bool → Bool
rule w (Z (a:_) b (c:_)) = testBit w (sb 2 a .|. sb 1 b .|. sb 0 c) where
    sb n b = if b then bit n else 0

move :: Int → Z a → Z a
move i u = iterate (if i < 0 then left else right) u !! abs i

toList :: Int → Int → Z a → [a]
toList i j u = take (j - i) $ half $ move i u where
   half (Z _ b c) = b : c

testRule :: Word8 → IO ()
testRule w = let u = Z (repeat False) True (repeat False)
      in putStr $ unlines $ take 20 $
         map (map (λx → if x then '#' else ' ') ∘ toList (-20) 20) $
         iterate (=>> rule w) u
```

Application: 2-D cellular automata – Conway's Game of Life

```
data Z2 a = Z2 (Z (Z a))

instance Functor Z2 where
    fmap f (Z2 z) = Z2 (fmap (fmap f) z)

instance Comonad Z2 where
    extract (Z2 z) = extract (extract z)
    duplicate (Z2 z) = fmap Z2 $ Z2 $ roll $ roll z where
        roll a = Z (iterate1 (fmap left) a) a (iterate1 (fmap right) a)
```

Application: 2-D cellular automata – Conway's Game of Life

```
countNeighbours :: Z2 Bool → Int
countNeighbours (Z2 (Z
    (Z (n0:_) n1 (n2:_):_)
    (Z (n3:_) _  (n4:_))
    (Z (n5:_) n6 (n7:_):_))) =
        length $ filter id [n0, n1, n2, n3, n4, n5, n6, n7]

life :: Z2 Bool → Bool
life z = (a && (n == 2 || n == 3))
       || (not a && n == 3) where
    a = extract z
    n = countNeighbours z
```

```
laplace2D :: CArray (Int, Int) Float → Float
laplace2D a = a ? (-1, 0)
            + a ? (0,  1)
            + a ? (0, -1)
            + a ? (1,  0)
            - 4 ∗ a ? (0, 0)

(?) :: (Ix i, Num a, Num i) ⇒ CArray i a → i → a
CA a i ? d = if inRange (bounds a) (i + d) then a ! (i + d) else 0
```

- `laplace2D` computes the Laplacian at a single context, using the focused element and its four nearest neighbours.
- `extend laplace2D` computes the Laplacian for the entire array.
- Output of `extend laplace2D` can be passed to another operator for further processing.

Application: Env (CoReader) for saving and reverting to an initial value

```
type Env e a = (e, a)

ask :: Env e a → e
ask = fst

local :: (e → e') → Env e a → Env e' a
local f (e, a) = (f e, a)


initial = (n, n) where n = 0

experiment = fmap (+ 10) initial

result = extract experiment

initialValue = extract (experiment =>> ask)
```

Other applications of comonads

- Signal processing: using a stream comonad.
- Functional reactive programming: it has been postulated (e.g. by Dan Piponi, Conal Elliott) that some sort of "causal stream" comonad should work well for FRP, but there don't yet seem to be any actual implementations of this.
- Gabriel Gonzalez's three examples of "OO" design patterns:
  - The Builder pattern: using `CoWriter` / `Traced` to build an "object" step-by-step.
  - The Iterator pattern: using `Stream` to keep a history of events in reverse chronological order.
  - The Command pattern: using `Store` to represent an "object" with internal state.

At least two different proposals for a comonadic equivalent of `do` notation for comonads:

- Gonzalez's `method` notation – "OOP-like" with `this` keyword representing the argument of the function passed to `extend`.
- Orchard & Mycroft's `codo` notation – resembles Paterson's arrow notation.

Unsugared	Gonzalez	Orchard & Mycroft

λwa $\rightarrow$
  let wb = extend (λthis $\rightarrow$ expr1) wa
      wc = extend (λthis $\rightarrow$ expr2) wb
  in   (λthis $\rightarrow$ expr3) wc
	method
	  wa> expr1
	  wb> expr2
	  wc> expr3
		codo wa $\Rightarrow$
		  wb $\leftarrow$ λthis $\rightarrow$ expr1
		  wc $\leftarrow$ λthis $\rightarrow$ expr2
		  λthis $\rightarrow$ expr3

λwa $\rightarrow$
  let wb = extend func1 wa
      wc = extend func2 wb
  in   func3 wc
	method
	  wa> func1 this
	  wb> func2 this
	  wc> func3 this
		codo wa $\Rightarrow$
		  wb $\leftarrow$ func1
		  wc $\leftarrow$ func2
		  func3

```haskell
-- Monad transformers
class MonadTrans t where
    lift :: Monad m ⇒ m a → t m a


-- Comonad transformers
class ComonadTrans t where
    lower :: Comonad w ⇒ t w a → w a
```

The *comonad* package provides a few standard transformers:

- `EnvT` – analogous to `ReaderT`
- `StoreT` – analogous to `StateT`
- `TracedT` – analogous to `WriterT`

```haskell
-- Free monad
data Free f a = Pure a | Free (f (Free f a))
instance Functor f => Monad (Free f) where
    return = Pure
    bind f (Pure a) = f a
    bind f (Free r) = Free (fmap (bind f) r)

-- Cofree comonad
data Cofree f a = Cofree a (f (Cofree f a))
instance Functor f => Comonad (Cofree f) where
    extract (Cofree a _) = a
    extend f w@(Cofree _ r) = Cofree (f w) (fmap (extend f) r)
```

- `Cofree Identity` is an infinite stream.
- `Cofree Maybe` is a non-empty list.
- `Cofree []` is a rose tree.

```
-- Kleisli category identity and composition (monads)
return :: Monad m ⇒ a → m a
(>=>) :: Monad m ⇒ (a → m b) → (b → m c) → (a → m c)
f >=> g = λa → f a >>= g

-- Co-Kleisli category identity and composition (comonads)
extract :: Comonad w ⇒ w a → a
(=>=) :: Comonad w ⇒ (w a → b) → (w b → c) → (w a → c)
f =>= g = λw → f w =>> g
```

- Each monad has a corresponding Kleisli category with morphisms `a → m b`, identity `return` and composition operator `(>=>)`.
- Each comonad has a corresponding Co-Kleisli category with morphisms `w a → b`, identity `extract` and composition operator `(=>=)`.

Category laws

Monad laws

Left identity	`return >=> f`	=	`f`
Right identify	`f >=> return`	=	`f`
Associativity	`(f >=> g) >=> h`	=	`f >=> (g >=> h)`

Comonad laws

Left identity	`extract =>= f`	=	`f`
Right identify	`f =>= extract`	=	`f`
Associativity	`(f =>= g) =>= h`	=	`f =>= (g =>= h)`

Category laws

Left identity	$\mathtt{id} \circ \mathtt{f}$	=	$\mathtt{f}$
Right identify	$\mathtt{f} \circ \mathtt{id}$	=	$\mathtt{f}$
Associativity	$(\mathtt{f} \circ \mathtt{g}) \circ \mathtt{h}$	=	$\mathtt{f} \circ (\mathtt{g} \circ \mathtt{h})$

Table of Contents

Further reading

- http://blog.sigfpe.com/2006/12/evaluating-cellular-automata-is.html
- http://blog.sigfpe.com/2008/03/comonadic-arrays.html
- http://blog.sigfpe.com/2008/03/transforming-comonad-with-monad.html
- http://blog.sigfpe.com/2014/05/cofree-meets-free.html
- https://www.fpcomplete.com/user/edwardk/cellular-automata
- http://www.jucs.org/jucs_11_7/signals_and_comonads/jucs_11_7_1311_1327_vene.pdf
- http://www.haskellforall.com/2013/02/you-could-have-invented-comonads.html
- http://www.cl.cam.ac.uk/~dao29/publ/codo-notation-orchard-ifl12.pdf
- http://conal.net/blog/tag/comonad