(Co)Monads for ~~Dummies~~ *Grad Students*

Math's answer to 'It Depends'

Paul Hachmann

hachmap@mcmaster.ca

Outline

- Review of Monads
- The Dual Link
- Programming with Co-Monads

Monads - what are they good for?

In a world without monads ...

- All programs (λ-expressions) would reduce to a single 'unit'.
- The result of these programs would be the same no matter what input is given to the program (Since 'reading in inputs' requires monads in the first place)
- This is boring.

We want something to handle the vagaries of life.

Enter Monads - Spice up your life!

- Monads are a mathematical structure which can abstractly handle "uncertainty" (at compile time)
- You use them in your everyday programs without even knowing!
- It's time to bring them to light!

Example 1 : The Maybe Monad

- Haskell definition of 'Maybe':

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

- But which *is* it?

Knowing that an object is of type `Maybe a` does not tell you which of the above two it actually represents, just as we cannot tell from the outside of the box enclosing Schrdinger's cat if it holds a live cat or a dead one

A Monad - From the Outside

- Perspective 1 : View a monad as a 'black box'

  We cannot safely open up the box, as we don't know for sure what's in it

- Perspective 2 : View a monad as a (delayed) 'computation'

  We cannot (in general) know the result of a monad until runtime

A Monad - From the Inside

A monad has the following operations on it:

- `return : Monad m => a -> m a`

  Encloses a value into a monadic one. Note the similarity of this and the 'hungry' type

- `join : Monad m => m (m a) -> m a`

  Allows us to 'flatten' multi-layer monadic values into single-layer ones (the internal machinery gets composed (somehow - it depends on the monad in question)

- `bind :`
  `Monad m => m a -> (a -> m b) -> m b`

  Allow us to transform monadic values of one type into another

'`bind`' is also written infix as '`>>=`'

Also helpful is `fmap` with signature:
`Functor f => (a -> b) -> (f a -> f b)`
This allows us to modify the 'value' inside a monad without (unsafely) extracting it
Note: Monads are functors

Monad satisfaction laws

The previous operations just had signatures - how do we know what they mean?

- Monads must satisfy the following three laws:
- `(return x) >>= f == f x` - Left identity
- `m >>= return == m` Right identity
- `(m >>= f) >>= g`
  `==`
  `m >>= (\ x -> f x >>= g)` Associativity

Example 1 revisited : Maybe?

Let's see how 'Maybe' is a monad!

- `return = Just`
- `Just x >>= k = k x`
  `Nothing >>= k = Nothing`
- or, `join x = case x of`
  `(Just y) -> y`
  `Nothing -> Nothing`
- `xs >>= f = join (fmap f xs)`

How to get a value from a 'Maybe'? Use 'fromJust' (not safe!)

Exapmle 2: The List Monad

Yes, lists are Monads!
They plainly express non-determinism (consist of a varying number of 'results')
How to implement as a monad?

- `return x = [x]`
- `>>= = flip concatMap`

How to get a value from a List? Use 'head' or 'lookup' (not safe!)

Example 3 : A State Monad

Let us define a type such as :
`State s a = {runState ::  s -> (a,s)}`
This is a monad!

- `return x = State (λs -> (x,s))`

  Given a value, creates a state monadic value which always just outputs that value and stays in the same state

- `fmap f (State m) = State (onVal f .  m)`
  `where onVal f (x, s) = (f x, s)`

  Allows us to modify a state monadic value by applying a function to what would be its output

- `join xss = State (`
  `λs -> uncurry runState (runState xss s)`
  `)`

  Join 'flattens' two layers of state monads (transitions) by running the two transitions in sequence

A final example : The IO Monad

Finally, something worth `write`ing about!
For IO we have `do` notation which allows us to bypass the use of 'bind'.
One can think of an IO monad as a state monad where the state is the state of the machine.
(With the exception that you can't backtrack)

Comonads at last!

- Comonads are the dual of monads.
- While monads push things into a 'box', comonads allow one to pull things *out* of a box.
- Simple example: Streams
  `Stream s = s` $\times$ `Stream s`
- This allows one to pull out items from a stream, while still retaining a stream.

Comonads defined

Comonads have the following operations in Haskell:

- `extract :: w a -> a` (a.k.a. coeval) Dual of return
- `duplicate :: w a -> w (w a)` Dual of join
- `extend :: (w a -> b) -> w a -> w b` A flipped kind of =>>
- `(=>>) :: w a -> (w a -> b) -> w b` Dual of bind
- `(.>>) :: w a -> b -> w b` A kind of 'seq'

Comonad operation Rules

- `(=>> coeval) = id` Right identity
- `coeval . (=>> f) = f` Left identity
- `(=>> f).(=>> g) = (=>> (f . (=>> g)))` Associativity

Monads vs. Comonads

Operation	Monads	Comonads
Extracting values	unsafe	safe
Creating	safe	unsafe
return / coeval	`a -> m a`	`w a -> a`

Using Stream comonads

```
data Stream a = S b (b -> a) (b -> b)
```

The first function generates an object from the current stream, wheras the second function modifies it

Then we have

```
coeval (S s f g) = f s
extend h (S s f g) = S s (λ s' .  h (S s' f g ) ) g
```

Can use two seperate operations on streams:

```
shd :  Stream a -> a
stl :  Stream a -> Stream a
```

Alternatively,

```
counit (v :< _) = v
cobind f val@(x :< xs) =
    f val :< cobind f xs
```

Anticipation and Delay

Can't wait for the rest of the stream?

```
next (_ :< xs) = xs
```

Want to wait a bit?

```
a `fby` b = a :< b
```

One can construct streams (unsafely!) via 'produce':

```
produce ::  (a -> a) -> a -> Stream a
produce fun init =
    let x = fun init in x :< produce fun x
```


Using aniticipation and delay

```
pos = 0 fby (pos + 1)
fact = 1 fby (fact * (pos + 1))
sum x = (0 fby sum x) + x
avg x = ((0 fby x) + x + next x) / 3
fib = 0 fby (fib + (1 fby fib))
```

Example: A stream of fibs

`0 fby (fib + (1 fby fib) )`	0 1 1 2 3 5 ...
`1 fby fib`	1 0 1 1 2 3 5 ...
`fib + (1 fby fib)`	1 1 2 3 5 ...

In Haskell, we would define `fib` as a function which takes a comonad as an argument, e.g.

```
fib x = 0 'fby' cobind ( e -> fib e + (1 'fby' cobind fib e ) ) d
```

As a somewhat simpler example, we would write `sum` as

```
sum x = (0 'fby' cobind sum x) + counit x
```

(See reference 4 for details, if interested)

The OI Comonad

We can have a 'comain' function with type:

```
main ::  OI () -> ()
```

With operations such as:

```
hGetChar' ::  OI Handle -> Char
```

```
hPutChar' ::  OI Handle -> OI Char -> ()
```

In the IO monad, the resultant value of these functions carried the 'monadic baggage' along with them. When using the OI comonad, it is the objects that 'interact' with the outside world which carry state information with them (and thus must be enclosed in the OI comonad).

Using Context comonads

We construct a context:

```
data Context c a = Context (c -> a) c
```

Then we can use the following operations:

```
get ::  Context c a -> c
```

```
modify ::  (c -> c) -> Context c a -> a
```

```
experiment ::
    [c -> c] -> Context c a -> [a]
```

```
liftCtx ::  (a -> b) -> Context c a -> b
```

Context comonad example

We define a function within the 'context' of having an argument of 3:

```
> let x = Context (λ n -> take n [1..10]) 3
> get x
```

Result: `3`

We can then modify that context ***later on****, before the function uses it:*

```
> modify (+1) x
```

Result: `[1,2,3,4]`

```
> modify (*3) x
```

Result: `[1,2,3,4,5,6,7,8,9]`

```
> experiment (fmap (+) [1..5]) x
```

Result: `[[1,2,3,4],[1,2,3,4,5],[1,2,3,4,5,6],[1,2,3,4,5,6,7],[1,2,3,4,5,6,7,8]]`

References

1. *All About Monads*, Jeff Newbern (ed.), (September 2005) link
2. *Codata and Comonads in Haskell*, Richard B. Kieburtz, Oregon Garduate Institute (June 1999) link
3. *The Essence of Dataflow Programming*, Tarmo Uustalu and Varmo Vene, APLAS 2005, LNCS 3780 (2005) link
4. *Comonadic functional attribute evaluation*, Tarmo Uustalu and Varmo Vene, in **6th Symposium on Trends in Functional Programming** (September 2005) link
5. *Comonad Dataflow Programming*, Tarmo Uustalu (November 2005) link
6. Haskell Comonad Wiki : http://www.haskell.org/hawiki/CoMonad
7. Haskell Comonads (& Context Comonad) APIs :
8. http://www.eyrie.org/ zednenem/2004/hsce/Control.Comonad.Context.html
9. http://www.eyrie.org/ zednenem/2004/hsce/Control.Comonad.html
10. *Comonads and Haskell*, Einar Karttunen (September 2005) link
11. *Arrows : A general Interface to Computation*, Ross Paterson (August 2006) link