Comonads

Musings on ‘Signals and Comonads’
by Tarmo Uustalu and Varmo Vene

Andrew Hughes

Theory SIG - 28/10/2005

Outline

Monads

- Monads look at sequential computations with a context.
- For example, a monad may be used to compose a series of functions which manipulate a state.
- Values become monadic via use of the unit function, `return`. This associates the context with the value.
- The bind operator, »=, allows functions to be performed on the value within the monadic wrapper. This can allow side effects, as essentially two functions are performed.
- These two functions form the `Monad` type class. A type class specifies the functions that must be implemented for a type to be considered an instance of that class.
- Note that there is no function in the `Monad` type class for retrieving the pure value again.

Comonads

- Comonads look at sequential computations in a context.
- For example, a comonad may be used to represent data within a stream.
- Values are retrieved from the context using the `counit` function.
- The `cobind` function allows the value to be manipulated within its context.
- These two functions form the `Comonad` type class.
- Note that there is no function in the `Comonad` type class for placing a value in a context.

The Relationship

- Comonads are effectively the inverse of monads.
- While the monadic unit function wraps a value in a monadic context, the comonadic unit function does the inverse and retrieves the value from the context.
- Likewise, the function used by »= takes a value and returns a monadic result, while `cobind`'s function takes a comonadic input and returns a value.
- This is reflected in category theory, as the category of comonads is the dual of the category of monads. But more of this next time. . .

Arrows

- Arrows are a more general construct.
- Monads and comonads can both be represented by arrows. . .
- . . . but it's a bit like using a chainsaw to cut cake.
- We don't need the power of arrows where monads or comonads will do.
- Part 3 of the *Functional Computation* reading group will look at these.

The `Monad` type class

- Recall the `Monad` type class:

Definition

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

- To create a new type of monad, we simply implement these two functions for a particular type. The type can then engage in sequential composition via the »= function.

Handling Errors

- Functions don't always manage to compute a value.
- In many situations, an error may occur.
- We need some way of modelling the fact that a function resulted in an error.
- Effectively, this means that a function that may err produces an error value in addition to its normal set of results.
- For example, a function returning a boolean value may actually produce one of `True`, `False` or `Error`.
- In C++ and Java, the error value is represented by exceptions.
- In Haskell, we can use the `Maybe` type.

An Example Function

- Imagine a function which searches for a particular name in a list, and returns its index.
- How do we deal with the case where the name doesn't exist? Simply returning an integer won't handle this.
- The `Maybe` type is defined as:

Definition

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

- For a type, a, an instance of `Maybe` can represent either 'just' the value or nothing (indicating an error).

A Monadic Solution

- So, we can type our search function as:

Example

```
search :: [String] -> String -> Maybe Int
```

- But – now we have another problem. . .
- It is difficult to use the result of our search as the input to other functions.
- The value we retrieved from the function is trapped inside the `Maybe` data structure, which carries the additional information about whether or not an error occurred.
- This is analogous to the idea we introduced earlier of a monad associating a value with additional information.

Maybe Becomes A Monad

- We can define an instance of the `Monad` class for our `Maybe` type like so:

Definition

```
instance Monad Maybe where
return a = Just a
Just a »= k = k a
Nothing »= _ = Nothing
```

- If our function returns a normal result, »= will simply pass the result in as input to the next function, `k`. Otherwise, `Nothing` is returned, regardless of `k`.

Maybe and »=

- With the bind function, »=, we can feed the possibly erroneous result of one function in as input to another, without the other function having to expect a `Maybe` type as input.
- For example, we could add another function, `findNumber`, which finds the telephone number of a person, using the index of their name in the original list. This may receive an invalid index, and would thus have type `Int -> Maybe Int`. Note that the input does not need to be of type `Maybe`.
- But what about functions that don't return something of type `Maybe`? Well, we can use the unit function, `return`, to wrap any given value inside a `Maybe` structure e.g. `return.(>2)`

How Can We Use Monads?

- How could we use monads to carry around some extra state information? For example, imagine that calling a function incurs some cost whether this be monetary, timewise, or whatever. Can this be modelled using monads? Remember that monads hold the possibility of side-effects, as using »= can cause both the defined bind operation and the supplied function to be performed.
- What other possibilities are there for using monads?
- What things can be more easily modelled with a monad?

Why Use Comonads?

- Are monads not sufficient to model what we need?
- In some cases, monads are simply impractical. Alternatively, comonads may just provide a better semantic fit.
- Streams are a prime example of something monads struggle with.
- With a stream, we generally want to pull data out and use it. But, if the stream is represented by a monad, we simply can't do this.
- Comonads thus fit perfectly, as they perform the inverse, and retrieve values from a context.
- The semantic fit is also better, as we think of data being in a stream, rather than being associated with it.

The Comonad type class

- The `Comonad` type class is an inversion of the `Monad` type class:

Definition

```
class Comonad c where
counit ::  c a -> a
cobind ::  (c a -> b) -> c a -> c b
```

- Recall:

Definition

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


Creating a Stream Type

- To illustrate the use of comonads, we create a stream with:
  - a finite history
  - a present
  - an infinite future

Creating a Stream Type

- We define the type `List` to represent the history.

Definition

```
data List a = Nil | List a :> a
```

- Coupling this with a present value gives us a stream with a present value and finite history:

Definition

```
data LV a = List a := a
```


Creating a Stream Type

- The future of the stream is represented by an infinite type. Hence, there is no base case, only a recursive one.

Definition

```
data Stream a = a :< Stream a
```

- We combine this with our `LV` type to create our final stream of type `LVS`:

Definition

```
data LVS a = LV a :| Stream a
```


Making The Stream Comonadic

- We now make the stream comonadic, which allows us to use stream instances with the `counit` and `cobind` functions.

Definition

```
instance Comonad LVS where
counit (az := a :| as) = a
cobind k d = cobindL d := k d :| cobindS d
```

- The `counit` function allows us to pick out a value from the present stream position.
- The `cobind` function applies a given function throughout the stream. We define `cobindL` and `cobindS` functions to handle the `cobind` operation on the history and future respectively.

Making The Stream Comonadic

- Two cases exist for handling `cobind` over the history list; one for the base case, and one for the recursive case.

Definition

```
cobindL (Nil := a :| as) = Nil
cobindL (az' :> a' := a :| as) = cobindL d' :>
k d'
where d' = az' := a' :| (a :< as)
```

- `d'` is a recreation of the stream in its previous state, when the first item of the history (`a'`) was the present value.
- `cobindL` applies `k` to the history by recreating the stream at each point in history, and then applying `k` to that particular stream.

Making The Stream Comonadic

- `cobindS` only has one case as the stream is infinite.
- The principle is the same as for `cobindL`, except `d'` is now the next point on, rather than the last.

Definition

```
cobindS (az := a :| (a' :< as)) = k d' :<
cobindS d'
where d' = az :> a := a' :| as
```

- Unlike the function used by in »=, `cobind`'s function expects a comonadic input and returns a normal value.

An Example Stream

- We can create streams simply by specifying the values they will contain.

Example

```
Nil :> 4 := 5 :| fun2str (6+)
```

- `fun2str` simply uses a function, `Int -> a`, to create a stream.
- This is not necessarily true of all comonads. Remember: construction is not part of the `Comonad` type class, as it is with monads.

An Example Stream

- The `counit` function can then be used to retrieve the present value.

Example

```
counit (Nil :> 4 := 5 :| fun2str (6+)) = 5
```


An Example Stream

- We can also define functions to manipulate the stream, e.g.

Example

```
next ((_ := _) :| (x :< _)) = x
```

- and then use `cobind` to apply them.

Example

```
counit $ cobind next (Nil :> 3 :> 4 := 5 :|
fun2str (6+)) = 6
```

How Can We Use Comonads?

- Recall the `Parser` example from our first reading group. As comonads are a dual to monads, could a comonad be created which serializes a parsed structure?
- What other possibilities are there for using comonads?
- What concepts are more semantically appropriate as comonads, as opposed to monads?

In Conclusion. . .

- Monads provide a useful way of composing functions where some contextual information needs to be carried around.
- Comonads complement monads, and allow us to represent values immersed in some context e.g. data within a stream.
- So what can arrows achieve that these methods can't?
- Hopefully, we will find out when we cover this topic.
- The mailing list (`theory@dcs.shef.ac.uk`) and wiki are available for further discussion.
- Thanks for listening.

`http://www.dcs.shef.ac.uk/wiki/bin/view/TheorySIG`