Declarative UIs are the Future — And the Future is Comonadic!

Phil Freeman

freeman.phil@gmail.com

```
class Functor w => Comonad w where
  extract :: w a -> a
  duplicate :: w a -> w (w a)

instance Comonad (Store s) where
  extract (Store here view) =
    view here
  duplicate (Store here view) =
    Store here (\next -> Store next view)
```

Figure 1. The `Comonad` type class and `Store` instance

```
newtype Co w a = Co
  { runCo :: forall r. w (a -> r) -> r }

instance Comonad w => Monad (Co w)

select
  :: Comonad w => Co w (a -> b) -> w a -> w b
select co w = runCo co (extend dist w) where
  dist fs f = fmap (f $) fs
```

Figure 2. The `Co w` monad for a `Comonad w` (Kmett 2011)

Abstract

There are many techniques for the declarative specification of user interfaces, but it is not clear how to study their similarities and differences. The category-theoretic notion of a *comonad* captures some essential aspects of these specification techniques. The approach presented here generalizes several known techniques, but perhaps more interestingly, it also generates several new comonads as we search for ways to represent existing user interfaces.

Keywords Haskell, functional programming, specification, laziness, user interfaces, comonads

1. Motivation

Here is a simple model of a user interface: a user interface consists of a type of states, a value of that type which represents the initial state, and a function from states to views. This data is captured precisely by the `Store` comonad:

```
data Store s a = Store
  { here :: s
  , view :: s -> a
  }
```

We can move to a new state using a helper function:

```
move :: s -> Store s a -> Store s a
move s store = view (duplicate store) s
```

The `duplicate` function is taken from the `Comonad` type class (figure 1).

A comonad represents a (lazy) unfolding of all possible future states of our user interface, as well as the transitions allowed between those states. The `extract` function returns a value for the current state, and `duplicate` replaces each future state with its own structure of future states.

2. Specifications from Comonads

Kmett (2011) defines a monad `Co w` which is constructed from a comonad `w`. For our purposes, we think of its actions as selecting some possible future state from a collection of future states described by `w`. Figure 2 defines the `Co w` monad, and the `select` function which selects a future state.

We can now define a general framework for user interfaces. A specification for a user interface will be described by a comonad `w` which describes the type of reachable next states. The `Co w` monad for that comonad will describe the transitions which are allowed.

We can use the `Co w` monad to understand the user interfaces that we can describe using the comonad `w`.

An implementation would `extract` the current view at each state, and user events would be associated with state transitions via the `Co w` monad.

3. Examples

In this section, we will interpret several known comonads in this new context, and characterize the user interfaces that they describe. We will see that our approach generalizes some known techniques.

3.1 The Store Comonad

We have already seen that the `Store s` comonad provides a useful model of user interfaces with a state of type `s`.

The `Co (Store s)` monad is isomorphic to the usual `State s` monad, providing full read/write access to the current state. In this sense, the `Store s` comonad is a very unrestrictive model.

As an example, we can rewrite our `move` function from the previous section in terms of `Co (Store s)`:

```
moveT :: s -> Co (Store s) ()
moveT s = Co (\w -> view w s ())
```

3.2 Moore Machines

Moore machines form a comonad:

```
data Moore i a = Moore a (i -> Moore i a)
```

Transitions are restricted — in order to change state, the user must provide an input of type `i`.

This approach is similar to the Elm architecture (Czaplicki 2013).

3.3 The Cofree Comonad

Moore machines are a special case of a *cofree comonad*:

```
data Cofree f a = Cofree a (f (Cofree f a))
```

Here, transitions can be precisely described by the choice of some functor `f`. It is possible to allow transitions limited read/write access to the current state.

In fact, under certain conditions on `f`, the `Co (Cofree f)` monad is isomorphic to a free monad which is determined by `f`.

This approach is reminiscent of the approach taken in the Halogen user interface library (De Goes et al. 2015).

4. Composing Specifications

In this section, we will work in the other direction, looking for comonads which correspond to common patterns of composition in user interfaces.

4.1 Sums

Given two user interfaces, a common pattern is to display one or the other at a time, and to allow the user to switch between the two.

To achieve this, we need to store all future states of both user interfaces, and an additional bit of information to indicate which user interface is currently visible. This data is packaged in the following data type, which we observe to be a `Comonad`:

```
data Sum f g a = Sum Bool (f a) (g a)
```

Theorem 1. *The `Sum` of two comonads is itself a comonad.*

Proof. See Freeman (2017). □

4.2 Day Convolution

Another common pattern is to display two user interfaces side-by-side, with their states evolving independently.

A comonad to specify such an interface would again need to store all future states of both components, and we would also need a function to render any two current states. This data is captured by the *Day convolution* (Day 1970) of the two functors, expressed here as an *existential data type*:

```
data Day f g a =
  forall x y. Day (x -> y -> a) (f x) (g y)
```

The following result becomes necessary:

Theorem 2. *The Day convolution of two comonads is itself a comonad.*

Proof. See Freeman (2016). □

The `Co (Day f g)` monad for a Day convolution of comonads is difficult to describe in general, but there exist natural transformations from `Co f` and `Co g` to `Co (Day f g)` which embed transitions for the individual components as transitions for the composition.

4.3 Discussion

Here, we started with existing, common user interface patterns and discovered new comonads. What other idioms lead to useful `Comonad` instances?

The Day convolution gives the category of comonads the structure of a *symmetric monoidal category*. Can this be useful for studying user interfaces?

5. Conclusion

Comonads provide a way to generalize several approaches to the specification of user interfaces, and compositions of comonads correspond to compositions of those specifications. This generalization suggests a principled approach to the design and implementation of user interfaces, in which the specifications themselves become the objects of study.

Acknowledgments

Many thanks to Edward Kmett and Jeff Polakow for providing useful feedback, and to Arthur Xavier, for testing the ideas presented here.

References

E. Czaplicki, *The Elm Architecture*, 2013 `guide.elm-lang.org/architecture`.

Day, Brian J. *On closed categories of functors*, 1970 Reports of the Midwest Category Seminar IV, Springer Berlin Heidelberg, 1–38.

J. De Goes, G. Burgess, et al., `purescript-halogen`, 2015 `github.com/slamdata/purescript-halogen`.

Freeman, P., *Comonads and Day Convolution*, 2016 `blog.functorial.com/posts/2016-08-08-Comonad-And-Day-Convolution.html`.

Freeman, P., *Comonads for Optionality*, 2017 `blog.functorial.com/posts/2017-10-28-Comonads-For-Optionality.html`.

E. Kmett, *Monads from Comonads*, 2011 `comonad.com/reader/2011/monads-from-comonads`.