A Real-World Application with a Comonadic User Interface

Arthur Xavier[1]

Supervised by Roberto da S. Bigonha[1]
& Phil Freeman

[1]Universidade Federal de Minas Gerais, Department of Computer Science

June 29, 2018

Abstract

The category-theoretical notion of a *comonad* is used by Freeman [2016b] to develop a pure functional model that summarizes and abstracts many different user interface paradigms (or architectures) and their behaviour in applications, whereby each different user interface (UI) paradigm corresponds to a different choice of comonad.

This work presents a real-world application developed using an extended model based on the original by Freeman. These extensions to the original model should address some of its shortcomings, namely the support for asynchronous effects in response to user actions and the composition of user interfaces with communication, both indispensable assets in the development of real-world applications.

Keywords Haskell, PureScript, functional programming, user interfaces, comonads

1 Introduction

User interfaces (UIs) are not only difficult to *design*, but also to *implement*. They are important parts of real-world applications, which, traditionally, have been developed in a heavily imperative and informal style. Recently, however, many techniques for the declarative specification of user interfaces have come into play. They aim to *describe* a user interface in terms of its internal state, instead of giving instructions on *how to build* it from transitions. These techniques, however define several different architectural styles or paradigms for implementing and organizing UIs in applications and, mostly, lack constructs for combination, composition and communication between interface components.

In spite of this multiplicity of techniques for the specification of user interfaces, Freeman [2016b] proposed the use of category-theoretical structures called *comonads* and *monads* in a generalized model that could abstract these several different UI architectures. It was shown in the first part of this work that the original model by Freeman could abstract at least three different known UI architectures—even when used in the same application—, and that it could be extended to support *synchronous* effects in response to user actions and a limited form of composition of UI components with communication [Xavier, 2017a].

For the purposes of this further development, however, the extensions developed in Xavier [2017a] do not suffice. In this work, by "real-world application", we intend to designate an application that has to deal with the concerns pointed by Myers [1993], namely: having to react to user input on real-time in a multiprocessed environment while having to account for modularity and correctness. This implies the need for proper handling of *asynchronous* effects (e.g. network requests) in response to user input, and for powerful methods of composition, combination and communication between user interface components.

1.1 Related work

Freeman [2016b] shows that comonad and monad transformers could be used to hierarchically compose user interface components, and Xavier [2017a] shows that this form of composition is not very flexible in terms of communication between components when used alone by itself.

In the work of Xavier [2017a] a limiting—but conceptually more adequate—method for the execution of user interface components specified as comonads makes it chal-

lenging to use arbitrary comonads as UI components. It also prevents the interleaving of state changes in these components with asynchronous effects in response to user interaction. The replacement of this method by the original one proposed by Freeman [2016b] should enable this possibility through the usage of the techniques described in Kmett [2011a,b].

Regarding the composition and combination of UI components, Freeman [2018] defines conjunctive and disjunctive combinations of comonads—and, consequently, of user interface components. These structures of combination are used in this work with a novel, more powerful model for communication between components, in order to develop complex applications with multiple, encapsulated but communicating user interface components.

1.2 Outline

This work consists of five main sections. In **Section 2**, a concise review of the works of Freeman [2016b] and Xavier [2017a] is given, highlighting the most important points for understanding user interfaces as comonads.

In **Section 3**, the greatest pain points in the models described in Section 2 are highlighted. In **Section 4**, we explain and justify what are the conditions a good approach for the development of user interfaces should satisfy, and what a "real-world" application is, and how one shall be used in this work in order to validate our model for the development of UI-based applications in an industrial or day-to-day setting.

In **Section 5** we present the developed application and the final model that suits the needs highlighted in Section 4. Finally, in **Section 6**, we evaluate the results and the model developed, highlighting its most striking advantages and disadvantages, comparing it to known approaches to the development of declarative user interfaces, and commenting on the development experience and challenges faced. Also, many possibilities for future work and room for improvement or experimentation over the developed model are described in this last section.

2 User interfaces as comonads

User interfaces are commonly described in terms of two constructs: an *internal state*, and a *rendering function*, which, given its internal state, returns a declarative representation of the visualization of the interface, that is, what is to be presented to the user.

Thus, this type of user interface components could be described by a tuple $S \times (S \to A)$, where S is the type (or set) of possible internal states for this component, and A is the type of declarative representations of the interface. This tuple is represented by the `Store` data type:

```
data Store s a = Store (s -> a) s
```

It so happens that, as shown in Xavier [2017b], the `Store` data type is a *comonad*. *Comonads* are mathematical structures that are traditionally used in pure functional programming to model context-dependent notions of computations, or computations within a context [Kieburtz, 1999; Orchard & Mycroft, 2012; Uustalu & Vene, 2005, 2008], and are given in Haskell by the type class:

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

Interpreting the `Comonad` type class in terms of user interfaces, one may think of extract as a function that renders the UI's current state, and of duplicate as a function that captures the possible future interfaces and their respective states. This can be better viewed in the `Comonad` instance for the `Store` data type:

```
instance Comonad (Store s) where
  extract (Store view state) = view state
  duplicate (Store view state) =
    Store (Store view) state
```

In the case of the `Store` data type, the `duplicate` function essentially replaces all future outcomes of the rendering function of type `s -> a` with a new user interface component of type `Store s a` with a different, replaced internal state.

Thus, a comonad can be thought of as a space shaped by the definition of the comonad (in the case of `Store s` it should be a space with coordinates given by the type `s`), where every point of the space is associated with a value of type `a`, and centered around a distinct base point [Freeman, 2016b]. In this sense, executing a user interface consists in *exploring* this *comonadic space*, that is, repeatedly exploding the possible future states and selecting one of those.

If `w` is a comonad, the process of executing a user interface component given by `w`, consists in repeated applications of the `duplicate` and `select` functions below, where `t w` points to a specific future state in `w (w a)`:

```
duplicate :: w a -> w (w a)
select :: t w -> w (w a) -> w a
```

The `duplicate` function allows for peeking into the possible future values of the comonad `w`, while `select`

allows for selecting a specific future. It remains, however, undefined how to develop such a `select` function to represent the transitions or movements in the comonadic space.

2.1 Pairings of functors

It was shown in Freeman [2016b] that such a `select` function could be constructed by using the concept of *pairings of functors* [Kmett, 2008].

```
class Pairing m w | m -> w, w -> m where
  pair :: (a -> b -> c) -> m a -> w b -> c
```

The intuition behind `Pairing` is that the structure of both functors `m` and `w` is related in some way, so that they annihilate each other with the application of the `pair` function. In this sense, the type `m` represents state transitions for a user interface given by `w`, that is, a way of navigating the comonad `w`. Given the type of the `pair` function, it is, then, simple to implement the `select` function:

```
select :: Pairing m w => m b -> w (w a) -> w a
select = pair (\_ wa -> wa)
```

Some monads and comonads provide interesting examples of pairings, as shown in Freeman [2016b]. The case for the pairing functor `m` being a monad is useful when thinking of composed state transitions. If the monad `m`, when paired with a comonad `w`, represents transitions or movements in the comonadic space, then, precisely because it is a monad, multiple sequential transitions can be composed by using the monadic `bind` operation.

As an illustrative example, consider the `State` type:

```
data State s a = State (s -> (s, a))
```

There exists a pairing between `State` and `Store`:

```
instance Pairing (State s) (Store s) where
  pair f (State run) (Store get state) =
    let (a, next) = run state
    in f a (get next)
```

In the pairing above, the current state encapsulated in `Store` is passed to the transition function in `State` that, in turn, returns a value of type `a` and a new state that can be used to retrieve a value from `Store`.

Then this pairing is applied to a value of type `Store s (Store s a)` by means of the `select` function, the value of type `State s b` is used to navigate in the comonadic space defined by `Store s`, and, in terms of user interfaces, it represents a state transition.

Thus, starting with a value `w` of type `Store Int Int`, and applying a sequence of monadic actions of type `State Int ()` with the `select` function, it is possible to obtain a new, transformed `w`:

```
w :: Store Int Int
w = Store id 0

actions :: State Int ()
actions = do
  put 5
  modify (+5)
  s <- get
  put (s * 3 + 1)

w' :: Store Int Int
w' = select actions (duplicate w)
```

In this case, extracting the value contained in `w'` with `extract w'` should return `31`. Given the definition of both `State s` and `Store s`, it is possible to see that, in the context of user interfaces, it models any of those which have an internal state of type `s` that can be read and modified freely.

This technique is directly related to the interpretation of programs described in a Free monad by coalgebras of a Cofree comonad [Kmett, 2008].

3 Addressed issues

Despite its conceptual simplicity, the implementation and usage of the model described in the previous section for the development of user interface based applications poses some difficult problems.

As shown in Xavier [2017b], having to deal with another specific data type—the pairing functor in this case—drastically hurts the quest for generalization, and also makes it extremely laborious to experiment with and to use different comonads to represent different UI architectures. The comonad combinators described in Freeman [2018], for example, cannot be used in such a setting, as coming up with a pairing functor for these combinators is a somewhat arduous task.

Another direct consequence of the use of pairings between a comonad and a monad describing state transitions is the impossibility of interleaving effectful computations and state transitions in response to user actions. As state transitions of a user interface component given by a comonad `w` are given by a monad `m` such that there exists a *pairing* between `m` and `w`, the only possibility for interleaving effectful actions and state transitions is with *monad transformers* [Xavier, 2017b]. It is the case,

however, that monad transformers, in this setting, are used for representing actions in hierarchically composed components, as monad transformers pair with comonad transformers.

Another pain point described in Xavier [2017b] is the lack of powerful methods for abstracting communication between components in such a way that the encapsulation property of components is maintained. That is, proper ways for a component to communicate outputs or receive inputs from other components even if its comonad does not allow for these operations.

4 Methodology

In order to validate the assumption of Freeman [2016b] that "the approach can probably be extended to support real-world applications", it is, first, necessary to understand or propose what a "real-world" application could be.

In the same paragraph, Freeman talks about the possibility of support for external input or side effects in response to user actions, an important characteristic of any application developed and used in an industrial or day-to-day setting. However, only this trait is not sufficient for defining real-world applications [Myers, 1993]:

- The nature of the side effects that may be executed in response to user actions is important. Some applications may produce only synchronous effects (e.g. writing to a local database), while others might produce asynchronous effects as well (e.g. performing a network request) whose response must affect the state of the UI.

- Some applications contain just one or a few user interface components, while others contain dozens of them that must communicate with each other.

- Some applications are composed of a single screen, while others have multiple, each one presenting the user unique information and different components.

In this sense, our proposal of a real-world application to validate the extended model of Freeman [2016b] must take these matters in account. Because of that, we propose the development of an *RSS feed reader* as an application capable of validating the use of this model in an industrial or day-to-day setting.

4.1 An RSS feed reader

Really Simple Syndication (RSS) is a Web content syndication format based on the XML 1.0 specification [Winer, 2005]. Services may provide content to users in the form of an *RSS feed*, a periodically updated document maintained by the service that typically contains itemized information such as announcements on Web sites or updates to weblogs [Liu et al., 2005].

An RSS feed reader is, thus, a client application capable of accessing and parsing RSS feed documents, viewing metadata of such documents (such as URL, title and description), persisting the access to these documents (so that the user can periodically view updated content), and displaying the content of these documents to the user.

This type of application satisfies our requirements for a "real-world application", as it: performs synchronous effects when storing a document's URL for future access; performs asynchronous effects when downloading the contents of a document; and may be composed of several components; may present the user at least three different screens, one for adding a new RSS feed, one for viewing the stored feeds, and one for viewing the content of each feed.

5 Results and discussion

Our real-world application [Xavier, 2018] was developed in the PureScript language [PureScript, 2013] and used the React library [Facebook, 2013] for rendering the user interfaces, that is, presenting them to the user. It makes use of an external library for parsing the RSS feed documents into PureScript data types and is composed of five user interface components, namely:

1. `ViewFeed`: a component for viewing a paginated list of items from an RSS feed;

2. `ViewItem`: a component for reading a selected item;

3. `Router`: a router component that combines the two aforementioned components, that switches between them as requested, and that displays a "No feed selected" message when the application is initialized;

4. `NewFeedModal`: a modal window component that appears on top of the other components when requested and displays a two-step form for adding a new RSS feed;

5. `App`: a global container component that contains and manages the application data and has all the

previously mentioned components as children, besides displaying a side navigation bar where the user can view and access the stored feeds, or show the `NewFeedModal` component to add a new one.

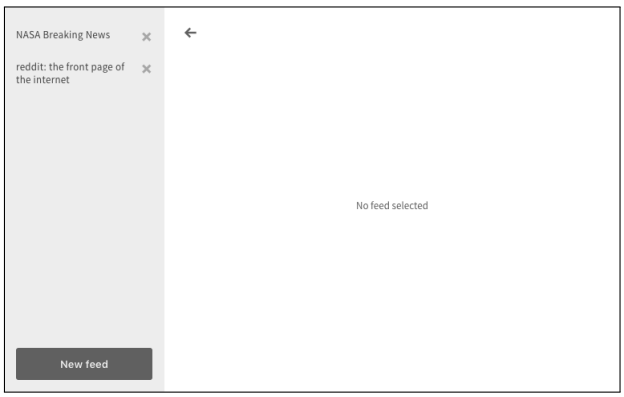

Figure 1: Initial screen of the application with two RSS feeds loaded that can be viewed by clicking on the side navigation menu.

The following subsections discuss the techniques and extensions to the original model used in the development of this application and its user interface components.

5.1 Generating pairing monads for arbitrary comonads

As stated in Section 3, the model used in the first part of this work (where user interface components are given by explicit pairs of comonads and monads) drastically increases the difficulty of using some specific comonads for representing UI architectures.

One highly recurrent pattern that can be expressed by a comonad is that of user interfaces with an implicit state given by a list of other interfaces. This pattern can be used to model sequential pages, paginated content, and step-by-step forms (also known as *wizards*), and is modelled by the `Zipper` data type below:

```
data Zipper a = Zipper [a] a [a]
```

This type describes a non-empty list with a focused element. Its comonad instance describes this behaviour:

```
instance Comonad Zipper where
  extract (Zipper _ focus _) = focus
  duplicate z =
    Zipper (iterate left z) z (iterate right z)
```

The `extract` function simply returns the focused element, while `duplicate` replaces every element in the zipper with a new zipper shifted to have this same element in focus, where `left` and `right` shift a zipper to the left and to the right, respectively.

However, it is not easy to build a pairing monad for this important `Zipper` data type. For this (and other reasons described in Section 3), we decided to make use of a data type described in Kmett [2008] and used in Freeman [2016b] that is itself a pairing monad for any given comonad:

```
newtype Transition w a = Transition {
  runTransition :: forall r. w (a -> r) -> r
  }
```

The `Monad` instance for the `Transition` type can be viewed in Kmett [2011a] (under the name `Co`).

With this data type, it is possible to write a `pairTransition` function that works as the one defined in the `Pairing` type class for any given comonad `w` and any `Transition w`:

```
pairTransition
  :: Functor w
  => (a -> b -> c) -> Transition w b -> w a -> c
pairTransition f t w =
  runTransition t (map (flip f) w)
```

Now, in order to use the `Zipper` comonad as a user interface component, it is only necessary to write monadic actions of type `Transition Zipper` that operate on a zipper. Thus, given a `left` function that shifts a zipper to the left, a monadic transition on zippers that, when paired against a `Zipper` comonad, shifts it to the left is given by:

```
left :: Zipper a -> Zipper a

moveLeft :: Transition Zipper ()
moveLeft = Transition $ \z -> extract (left z) ()
```

Notice how `moveLeft` is a value (not a function), and, as so, does not depend on a `Zipper` value to operate on until it is ran. That is the purpose of using monadic actions to represent state transitions in user interfaces.

As a further example, recall from Section 2 the `Store` data type. A pairing monad for this comonad can be constructed by using the `Transition` data type. Also, the monadic actions of the `State` monad can be replicated and represent the possible state transitions for the `Store` comonad:

```
get :: Transition (Store s) s
get = Transition $
  \s@(Store _ state) -> extract s state
```

```
put :: s -> Transition (Store s) ()
put state = Transition $
  \(Store view _) -> view state ()
```

The `modify` action can be constructed akin to the definition of `put` or by sequencing the `get` and `put` actions.

5.2 Hierarchical composition of comonads

It is known that *monad transformers* are a good approach for extending monads with additional functionality [Liang et al., 1995; Xavier, 2017a]. In the first part of this work, however, monad transformers have been used to model hierarchical composition of user interface components, by expressing the composition of state transitions in these components (given by monads).

With the introduction of the `Transition` monad, however, it is possible to use only `comonad transformers` to represent hierarchical composition of UI components, as proposed by Freeman [2016b].

As detailed in Xavier [2017a], a simple example of a comonad transformer is the `StoreT` comonad, the comonad transformer version of the `Store` comonad:

```
data StoreT s w a = StoreT (w (s -> a)) s

instance Comonad w => Comonad (StoreT s w) where
  extract (StoreT wf s) = extract wf s
  duplicate (StoreT wf s) =
    StoreT (extend StoreT wf) s
```

Its definition can be seen as imbuing the `StoreT` comonad with additional behaviour given by the transformed (or *child*) comonad `w`, as well as giving the child comonad access to the context of `StoreT`. This comonad transformer, however, poses a problem, as the `Transition` monad can only specify transitions for the base (or *parent*) comonad (not for the transformed ones). It is, however, possible to define a function that transforms transitions in a child comonad to transitions in a parent one, given a way of transforming a parent comonad into a child one:

```
hoistTransition
  :: (forall x. v x -> w x)
  -> Transition v a -> Transition w a
hoistTransition transform (Transition t) =
  Transition (t . transform)
```

This allows, for instance, for a parent user interface component to perform transitions or execute actions inside a child component. This is how the `App` component shows the `NewFeedModal` component when the user clicks on the "New feed" button in the side navigation menu.

5.3 Combination of comonads

Other forms of composition of user interfaces and comonads involve horizontal combination instead of hierarchical (vertical) composition. As of Freeman [2018], comonads (and, so, user interfaces) can be combined in at least two distinct ways: products and sums.

Products. The product of two user interfaces can be understood as a simultaneous occurrence of both, that is, both user interfaces can be displayed and accessed at the same time, with their individual states evolving independently.

As shown in Freeman [2018], the *Day convolution* [Day, 1970] of two functors `f` and `g` is itself a comonad whenever `f` and `g` are comonads [Freeman, 2016a]. This concept can be represented in Haskell by the following existential data type:

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

This encoding allows for the encapsulation of the contents of both `f` and `g` while maintaining their functorial and comonadic properties. The `Day` type may be interpreted as containing both comonads simultaneously (what allows their states to evolve independently) and a function that, given the contents of both comonads, is able to produce the final result. This function can be viewed as a rendering function for the product, in the case where `f` and `g` represent user interface components.

Thus, by using the `Day` data type it is possible to combine two independent user interface components into a single one where both coexist and unfold independently.

Sums. The sum of two user interfaces represents a disjoint union, that is, either one or the other interface can be displayed and accessed at a given time, though their states evolve independently.

This behaviour can be represented in a data type similar to `Day`. However, instead of providing a function that produces a final value from the value contained in both comonads, this new data type must only provide a way of indicating which one of the two comonads is "active" at a given moment:

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

The `Comonad` instance for the `Sum` data type must make sure to properly change the boolean value when focusing each of the two comonads to properly represent the disjoint union of user interfaces.

A user interface component for routing. Multiple applications of the `Day` data type may be used to encode the product of multiple user interface components. This can be useful to properly combine multiple independent UI components that may be displayed side-by-side in an application.

By combining multiple methods for composition and combination of components, however, some interesting behaviours may arise. One example is the construction of the `Router` component. It is given as the composition of a `StoreT` comonad transformer and a `Day` product of the `ViewFeed` and `ViewItem` components.

This `Router` component must select between an empty user interface, the `ViewFeed`, and `ViewItem` components based on whether a feed, an item from a feed, or none has been selected. Let `Feed` be the data type that represents an RSS feed and `Item` the type that represents an RSS item. By using the `StoreT` data type, then, its internal state can be given by:

```
data RouterS
  = NoSelection
  | SelectedFeed Feed
  | SelectedItem Feed Item
```

From the definition of `StoreT`, it is possible to see that, if the `Day` product is the comonad transformed by `StoreT`, then, the rendering function provided by `Store` will have access to the context of the product, that is, to both `ViewFeed` and `ViewItem` components:

```
viewFeed :: ViewFeed a
viewItem :: ViewItem a

router
  :: StoreT RouterS (Day ViewFeed ViewItem) a
router =
  StoreT
    (Day render viewFeed viewItem)
    NoSelection
  where
    render :: a -> a -> RouterState -> a
    render viewFeed' viewItem' state =
      case state of
        NoSelection -> ...
        SelectedFeed feed -> ...
        SelectedItem feed item -> ...
```

This component gives a model for an architecture of user interface routers. With repeated applications of the `Day` product, multiple user interface components can be combined together and selected through the state value provided by `StoreT`.

It is important to notice, however, that other different UI architectures may be constructed with the combination and development of more interesting comonads, combinators and combinations of comonads.

5.4 Asynchronous effects

One of the requirements of the real-world application is a way of properly handling effects (synchronous or asynchronous ones), and a way of interleaving effectful actions with state transitions of UI components in response to user interaction.

The proper way of achieving this goal is to encode these effects in the state transitions themselves, that is, in the `Transition` data type. It is known that *monad transformers* are an appropriate tool for adding new computational features to existing monads. In this sense, we need a way of transforming a monad that represents effects in the application by the `Transition` monad. That is, we need to make `Transition` into a monad transformer.

From Kmett [2011b] it is known that such a monad transformer exists, that is, a parametrized data type that is itself a monad transformer whenever its parameter is a comonad. Such a data type is quite similar to `Transition`:

```
newtype TransitionT w m a = TransitionT {
  runTransitionT :: forall r. w (a -> m r) -> m r
  }
```

The `Monad` and `MonadTrans` instances for this data type are given by Kmett [2011b].

Through the use of these instances (mainly the `MonadTrans` one), it is possible to perform state transitions, then side effects in a base monad `m`, then perform again new state transitions based on the possible responses of these side effects.

This behaviour is used in the `NewFeedModal` component, where the user is presented a modal window containing a text box accepting a URL for an RSS feed. When the "Search" button is clicked, a state transition is performed in order to replace the button by an image that indicates the downloading process. When the network request is finished, and the RSS feed metadata is fetched, a new state transition is performed, displaying the next step of the form filled with the fetched data:

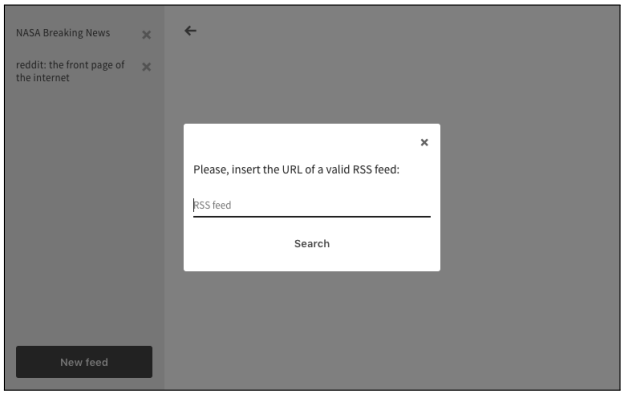

Figure 2: When the "New feed" button on the side navigation menu is clicked, an `Optional` component switches its state in order to display a modal window.

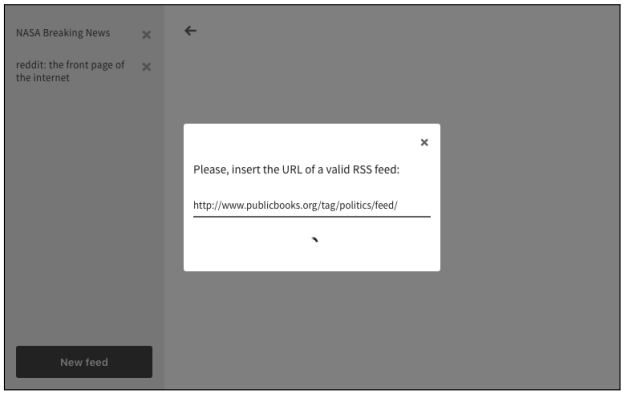

Figure 3: When the "Search" button is clicked, the modal window component performs a state transition and runs an asynchronous HTTP request to the provided address to retrieve metadata for the RSS feed.

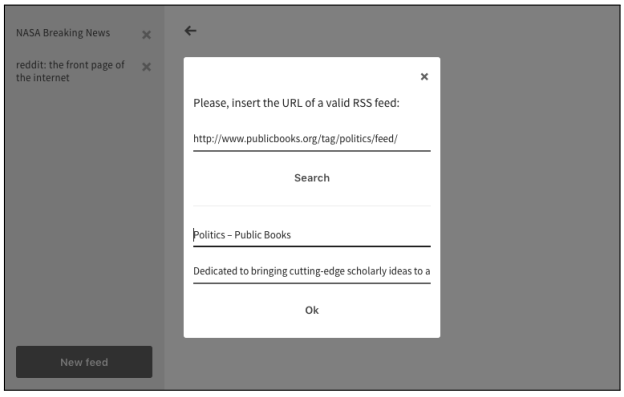

Figure 4: When the HTTP request is completed, the component performs another state transition in order to display the next step of the form.

5.5 Communication patterns

The only requirement of the real-world application left to fulfill, and perhaps the most important one, is a method for composed and combined user interface components to communicate between themselves. The exchange of encapsulated data is one of the most challenging aspects of heavily modularized systems Myers [1993].

Until now, the only two ways two components could communicate with each other are: through state transitions lifted into a component, and through operations provided by a comonad for calculating data. In the case of the `Store` comonad, one of these operations is called `pos`, and provides a way of extracting the current state of the comonad:

```
pos :: Store s a -> s
pos (Store _ state) = state
```

However, as shown in Xavier [2017a], in order for another component to access the state of any other, it must naturally have access to its context. This can be done in the case of composition of comonads with comonad transformers by applying the `extend` function to the child comonad and providing its context to the parent comonad:

```
extend :: Comonad w => w a -> (w a -> b) -> w b
extend = map f . duplicate
```

This is, nevertheless, a limiting approach, as the most interesting forms of composition are given by comonad combinators. Also, because in most cases, for the sake of encapsulation, the comonads cannot provide such operations for the public access of their internal state.

It is necessary, then, to develop a new method for modelling communication between independent user interface components.

In Section 2, we defined a user interface component as a comonad `w` containing interface representations of type `a`. It is possible, however, to change this definition to suit the need for a better model of communication between components. By introducing what we have named a *communication functor* `f` to this definition, a user interface component as a comonad, thus, becomes a value of type `w (f a)`. This communication functor `f` can model different communication architectures between components depending on its type.

Child-to-parent communication. In the case where `f` is the `(,) q` functor for a given type `q`, it abstracts child-to-parent communication. That is, when a parent

component applies the `extract` function to extract the interface representation contained by a child comonad, another value of type `q` is returned alongside the interface representation of type `a`. This value `q` may be encoded in the child component and may be calculated out of its internal state, depending on its implementation.

Parent-to-child communication. In its turn, in the case where `f` is the `(->) p` functor for a given type `p`, it abstracts (a limited form of) parent-to-child communication. That is, when a parent component applies the `extract` function to extract the interface representation contained by a child comonad, a function from `p` to interface representations of type `a` is returned. The parent component must, then, supply a value of type `p` in order to finally have access to the interface representation of its child component.

This type of communication encoded by the `(->) p` functor is, however, limited in the sense that the value of type `p` supplied to the child component cannot affect its internal state, only part of its interface representation.

Bidirectional communication. Two bidirectional communication schemes may be obtained based on the previously defined ones. By composing the two functors `(,) q` and `(->) p` in both directions, two new functors are obtained:

```
type Bidirectional1 p q a = p -> (q, a)
type Bidirectional2 p q a = (p -> a, q)
```

It is important to notice that `Bidirectional1` is isomorphic to the `State` monad and that `Bidirectional2` is isomorphic to the `Store` comonad. Both `State` and `Store` will be used instead of the formers from now on.

The `State` monad, then, defines a communication architecture that, provided an input of type `p` from a parent component, produces an output of type `q` from the child component along with its interface representation of type `a`.

The `Store` comonad, in its turn, defines a communication architecture that produces an output of type `q` independent of any input, and an interface representation of type `a` that depends on an input of type `p`.

It is, however, obvious that the communication scheme described by the `State` data type is more powerful than the one described by `Store`.

6 Conclusions and future work

The approach to the development of user interfaces described in this work is very powerful. As shown in in Freeman [2016b]; Xavier [2017a], comonads can describe several different user interface paradigms, and as shown in this work and in Freeman [2018], more interesting comonads or UI architectures may be discovered through experimentation.

Besides that, by using such classical and somewhat ubiquitous structures such as comonads, this method can leverage a highly developed mathematical framework for talking about user interfaces. This can help better understand the nature of user interfaces and UI-based applications, as well as better understanding why some traditional problems in the development of user interfaces are difficult to solve while some are easier to handle.

When compared to other declarative approaches to the development of user interfaces such as the React library [Facebook, 2013] or the Elm language [Czaplicki, 2012], our method is not only able to reproduce the same architecture of these other approaches, but also to mix and match them in the same application according to the user's needs [Xavier, 2017a]. This enables the possibility of separate applications developed by different teams using different architectures to be merged into a single application if needed.

However, the ergonomics of the method does not yet correspond to its full potential. Being it a novel method, it is probably not yet ready for use in a production environment, as some common tasks are still boilerplate-heavy and others are still difficult to understand. For the most common use cases, however, as this work demonstrates, this approach is perfectly suitable and provides the developer immense power and, sometimes, also agility.

As a direct consequence of using comonads to represent user interfaces, some difficulty may arise. One of the strongest pain points of this approach is the necessity of a static construction of the (lazily evaluated) space of all possible future states of the whole user interface. The use of comonadic combinators does help remedy or ease this process. Nevertheless, the development of new comonads that may be significant as user interface components is hindered by this difficulty.

One other consequence of this fact and also related to the prior construction of the space of all future states is the impossibility for child components to be recreated by their parent components during their life cycle. This "resetting" of components must be encoded as a state transition available to the parent components.

6.1 Future work

The amount of possibilities for future work highlighted throughout this project are rather large. Ranging from the experimentation with interface representations to category theoretical research, the main opportunities are related to the model itself and its use, as its application has been enough proved successful.

Simultaneously explore different components in different rendering structures. It must be investigated whether it is possible to simultaneously execute different, independent user interface components into different rendering structures while account for their communication. This result could provide useful insights on the interpretation of programs.

Further investigate communication functors. It has been shown that different choices of communication functors imply different communication architectures between components. The ones studied in this work, however, arise from the tuple-function adjunction in Haskell. Perhaps more interesting communication schemes could be derived from other interesting choices of communication functors.

Investigate different component life cycles. Currently, the life cycles of UI components are intrinsically bounded to the way they are executed and composed. Further investigation in this matter, be it by studying new ways of executing UI components or by studying new forms of composition, could reveal more interesting and different behaviours for components.

Category theoretical interpretation to pairings of functors. From the type signatures, it can be recognized that a pairing between two functors is a natural transformation from their Day convolution to the identity functor. Further investigation on the category theoretical meaning of pairings could produce interesting developments in this research.

Automatic wiring of communication functors of multiple components. In this work, the introduction of communication functors made it possible to pass information between composed components. This approach, however, requires manual handling of the information components produce or request. By automatically handling this information passing, some boilerplate could be avoided.

Acknowledgements

Many thanks to Phil Freeman for the wonderful collaboration, supervision, teaching and mentorship in this project, and for the development of such an incredible idea that led to this work.

References

Czaplicki, E. (2012). Elm: Concurrent FRP for functional GUIs. *Senior thesis, Harvard University.*

Day, B. J. (1970). On closed categories of functors. *Lecture Notes in Math*, 137:1–38.

Facebook (2013). React, a JavaScript library for building user interfaces. `https://facebook.github.io/react/`.

Freeman, P. (2016a). Comonads and day convolution. `https://blog.functorial.com/posts/2016-08-08-Comonad-And-Day-Convolution.html`. Accessed: 2018-06-26.

Freeman, P. (2016b). Comonads as spaces. `https://blog.functorial.com/posts/2016-08-07-Comonads-As-Spaces.html`. Accessed: 2018-06-26.

Freeman, P. (2018). Declarative uis are the future — and the future is comonadic! *Unpublished.* `https://github.com/paf31/the-future-is-comonadic`. Accessed: 2018-06-26.

Kieburtz, R. B. (1999). Codata and comonads in haskell. *Unpublished manuscript.*

Kmett, E. (2008). The Cofree comonad and the expression problem. `http://comonad.com/reader/2008/the-cofree-comonad-and-the-expression-problem/`. Accessed: 2018-06-26.

Kmett, E. (2011a). Monads from comonads. `http://comonad.com/reader/2011/monads-from-comonads/`. Accessed: 2018-06-26.

Kmett, E. (2011b). Monads transformers from comonads. `http://comonad.com/reader/2011/monad-transformers-from-comonads/`. Accessed: 2018-06-26.

Liang, S., Hudak, P., & Jones, M. (1995). Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343. ACM.

Liu, H., Ramasubramanian, V., & Sirer, E. G. (2005). Client behavior and feed characteristics of rss, a publish-subscribe system for web micronews. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, pages 3–3. USENIX Association.

Myers, B. A. (1993). Why are human-computer interfaces difficult to design and implement? Technical report, Pittsburgh, PA, USA.

Orchard, D. & Mycroft, A. (2012). A notation for comonads. In *Symposium on Implementation and Application of Functional Languages*, pages 1–17. Springer.

PureScript (2013). PureScript programming language. `http://www.purescript.org`. Accessed: 2017-08-16.

Uustalu, T. & Vene, V. (2005). The essence of dataflow programming. In *Central European Functional Programming School*, pages 135–167. Springer.

Uustalu, T. & Vene, V. (2008). Comonadic notions of computation. *Electronic Notes in Theoretical Computer Science*, 203(5):263–284.

Winer, D. (2005). What is rss and how can it serve libraries? `https://cyber.harvard.edu/rss/rss.html`. Accessed: 2018-06-26.

Xavier, A. (2017a). Comonads for user interfaces. *Unpublished*.

Xavier, A. (2017b). `purescript-comonad-ui-todos`. `https://github.com/arthurxavierx/purescript-comonad-ui-todos`. DOI: 10.5281/zenodo.1064750.

Xavier, A. (2018). `purescript-comonad-rss`. `https://github.com/arthurxavierx/purescript-comonad-rss`.