FEDERAL UNIVERSITY OF MINAS GERAIS

DEPARTMENT OF COMPUTER SCIENCE

Comonads for user interfaces

Arthur Xavier

Bachelor Thesis

Supervised by Roberto da Silva Bigonha
& Phil Freeman

Belo Horizonte
2017

I confirm that this bachelor thesis is my own work and I have documented all sources and material used.

Belo Horizonte, 2017 Arthur Xavier

Acknowledgments

I would like to thank my supervisor Prof. Dr. Roberto Bigonha, whose attention and supervision have been indispensable to the conclusion of this project and, especially, of this report.

I'm also grateful to the support and encouragement given by my beloved partner, Luíza, and by my friends.

And, to Phil Freeman, a true icon in the functional programming community, for building the PureScript language and its wonderful community, for always coming up with incredible ideas, and for the amazing supervision, teaching and mentorship in this project, I leave you here my deepest thanks. Without your support none of this could have ever happened. Thank you!

Abstract

User interfaces (UIs) are difficult to implement. They are important parts of real-world applications, which, traditionally, have been developed in a heavily imperative style. Recently, however, many techniques for the building of declarative user interfaces have come into play. These techniques, for the most part, define different architectural styles or paradigms for implementing and organizing user interfaces in applications. Inspired by this, Freeman [2016b] makes use of category-theoretical structures called *comonads* in order to develop a pure functional model that summarizes and abstracts many different user interface paradigms and all their subtleties, whereby each different UI paradigm (or architecture) corresponds to a different comonad.

This thesis presents itself as an extension of the work of Freeman [2016b] in the sense that we validate the suitability of the model proposed in that work for real-world applications. For that matter, we extend this model to support side effects and composition of user interfaces as these are indispensable tools to develop useful user interfaces for complex applications. Furthermore, we demonstrate that our extended model can be used for the development of real-world applications by building a fabricated but somewhat complex application that could be used in the real world.

Contents

1 Introduction

User interfaces are not only difficult to *design*, but also to *implement*. According to Myers [1993], having to react to user input on real-time in a multiprocessed environment while having to account for modularity and correctness, might explain the complexity of the tools available to programmers for the development of user interfaces (UIs).

One main cause for the difficulty and complexity of the tools for graphical user interface (GUI) programming may be the lack of a unified formal basis or formal abstract model for user interfaces, which is consistent with the way GUIs are used and developed in the industry. Libraries for GUI programming are often described in informal terms and lack constructs for combination and composition of interface components as these libraries rely often on mainly imperative models.

But in the recent years, there can be seen a growing interest in languages, libraries and techniques for the development of declarative user interfaces. And alongside the development of these, a handful of architectures and architectural styles for declarative GUIs has been created. In spite of these developments, in this thesis we make use of a very recent approach to UIs by Freeman [2016b] that attempts to generalize these many architectures under a single model. In this work we extend the work of Freeman [2016b] in order to verify its suitability for the development of complex real-world applications.

It is expected from the reader of this report to have some familiarity with the basic concepts of functional programming and of the Haskell language, as well as with the syntax of the language. For reference and instruction in these matters, we recommend the work of Bird & Wadler [1988]. More advanced concepts used throughout this report are detailed in Chapter 2.

1.1 Overview

Several architectures and architectural styles have been developed in order to attack the problem of building complex software systems while accounting for modularity. One most prominent example comes to mind when talking about user interfaces, and that is the model-view-controller architectural pattern, or *MVC*, which emerged in the development process of the user interface for the Smalltalk-80 programming environment [Krasner et al., 1988], and whose several related architectural patterns are still in use up to date [Bragge et al., 2013], mainly in object-oriented programming languages. The MVC and

its related architectural patterns are, however, inherently imperative and depend on imperative techniques such as *data binding* for output rendering.

On the other hand, declarative approaches to interactive user interfaces only started to gain traction with the emergence and popularization of Virtual DOM [Psaila, 2008] and Web Components [Russel, 2011], which brought up a consistent step up into the solution of the old problem of modularization and separation of concerns in GUI programming with techniques for the web. And with this recent popularization of the declarative paradigm, the Web has seen a strong surge of declarative languages, libraries and frameworks for the building of user interfaces.

But despite all the advances in the last years for the construction of user interfaces, there still exists many different models and architectures whose only commonality is their sharing of a declarative approach. It is still difficult to reason about user interfaces in a general way, without having to resort to a specific framework.

With this thesis we aim to attack these problems by making use of a general approach to user interfaces developed by Freeman [2016b] that is capable of formalizing and summarizing many different user interface architectures under a same single model. We expect to study, formally reproduce and validate the correspondences between architectures and the model proposed by Freeman [2016b] by using such model to build interactive programs for the Web, with the possibility of switching between architectures in a fairly easy way.

1.2 Related work

Declarative user interfaces are different from imperative ones in the sense that a greater focus is given to the specification of *what* shall be rendered to the user, instead of *how* to do it. Most of the time, models for declarative user interfaces treat UIs as a function from some state or data to UI elements, be them text, a description of pixels on a screen or elements of a DSL such as HTML.

The most traditional approaches to declarative UIs are based on *Functional Reactive Programming* (dubbed FRP). These models often treat a user interface as a composition of a series of functions that react to and handle input signals over time, producing the description of the current interface as output. Functional programming languages are a good use for such models, given their ease for expressing composition of functions as well as for creating domain specific languages (DSLs) of combinators.

However, one of the most prominent examples of modern approaches to declarative user

interfaces is the *Elm* language [Czaplicki, 2012a], which was heavily influenced by some approaches to user interfaces based on FRP and by the *Flapjax* language of Meyerovich et al. [2009]. The *Elm* language, in turn, influenced a subsequent generation of libraries, languages and architectures for GUI programming, such as *React* [Facebook, 2013] and *PureScript* [PureScript, 2013]. *React* is an example of a library that attempts to solve some problems present in the *Elm* architecture, namely the absence of local (component) state and boilerplate in composition of components.

As an attempt to summarize and generalize all these different models or architectures for GUI programming in declarative languages, Freeman [2016b] proposed the use of structures called *comonads* and *monads* in order to represent spaces of user interfaces and movements in this space, respectively. These structures are known in the field of theory of programming languages, especially to those familiar with functional programming, as they provide means to structure different forms of *computations* in these languages [Uustalu & Vene, 2008; Wadler, 1995].

These concepts will be further detailed in Chapter 2, but for now the reader may bear in mind that comonads are traditionally used for structuring context-dependent computations, whereas monads are commonly used for structuring context-producing or side-effecting computations – and side-effecting computations are impure ones, that is, those which do not preserve the property of referential transparency.

1.3 Outline

This work consists of five main chapters. In **Chapter 2** a short introduction to common patterns of the functional programming paradigm and their relationship to category theoretical concepts is given with focus on the relationship between both domains as a means to introduce the reader to the concepts of monads and comonads as well as to the possibilities of its applications.

In **Chapter 3** we present a concise review the work of Freeman [2016b], explaining the concept of pairings of functors and the take on comonads as a pointed space of values, which will, in turn, be used to model a pointed space of user interfaces.

The main chapter of this thesis is perhaps the **Chapter 4** where we report the main contributions of this thesis: our further explaining and extension of the work of Freeman [2016b] as to model side effects and composition of user interfaces.

In **Chapter 5** we develop a simple application (a personal task management application for the Web browser) using the PureScript [PureScript, 2013] language. We give the

description of this small application, a truly short introduction to the language is given by comparison with the Haskell language and a report of the development of this application using different user interface architectures modelled under our single general approach.

Furthermore, in **Chapter 6** we evaluate the model developed, analysing the overall improvements over the work of Freeman [2016b], the current limitations of our approach as well as the possible threats to this model. Several possibilities for future work and room for improvement or experimentation over the proposed model are also described.

2 Pure functional programming and category theory

A declarative paradigm, functional programming focuses on the *what* aspect of programming (instead of focusing on the *how*, as imperative paradigms do), and, as described by Hughes [1989], it is a style of programming in which a program is a function defined in terms of other functions through composition or application (of functions), which are, in turn, the primary methods for the construction of computations.

Pure functional programming is a stricter version of traditional FP that imposes some restrictions mostly on mutability and side effecting computations in exchange for a powerful property: *referential transparency* (or *purity*). Referentially transparent expressions, functions or programs allow for the replacement of their names by their definitions (as with mathematical expressions) and are, then, semantically comparable and produce no side effects. Most of the time enforced through the use of type systems, this property enables the use of equational reasoning in the design, construction, and verification of programs and is, then, an obvious contributor to the claim that functional languages help building less defectuous programs.

Although severely limited when compared to the abusive freedom of imperative paradigms, pure functional programming draws inspiration for overcoming these limitations from many fields of the mathematical sciences by using mathematical constructions and laws for building abstractions, most notably from the fields of *abstract algebra* and *category theory*. With this, not only do we keep properties such as referential transparency as well as obtain new ones derived from these mathematical abstractions.

The source of many abstractions in functional programming, category theory is a mathematical formalism that is essentially a *"theory of composition"*. It studies *objects* and *morphisms* between them; both as primitive and atomic concepts. Category theory is highly relevant to the study of programming languages as it provides a formalized language for stating abstract properties of structures by means of the analysis of universal relations. Category theoretical concepts are used in the context of functional programming for modelling useful concepts from composable structures to sequential and side effecting computations, as we shall show the reader through the course of this chapter. We shall not cover these concepts from category theory in this report, but only show how they are applied to pure functional programming.

2.1 Types and functions as a category

Statically typed functional languages have their deepest origins on the simply-typed λ-calculus [Barendregt et al., 2013], an extension of the original untyped λ-calculus by A. Church. They model functions as values of some *type*, namely of a *function type*, which is constructed from two other types: the type of the domain and the one of the codomain of the function. A function type is, then, a *type constructor*.

More interesting is the fact that the simply-typed λ-calculus (and some relevant aspects of the Haskell language) can be modelled – for the intents of the analysis presented here – by the category-theoretical concept of a *category* [Asperti & Longo, 1991], what leads us to the following definition:

Definition 2.1.1. A **category** $\mathcal{C}$ is given by a collection $\mathcal{C}_0$ of *objects* (written $A, B, C, \dots$) and a collection $\mathcal{C}_1$ of *morphisms* or *arrows* (written $f, g, h, \dots$) between these objects, whereby:

i. Every morphism has a *domain* and a *codomain* which are objects in the same category; one writes $f : A \to B$ if A is the domain of f and B is the codomain, or $A = \mathrm{dom}(f)$ and $B = \mathrm{cod}(f)$;
ii. Given objects A, B, and C, and two morphisms $f : A \to B$ and $g : B \to C$ (note that $\mathrm{dom}(g) = \mathrm{cod}(f)$), the *composition* of both morphisms must be defined and is written gf or $g \circ f : A \to C$ (read g *after* f);
iii. Furthermore, the composition operation must be *associative*, that is: given $f : A \to B$, $g : B \to C$ and $h : C \to D$, it is true that $h \circ (g \circ f) = (h \circ g) \circ f = h \circ g \circ f$;
iv. And for every object A there exists an *identity* morphism id_A or 1_A, such that, for every $f : A \to B$ and $g : B \to A$, we have that $f \circ 1_A = f$ and $1_A \circ g = g$.

Example 2.1.1. The canonical example of a category is that of sets: the **Set** category. In **Set**, every object is a set, and functions between sets are the morphisms, with specified domain and codomain. Composition of morphisms is composition of functions and the identity arrows are the identity functions of each set.

A category for Haskell

Being a statically typed language, in Haskell, every expression has a type. By applying the categorical reasoning to analyze the Haskell language, one may be able to verify important properties or to come up with interesting abstractions as stated in the introduction for

this chapter. It is, then, natural a process to define a category for this language: the **Hask** category. Apart from the fact that, in Haskell, an expression may not terminate (yielding, thus, a $\perp$ value), mathematical functions and Haskell functions may be considered identical, so for the intents and purposes of the analysis presented in this work, we shall consider the **Hask** and **Set** categories as equivalent.

In this sense, the type of both values `a` and `b` below (that is, `Int` and `Maybe Int`) are objects in the **Hask** category:

```
a :: Int
a = 42

b :: Maybe Int
b = Just 10
```

Morphisms (or arrows) in this category are Haskell functions of any sort. Some difficulty may arise as Haskell functions are also types, and thus, are objects in the **Hask** category. This does, however, simplify the analysis of curried functions, as we can treat them as morphism from an object representing the type of the first argument to the remaining function type. Thus, the type of the value `c` below may be considered either a morphism from `[Int]` to `Int` or an object `[Int] -> Int`:

```
c :: [Int] -> Int
c = (+1) . sum
```

Identity morphisms and composition of morphisms are also well-defined operations for all values of every type in Haskell. They can be defined through the use of parametric polymorphism, as in:

```
id :: a -> a
id x = x

(.) :: (b -> c) -> (a -> b) -> (a -> c)
g . f = \x -> g (f x)
```

2.2 Containers as functors

It was shown that types and functions in Haskell have a category theoretical formalization. But what about *type constructors*? That is, what is `Maybe` or `[]` in a categorical sense? They are a part of a structure called a *functor*. In category theory, functors are simply stated as structure-preserving morphisms between categories themselves (that is, homomorphisms between categories).

Definition 2.2.1. Let $\mathcal{C}$ and $\mathcal{D}$ be categories. A **functor** $F : \mathcal{C} \to \mathcal{D}$ maps objects $A, B, \ldots$ of $\mathcal{C}$ to objects $F(A), F(B), \ldots$ of $\mathcal{D}$. In addition to this, a functor must also map morphisms $f : A \to B$ in $\mathcal{C}$ to morphisms $F(f) : F(A) \to F(B)$ in $\mathcal{D}$ such that:

i. $F(1_A) = 1_{F(A)}$, that is identity morphisms are preserved, and
ii. $F(g \circ f) = F(g) \circ F(f)$, that is composition of morphisms is preserved.

Example 2.2.1. An *endofunctor* is a functor from a category $\mathcal{C}$ to itself. These specific types of functors are used in the Haskell language through the use of type constructors (which map objects – or types – from **Hask** to **Hask**) and through the `Functor` type class (which maps morphisms – or functions):

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

It is important to notice that implementations (or instances) of the `Functor` type class must follow the two rules from definition 2.2.1, namely preservation of identity morphisms and preservation of composition of morphisms.

Functors (or endofunctors) in Haskell abstract the notion of a *container* or that of a result of a computation. Furthermore it abstracts the capability of lifting or applying functions to the values (if any) inside a container through the use of `fmap`.

Example 2.2.2. The `Identity` functor in Haskell is simply a unitary container and may be considered equivalent to the unwrapped value in some cases:

```
newtype Identity a = Identity a

instance Functor Identity where
  fmap f (Identity a) = Identity (f a)
```

Example 2.2.3. The `Maybe` functor can be thought of as a container that may or may not contain a value, or as the result a computation that might have failed:

```
data Maybe a = Nothing | Just a

instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap f (Just a) = Just (f a)
```

2.3 Polymorphic functions as natural transformations

In category theory, a higher abstraction than that of functors is that of *natural transformations*. As we have seen that types and functions in Haskell do form a category, and "functions" between types, that is, type constructors (along with the corresponding mapping of morphisms given by `fmap`) do correspond to functors from category theory, which are in essence morphisms between categories, we are left to consider what are morphisms between functors themselves.

Definition 2.3.1. For any given functors F and G, from a category $\mathcal{C}$ to a category $\mathcal{D}$, a **natural transformation** α from F to G is a family of morphisms in $\mathcal{D}$ such that:

i. For every object A in $\mathcal{C}$, a *component* of α at A is a morphism $\alpha_A : F(A) \to G(A)$ in $\mathcal{D}$; that is, the natural transformation maps objects in $\mathcal{C}$ transformed by the functor F (which are objects in $\mathcal{D}$) to objects transformed by the functor G.
ii. For every morphism $f : A \to B$ in $\mathcal{C}$, the components of a natural transformation must obey what is called the *naturality condition*: $\alpha_B \circ F(f) = G(f) \circ \alpha_A$.

Example 2.3.1. Natural transformations in Haskell can be represented as morphisms between parametrically polymorphic applications of type constructors, as in:

```haskell
alpha :: F a -> G a
```

The naturality condition is granted for free in Haskell by parametric polymorphism. One may also be able to define an infix type operator for natural transformations in Haskell (with the necessary language extensions activated):

```haskell
type f ~> g = forall a. f a -> g a
alpha :: F ~> G
```

Corollary 2.3.1. There may be multiple natural transformations from a given functor to another one.

Example 2.3.2. There are multiple functions that map values of type `[a]` to `Maybe a`:

```haskell
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (a:_) = Just a

nothing :: [a] -> Maybe a
nothing _ = Nothing
```

2.4 Monads

The reader was already introduced to the concept of *monads* in functional programming in the introduction chapter of this report. The application of said concept to computer science was first discovered by Moggi [1991] who used monads as a means to model sequential computations. Later Wadler [1995] studied its applications to functional programming as a *"convenient framework for simulating effects found in other languages"*.

Definition 2.4.1. In category theory a **monad** is a triple (T, η, μ), where T is an endofunctor from a category $\mathcal{C}$ to itself, and η and μ are natural transformations. The first natural transformation, η, maps the identity endofunctor on $\mathcal{C}$ ($1_{\mathcal{C}}$) to T and can be written as $\eta : 1_{\mathcal{C}} \to T$. Whereas the second natural transformation, μ, maps a composition of T with itself to T and can be written as $\mu : T \circ T \to T$ or $\mu : T^2 \to T$. Both natural transformations must be defined such that, for every object A in $\mathcal{C}$:

i. $\mu_A \circ T(\eta_A) = \mu_A \circ \eta_{T(A)} = 1_{T(A)}$
ii. $\mu_A \circ T(\mu_A) = \mu_A \circ \mu_{T(A)}$.

Example 2.4.1. Given that, in Haskell, the identity functor must not necessarily be represented as a type constructor, a monad may, then, be represented by the following type class (notice the requirement that `m` be an endofunctor):

```
class Functor m => Monad m where
  return :: a -> m a
  join :: m (m a) -> m a
  bind :: (a -> m b) -> m a -> m b

  join = bind id
  bind f = join . fmap f
```

The first two functions of the class are the η and μ natural transformations, respectively. Moreover it is usual to define a helper function called `bind` (aliased to the operator `>>=` with its arguments flipped) for reasons of both performance and ergonomics. Besides that, a monad in Haskell must follow rules derived from those stated in definition 2.4.1:

i. `bind return = id`
ii. `bind f .  return = f`
iii. `bind f .  bind g = bind (bind f .  g)`

Example 2.4.2. An example of a monad is the `Maybe` type constructor (as defined in example 2.2.3), which, as a monad, may be interpreted as a sequential computation that

might fail, where the failing can be considered a side effect, that is, an impure expression, from the point of view of the computation itself:

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
  return a = Just a

  bind _ Nothing = Nothing
  bind f (Just a) = f a
```

Example 2.4.3. By using the monad instance for `Maybe` defined above and the `safeHead` function defined in example 2.3.2, one may construct a potentially failing computation that gets the second element in the first list contained in a list of lists. The functions below demonstrate the use of the infix operator `»=` (as an alias to the `bind` function with its arguments flipped) as well as the of do-notation as syntactic sugar to the use of `bind`:

```
data User = User { _name :: String, _friends :: Maybe [User] }

firstFirstFriend :: User -> Maybe User
firstFirstFriend user = _friends user >>= safeHead >>= _friends >>= safeHead

firstFirstFriend_do :: [[a]] -> Maybe a
firstFirstFriend_do user = do
  friends <- _friends user
  firstFriend <- safeHead friends
  friendsOfFirstFriend <- _friends firstFriend
  safeHead friendsOfFirstFriend
```

The reader can observe that both variations of the `firstFirstFriend` function are equivalent, and they will fail as a computation (thus return a `Nothing` value) if either one of each subcomputation fails; that is, failed computations are propagated. This is the behavior of a monad: a context-producing (or side-effecting) sequential computation.

2.5 Comonads

All said, the most important concept in this theoretical chapter of this report is that of a *comonad*, to which the reader had already a short introduction in Section 1.2. Comonads are closely related to monads in category theory (though we shall not detail this relation) and have also been studied as a model for computations [Brookes & Geva, 1991], even though they model different types of computations as those that monads do [Uustalu & Vene, 2008].

Definition 2.5.1. In category theory, a **comonad** is a triple (D, ϵ, δ), where D is an endofunctor on a category $\mathcal{C}$ and ϵ and δ are natural transformations mapping D to the identity functor on $\mathcal{C}$ and D to a composition of D with itself (written $D \circ D$ or D^2), that is, $\epsilon : D \to 1_{\mathcal{C}}$ and $\delta : D \to D^2$ are natural transformations such that:

i. $T(\epsilon_A) \circ \delta_A = \epsilon_{T(A)} \circ \delta_A = 1_{T(A)}$.
ii. $T(\delta_A) \circ \delta_A = \delta_{T(A)} \circ \delta_A$.

Example 2.5.1. In Haskell, a comonad is given by a type constructor that implements the `Comonad` type class:

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

  duplicate = extend id
  extend f = fmap f . duplicate
```

The `extract` function represents the ϵ natural transformation, whereas `duplicate` represents δ from Definition 2.5.1 and `extend` is a helper function defined in terms of `fmap` and `duplicate`. Thus, for an instance of the `Comonad` type class to be considered a valid comonad in Haskell, it must obey the following rules derived from the category theoretical rules for a comonad:

i. `extend extract = id`
ii. `extract . extend f = f`
iii. `extend f . extend g = extend (f . extend g)`

Interpretation of comonads as computations. Comonads in functional programming provide means to structure context-dependent notions of computations [Uustalu

& Vene, 2008]. In this sense the reader can notice the duality in play between the interpretations of monads and comonads as well. In one hand, monads can model context-producing (or side-effecting) sequential computations, in the other, comonads can model context-dependent sequential computations.

Another valid (and not at all exclusive) interpretation is that comonads can represent computations within a context, that is, computations from a global context to a local value, which may be `extend`ed to the whole, thus producing a new global context. Two of the most acknowledgeable examples of this are that comonads form a good model for representing cellular automata or operations for image manipulation.

Example 2.5.2. One good example of a comonad is the `Stream` comonad, representing an infinite (lazy) stream. It is a type constructor together with an instance of the `Comonad` type class defined as such:

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

instance Functor Stream where
  fmap f (Cons a as) = Cons (f a) (fmap f as)

instance Comonad Stream where
  extract (Cons a _) = a
  duplicate stream@(Cons a as) = Cons stream (duplicate as)
```

The `Comonad` instance for the `Stream` type constructor allows us to extract the first value of a stream as well as to apply a function to all the whole stream, where this function has access to the future values of the stream and produces a single local value. This allows us, for instance, to define a function that adds the first and the second elements of the stream and then extend this function to the whole stream:

```
sumWithNext :: Num a => Stream a -> a
sumWithNext (Cons a (Cons a' _)) = a + a'

stream :: Stream Int
stream = Cons 1 (Cons 2 (Cons 3 ...))

stream' :: Stream Int
stream' = extend sumWithNext stream  -- Cons 3 (Cons 5 (Cons 7 ...))
```

Not only that, but, as stated already, comonads allow us to define context-dependent *sequential* computations. So one is able, for instance, to define a function that calculates

the first three elements of a stream *while having access to the whole context of the first three substreams*. Given that `=»` is an alias for `extend` (with its arguments flipped), such a function could be defined as the `first3` function below:

```
next :: Stream a -> a
next (Cons _ (Cons a _)) = a

first3 :: Stream a -> [a]
first3 stream =
  let stream2 = stream =>> next
      stream3 = stream2 =>> next
  in [extract stream, extract stream2, extract stream3]
```

Given the recent exposition and popularity this abstraction has been gaining in the past few years, some proposals for a syntactic sugar for comonads, equivalent to that of do-notation for monads have appeared. The most notable one is the proposal of Orchard & Mycroft [2012], whereby the `first3` function defined above could be written as:

```
first3_codo :: Stream a -> Stream [a]
first3_codo = codo stream =>
  stream2 <- next stream
  stream3 <- next stream2
  [extract stream, extract stream2, extract stream3]
```

2.6 Conclusions

Functional programming is a useful paradigm with strong relations to mathematics. This proximity provides reliable tools and properties for constructing programs. Category theory on the other hand has been shown to be a powerful tool for analysing concepts of functional programming. Furthermore, the category-theoretical framework can be used for building new abstractions for programming languages as well as for recognizing recurring patterns such as functors or monads. Despite its innate difficulty, working under the toolbelt of this field of mathematics is convenient for those doing research over programming languages.

The concept of comonads detailed in section 2.5 is known to be essentially used in the literature for modelling context-dependent computations; but in this thesis, another point of view on these structures will be explored: that comonads may represent spaces.

3 Comonads as spaces

The concept of a *comonad* plays a major role in this work. It has been shown that they have usually been used throughout the literature in order to abstract context-dependent computations, such as the calculation of the value of a cell given its neighbours in cellular automata, or the calculation of the value of a pixel of an image given its neighbours during an operation that blurs an image. In both examples, these context-dependent (local) computations, when applied (or *extended*) to their global context – the whole automata and the whole image, respectively –, produce new, modified contexts.

This notion of *context*, in some cases, can be understood as a neighbourhood, or as a pointed space, as of Freeman [2016b], that is, a space comprised by *points*, which are elements of a set X, where every point $x \in X$ has a neighbourhood, and where there is a distinct point x_0 called the base point. Pointed spaces can give an intuition to comonads, since the set X can be thought of as the type `a` encapsulated by a comonad, and the base point x_0 is the point retrieved by the `extract` operation. In its turn, the neighbourhood of a value in a comonad may be accessed through operations specific to each particular comonad (e.g. the `next` operation of the `Stream` comonad from Example 2.5.2).

With this intuition in mind, Freeman [2016b] proceeds to use comonads for representing (pointed) spaces of states of an application, where the current state of the application can be retrieved by extracting the value out of the comonad (with the `extract` function), and the neighbourhood of the current value gives the possible future states for the application. In this way, movements in this space would correspond to state transitions in an application, and distinct comonads may produce different spaces of states with different possibilities for movements.

3.1 Pairings of functors

Before introducing the reader to the model developed by Freeman [2016b], the concept of a *pairing* of functors must be presented.

The first appearance of such concept in the literature is through what Kmett [2008] defines as *"dual functors"*, defined in Haskell through the use of a `Dual` type class, which can be alternatively but equivalently defined as a type synonym, as in the `Pairing` type of Freeman [2016b]:

```
type Pairing f g = forall a b. f (a -> b) -> g a -> b
```

In this thesis, however, we have used the definition for a pairing given by Kmett [2008] but with a different name:

Definition 3.1.1. Given two Haskell functors `f` and `g`, a **pairing** between these functors is an instance of the following type class:

```
class Pairing f g | f -> g, g -> f where
  pair :: (a -> b -> c) -> f a -> g b -> c
```

A good intuition for a pairing of functors is that, when it exists, it says something about the structure of both functors when they are compared and put together: that their structures match in some way and can annihilate each other yielding, then, values of the types encapsulated by both.

Example 3.1.1. The most trivial example of a pairing is the pairing between the `Identity` functor (as defined in Example 2.2.2) and itself, which simply applies the values encapsulated by both functors to the supplied function:

```
instance Pairing Identity Identity where
  pair f (Identity a) (Identity b) = f a b
```

Example 3.1.2. Another good example of what a pairing could be is the pairing between the `((->) a)` and `((,) a)` functors, that is, between the function and tuple functors, respectively. It can be understood as the correspondence between curried and uncurried functions; that is, the tuple provides two values, where one can be used as input for the function – which, in turn, produces the wanted output value for the pairing function –, and the other fits as the second input for the pairing function:

```
instance Pairing ((->) a) ((,) a) where
  pair f g (a, b) = f (g a) b
```

By using the pairing defined above, one can, for example, implement the `uncurry` function from Haskell's standard library:

```
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry = pair ($)
```

3.2 Exploring comonadic spaces with pairings

One may recall from the introduction of this chapter – where the analogy between comonads and pointed spaces was presented –, that Freeman [2016b] makes use of comonads to represent the space of states of an application. And as such, it is desirable to be able to transition between the current state of an application and a possible future state, having, then, movements in this comonadic space.

Freeman [2016b] makes use of monads (detailed in Section 2.4) in order to express movements inside spaces described by comonads, and, by combining a comonad `w` with a monad `m`, a pairing between both *"gives a way to explore the data in a comonadic value"*.

To better understand the possibility of moving around in a space described by a comonad, we can think of the `duplicate` function from the `Comonad` type class (described in Example 2.5.1). The intuition behind the use of `duplicate` is that, taking the pointed space analogy as a reference, duplicating a comonad means adding another dimension to the space and transforming it in such a way that, in every point x of the space, now lies a replica of the space with x as the base point.

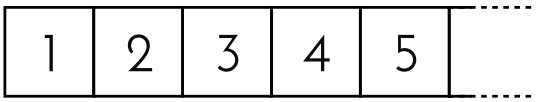
Example 3.2.1. To better illustrate this idea, recall the `Stream` data type and its `Comonad` instance from Example 2.5.2:

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

instance Comonad Stream where
  extract (Cons a _) = a
  duplicate stream@(Cons a as) = Cons stream (duplicate as)
```

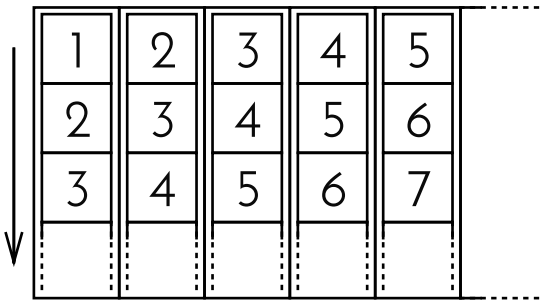
By applying `duplicate` to a value of a `Stream a` type, one obtains a stream of streams, where the first (or current) value of the stream is the original stream itself, and the future values are the old future streams themselves. An example in pseudo-Haskell is that:

```
stream :: Stream Int
stream = Cons 1 (Cons 2 (Cons 3 ...))

streams :: Stream (Stream Int)
streams = duplicate stream
  = Cons (Cons 1 (Cons 2 ...))
      (Cons (Cons 2 (Cons 3 ...))
        (Cons (Cons 3 (Cons 4 ...))
          ...))
```

That is, a stream can be viewed as an infinite tape, and duplicating this infinite tape yields an infinite grid such as in the figure below:



(a) A stream of integers can be thought of as an infinite tape.



(b) Duplicating a stream yields a displaced infinite grid.

Figure 3.1: Visualization of the duplication of a `Stream` comonad.

3.2.1 Comonads as spaces and monads as movements

By referring to Example 3.2.1 and to Figure 3.1b, the reader can observe that a movement in the space describe by the `Stream` comonad can be described as picking a specific point (a sub-tape) in the duplicated space. That is, thinking in terms of states of an application, by duplicating the current space of states, one obtains a space of all future spaces of states; and, in order to move to a future state one must simply pick one of the future spaces of states.

That's where pairings between comonads and monads come in. By pairing a comonad with a monad one is able to use monads to describe movements in the space, that is, in order to extract a new space from a (duplicated) space of spaces with the function `move` defined below:

```
move :: (Comonad w, Pairing m w) => w a -> m b -> w a
move space movement = pair (\_ newSpace -> newSpace) movement (duplicate space)
```

Example 3.2.2. By defining a monad which can produce a pairing against the `Stream` comonad we shall be able to move inside its space. We call this monad `Sequence` and define it as such (its `Functor` instance is omitted):

```
data Sequence a = End a | Next (Sequence a)

instance Monad Sequence where
  return = End

  bind f (End a) = f a
  bind f (Next next) = Next (bind f next)
```

In this way, the pairing between the `Stream` comonad and the `Sequence` monad can be defined as:

```
instance Pairing Sequence Stream where
  pair f (End a) (Cons b _) = f a b
  pair f (Next next) (Cons _ stream) = pair f next stream
```

Given these definitions, in order to displace the space defined by a `Stream` comonad, one must simply apply the function `move` defined in Section 3.2.1 to the space and a `Sequence` value describing a movement:

```
stream :: Stream Int
stream = Cons 1 (Cons 2 (Cons 3 ...))

third :: Stream Int
third = move stream (Next (Next (End ())))  -- Cons 3 (Cons 4 (Cons 5 ...))
```

Thinking of a `Stream` value as a space of states in an application, it would describe an application with a pre-defined sequence of states where the transition between states (given by `Sequence` values) only allow for jumping to later states in this pre-defined sequence, just as in a slide show application, where the user may only jump forward to the next slides.

3.3 Conclusions

Comonads may provide a useful abstraction for structuring applications. The notion of comonads as spaces of states of an application, allied to the notion of monads as movements in a comonadic space (through the use of a pairing) is the basis for structuring user interfaces using comonads. Different comonads provide different spaces (and different specific pairing monads), which, in turn, may be used to model different architectures for managing the state of an application.

Freeman [2016b] proposes several correspondences between specific comonads and known architectures for user interface paradigms. As an example of this correlation, we have seen an interpretation for the `Stream` comonad defined in Example 2.5.2 as an interface whose states are given by a pre-defined sequence (such as in wizard applications), and where the transitions between those states can be stated as simply choosing a next state. This could represent an interface for a slide show, for example.

We have now, then, the basic framework for describing and using comonads for the development of user interfaces.

4 User interfaces and comonads

Having defined comonads as an abstraction for the space of states in an application, we can extend this concept to use comonads as an abstraction for the space of user interfaces in an application – as they, themselves depend on the state of the application.

4.1 Basic model

User interfaces may be of any type depending on which application is being developed. They may be comprised of only text, pixels on a screen or HTML elements, for example.

As seen in Section 3.2.1, comonads may be a useful abstraction for the space of states of an application, where the current state of the application is always accessible through the use of the `extract` function of the `Comonad` type class. In this case, this space of states becomes the state of the application itself. This provides a useful abstraction for controlling how the states of the application may behave, as well as for talking about different architectures for user interfaces, the core of this work.

4.1.1 User interface components

A component of an application (or of a user interface) is generally thought of as a self-contained part of an application, that is, an distinct, individual part with its own internal state. A component may have, as well, its own distinct set of operations, which may expose its internal state or provide means to modify it.

Furthermore, user interfaces require user interaction. And as a response to a user interaction, it is common to change state of the application, which, in turn, produces a change in the interface being presented to the user. As we have seen that changes in the state of UI component may be abstracted through movements in its space of states, if a user interface component is represented by a comonad, movements, or *user actions* in this component, are represented by a pairing monad. In this way, when the user interacts with the interface, we want to be able to produce values of a pairing monad and use this value to update the state of the application and move around in the space of states of the component.

However, in order to dynamically update and maintain the state of an application throughout its execution we need some additional mechanisms.

4.1.2 Managing state

So far, we have seen comonads as an abstraction for spaces of user interfaces, where changing the position in this space changes the interface being presented (by changing the state). But how do we effectively manage the state of the user interface throughout the execution of an application?

From Wadler [1995] we know that, in pure functional languages, maintaining UI state usually demands the use of a monad. The user may not confuse this with using a monad for moving around in the space of states of an application described by a comonad.

Example 4.1.1. As an example of using monads to manage state in an application, consider a simple console application in which the following behaviour occurs in an infinite loop: the user is presented a counter which displays a number whose value is, initially, 0; then, the user is prompted to type any number, and, when a number is given, it gets added to the counter.

Displaying the counter to the user and prompting her to type a number through the console requires the use of the `IO` monad in Haskell. We can leverage this same monad to maintain the state of this interface through the use of the `IORef` type, which allows for mutable references in the `IO` monad, and through the `newIORef`, `readIORef` and `writeIORef` functions, which allows for creating and manipulating mutable references.

By using the `forever` function for infinitely repeating a monadic action, this simple application can be written as:

```haskell
ui :: IORef Int -> IO ()
ui ref = do
  counter <- readIORef ref      -- read value from the reference
  putStrLn (show counter)       -- print it to the console
  _ <- getLine                  -- reads a line from the console
  writeIORef ref (counter + 1)  -- increments the value of the reference

main :: IO ()
main = do
  ref <- newIORef 0
  forever (ui ref)              -- infinitely runs the ui function
```

The `ui` function simply reads the value of a mutable reference (containing an `Int` value) and prints it to the console. It then reads a number from the console's standard input and writes the read number to the mutable reference. This small process occurs, then, in an infinite loop.

4.1.3 Comonads as components of declarative user interfaces

By using a monad for managing the state of an application, we have, so far, all the elements needed to define a general model for user interface components.

A feature of declarative user interfaces is that they provide means to declaratively express *events*, that is, actions which may be executed when of specific user interactions with the application. By abstracting over the interface type, we may leave this detail to each specific rendering type, that is, to each specific type of declarative interface that may be presented to the user.

Thus, we can define a generic user interface in terms of the type `a` of the declarative specification of the interface and in terms of an `action` type representing the user actions this interface supports in reaction to user interaction:

```
type UI action a = (action -> IO ()) -> a
```

That is, given an event handler of type `action -> IO ()`, that is, a way to produce `IO ()` values when a user action (given by the `action` type) is dispatched by the interface in response to user interaction, a declarative specification of the interface (given by the type `a`) may be obtained.

Moreover, it is still possible to further abstract on this notion and eliminate the need for relying on the `IO` type. We may generalize these definitions to work with any monad `base` capable of managing the state of the interface:

```
type UI base action a = (action -> base ()) -> a
```

Therefore, basing ourselves on the notion of comonads as spaces presented in Chapter 3, specifically in Section 3.2.1, we may, then, be able to abstract user interface components as comonads containing user interfaces (of type `UI action a`). In this sense, this comonad represents the space of all possible user interfaces for the component, with the current interface, that is, the one being presented to the user, made accessible through the `extract` function.

The shape of this space, that is, the possibilities for moving around inside it, represent the operations of this component, and, as seen in Section 3.2.1, a pairing monad can effectively give a way of moving around. Hence, we would like to use a pairing monad as user actions in order to obtain the following definition for a user interface component:

Definition 4.1.1. Given the following `UI` type for a declarative user interface whose actions are given by a monad `m` and whose internal state is managed by the `base` monad:

```
type UI base m a = (m () -> base ()) -> a
```

A **user interface component** is given by the following type:

```
type Component base w m a = w (UI base m a)
```

which is equivalent to:

```
type Component base w m a = w ((m () -> base ()) -> a)
```

where `a` represents a declarative specification of an interface, `w` is a comonad representing the space of states of the component, `m` is a monad that pairs with the `w` comonad and represents user actions in the interface, and `base` is the monad used to manage the internal state of the component in an application. It must be noticed that the `m () -> base ()` type produces monadic actions in the `base` monad when a value of type `m ()`, that is, a user action or movement inside the space of states, is supplied.

In order to run this component, that is, in order to effectively use it in an application as to produce a user interface, we must provide a way to *explore* the space defined by the comonad `w`. This, in turn, needs an internal state managed by the `base` monad.

Definition 4.1.2. To **explore** a component given by a comonad (as a space of user interfaces) means exactly initializing an internal state in an application with the component itself, rendering (or presenting) to the user the current interface that can be extracted from the component, and, finally, providing means to update the internal state by moving around inside the component's space when a monadic action of type `m` is dispatched (by pairing the space and the action; see Section 3.2.1).

Thus, exploring a component means implementing a function of type:

```
explore :: (Comonad w, Pairing m w) => Component base w m a -> base ()
```

which is equivalent to:

```
explore :: (Comonad w, Pairing m w) => w ((m () -> base ()) -> a) -> base ()
```

Where `base` is the monad used for maintaining the component's state throughout the execution of the application.

Example 4.1.2. As now we have all the tools needed for writing proper applications where user interface components are represented by comonads, we shall rewrite the small application from Example 4.1.1 using the `Stream` comonad defined in Example 2.5.2:

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

As it is an infinite data type, we may define a function which, given an initial seed, is able to lazily generate this infinite structure:

```
unfoldStream :: s -> (s -> (a, s)) -> Stream a
unfoldStream initialState next = Cons a (unfoldStream nextState next)
  where
    (a, nextState) = next initialState
```

Using these definitions, we are able to construct a UI component given a data type that represents a declarative interface. We may define a declarative user interface in the console as an infinite loop of displaying a text message to the user and prompting for input. Thus, in each output-input cycle the user interface may be represented by the following `Console` data type:

```
data Console = Console { _text :: String, _action :: String -> IO () }
```

Therefore, the incrementing counter interface from Example 4.1.1 may be reproduced as a component whose space of user interfaces (or states) is given by the `Stream` comonad and movements (or actions) in this space are given by the `Sequence` monad. The space of interfaces may be defined, then, as an infinite stream of `Console` values:

```
counterComponent :: Component IO Stream Sequence Console
counterComponent = Cons (render 0) (Cons (render 1) (Cons (render 2) ...))
  where
    render :: Int -> UI IO Sequence Console
    render state = \send ->
      Console
        (show state)                    -- display the current counter value
        (\input -> send (Next (End ()))) -- move to next state when of user input
```

By using the `unfoldStream` function defined in this example to produce a lazily evaluated infinite `Stream` value, the infinite stream of user interfaces can be properly defined as:

```haskell
counterComponent :: Component IO Stream Sequence Console
counterComponent = unfoldStream 0 (\state -> (render state, state + 1))
  where
    render :: Int -> UI IO Sequence Console
    render state = \send ->
      Console
        (show state)                      -- display the current counter value
        (\input -> send (Next (End ()))) -- move to next state when of user input
```

And in order to run this component in an application, we must define an `explore` function following Definition 4.1.2. This function must initialize the component's internal state and provide an infinite loop of printing the interface's text to the user and prompting for input. By making use of the `IORef` type (as in Example 4.1.1), one is able to define such a function by simply extracting the current interface from the comonadic space, printing its contents to the console, prompting the user for input and handling user actions (given as a value in the pairing monad) by moving around in the comonadic space (through the use of pairings, as defined in Section 3.2):

```haskell
explore :: (Comonad w, Pairing m w) => Component IO w m Console -> IO ()
explore component = do
  ref <- newIORef component   -- initialize the reference with the initial space
  forever $ do
    space <- readIORef ref

    -- send receives an action dispatched by the UI
    -- and updates the state by moving around in the space
    let send action = writeIORef ref (move space action)

    let Console text action = extract space send  -- extract the current interface

    putStrLn text
    input <- getLine
    action input
```

The final application may, therefore, be written simply as:

```haskell
main :: IO ()
main = explore counterComponent
```

Example 4.1.3. A more complex example of a component of console user interfaces can be defined by using a different comonad. In general, user interfaces can be defined in terms of a state type `s` and a function which obtains an interface of type `a` based on

the current state. This description yields the `Store` comonad, which describes one of the most (if not the most) general space for user interfaces, indexed by the type `s`, where every position (or point) of this space contains a value of type `a`:

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

instance Comonad (Store s) where
  extract (Store f s) = f s
  duplicate (Store f s) = Store (Store f) s
```

In order to move around in the space described by this comonad, we can use the `State` monad:

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

instance Monad (State s) where
  return a = State (\s -> (a, s))
  bind f (State g) = State $ \s ->
    let (a, s') = g s
        State h = f a
    in h s'
```

This monad can be understood as a computation that, given an initial state of type `s`, produces a value of type `a` out of it and a new state. In this way, we may write two simple computations in the `State` monad for modifying a state. The first one applies a function to the current state, and the second one simply replaces the current state with a new one:

```
modify :: (s -> s) -> State s ()
modify transform = State (\state -> ((), transform state))

put :: s -> State s ()
put newState = State (\_ -> ((), newState))
```

With this small framework, we can use the notion of a comonad as a space of user interfaces to build a simple UI where we may have total control over the state of the interface. But, in order to move around in the space described by the `Store` comonad by using the `State` comonad, we are left to define, as of Section 3.2.1, a *pairing* between both:

```
instance Pairing (State s) (Store s) where
  pair f (State g) (Store get s) = f a (get s')
    where
      (a, s') = g s
```

Considering the `Console` data type from Example 4.1.2, and using the `Store` comonad and the `State` monad, we may finally define a simple component whose state, a list of strings, gets printed to the console in every cycle. When the user types a message in the console, it gets added to the list, but when the line typed by the user is empty, the state is reset to an empty list:

```
listComponent :: Component IO (Store [String]) (State [String]) Console
listComponent = Store render []
  where
    render :: [String] -> UI IO (State [String]) Console
    render list = \send ->
      Console
        ("I've received: " ++ show list)
        (\input -> send (if input == "" then put [] else modify (++[input])))
```

By using the `explore` function defined in Example 4.1.2, we may, then, run an application with this component as:

```
main :: IO ()
main = explore listComponent
```

4.2 General model with side effects

Useful applications need side effects. These are impure computations that interact with the real world, be it by printing a message to the screen, by reading a file from the disk, accessing a database, making an external HTTP request, etc. Pure functional programming, however, teaches that side effects in a program must be contained or minimized.

It is desirable, then, that in this abstraction for user interface components given by comonads, the component be able to produce side effects in response to user interaction. For example, we would like to be able to read the first line of a file from the disk when the user gives its name to the interface.

Recalling the definition of the `Component` type (from Definition 4.1.1), one may notice the use of the `base` monad for maintaining the internal state of the UI component:

```
type Component base w m a = w (UI base m a)  -- w ((m () -> base ()) -> a)
```

This `base` monad is used for side-effecting purposes: the maintenance of an internal state throughout the execution of an application. In this way, we may be able to make

use of this monad in order to produce arbitrary side effects alongside user actions (given by the `m` monad).

Definition 4.2.1. A user interface component that may produce side effects is defined similarly to a normal user interface component (from Definition 4.1.1). The main difference is in the fact that, in the case of the former, user actions may carry side effects. This can be done by redefining the `UI` type:

```
type UI base m a = (base (m ()) -> base ()) -> a
```

That is, producing user actions in response to user interaction may be understood as calling an event handler function with a (possibly) side-effecting monadic action in the `base` monad encapsulating a user action (or movement in the space of states of the component) given by the `m` monad.

In this sense, a side-effecting user interface component may be defined as:

```
type ComponentT base w m a = w (UI base m a)
```

which is equivalent to:

```
type ComponentT base w m a = w ((base (m ()) -> base ()) -> a)
```

With this change, the `explore` function must also be changed to:

```
explore :: (Comonad w, Pairing m w) => ComponentT IO w m a -> IO ()
```

From now on, the types defined above will be used instead of those from Definition 4.1.1.

Example 4.2.1. An example of a simple side-effecting component is a component for a console interface in which, when the user responds to the input prompt with a string p, a file of name p will be searched for in the current directory; if it exists, then its first line is read and printed to the screen. Furthermore, the component keeps track of whichever files have been already found and read and erases this record if an invalid file name is supplied. The user may refer to Example 4.1.3 for a similar component.

That being, a definition of this component may be given as:

```
import System.Directory

filesComponent :: ComponentT IO (Store [String]) (State [String]) Console
filesComponent = Store render []
  where
```

```
render :: [String] -> UI IO (State [String]) Console
render list send =
  Console
    ("Files read: " ++ show list)
    (send . tryReadFile)

tryReadFile :: String -> IO (State [String] ())
tryReadFile input = do
  fileExists <- doesFileExist input
  if fileExists
    then do
      contents <- readFile input
      putStrLn (takeWhile (/= '\n') contents)
      return (modify (++[input]))
    else do
      putStrLn "File not found"
      return (put [])
```

In addition to that, as the type for the specification of components has been changed, it is also needed to change the `explore` function used to run the component inside an application. The only difference from the one defined in Example 4.1.2 is that the function that handles user actions (the `send` function) must be defined in such a way that it extracts the user action (of type `m`) encapsulated in the `base` monad, thus executing the side effects in the `base` monad:

```
exploreT :: (Comonad w, Pairing m w) => ComponentT IO w m Console -> IO ()
exploreT component = do
  ref <- newIORef component   -- initialize the reference with the initial space
  forever $ do
    space <- readIORef ref

    -- send receives an action dispatched by the UI encapsulated in the base monad
    -- and updates the state by moving around in the space
    let send baseAction = do
          action <- baseAction
          writeIORef ref (move space action)

    let Console text action = extract space send  -- extract the current interface

    putStrLn text
    input <- getLine
    action input
```

With this, user interface components may now, besides changing its state by moving around in its space of states, produce side effects in response to user interaction. This modification to the original model of Freeman [2016b] is a step forward in the direction for using this technique for building real-world applications.

This modification does, however, present a limitation in the relationship between the side effects that may be produced and the user actions, which change the state of a component. This is limitation is in the fact that user actions cannot be interleaved with side effects. As side effects in response to user interaction are described by a side-effecting computation in a `base` monad, which, in turn, produces a user action when finished, the state of a component can only be changed when of the end of a side-effecting reaction.

4.3 Composing user interface components

Another desirable trait of user interfaces in real applications is the possibility of composing interfaces, that is, of having a parent component contain a child one inside itself and having access to the state of its child. This allows for great modularity and separation of concerns.

Freeman [2016b] has proposed the use of structures called *comonad transformers* in order to compose comonadic spaces (of user interfaces) and *monad transformers* to compose monadic actions (or movements inside the comonadic space).

Monad transformers

Monads have been an indispensable tool in pure functional programming since their introduction by Moggi [1991] and Wadler [1995]. They are mainly used to model computational effects and are commonly associated with a set of operations, for example, a state monad provides operations for retrieving, replacing and updating (with a function) the current state, whereas the IO monad may provide operations for accessing files or printing messages to the console. In this way, when structuring complex programs it is usual to want to combine multiple computational effects, and this must be done by combining monads as building blocks of computations.

The most used way of combining monads is through the use of *monad transformers*. Introduced by Liang et al. [1995], these structures are used to combine monads in a hierarchical fashion, building new complex monads by adding computational features of another monad to a pre-existing one, forming what is called a *monad transformer stack*.

This incremental approach is popular in functional programming, because monad transformers are easy to implement. Take for example the case where an application must be designed where it has some form of state and must account for failures. As we have seen in Section 2.4, specifically in Example 2.4.2 and Example 2.4.3, the `Maybe` monad can be used for expressing computations which may fail. Also, from Example 4.1.3 we know that the `State` monad can be used for expressing computations with internal state.

In order to build, then, an application that must account for failures while having an internal state, we would like to make use of some form of combination of both monads. That's where monad transformers come in:

Definition 4.3.1. As of Liang et al. [1995], a **monad transformer** is any type constructor `t` such that, if `m` is a monad, then `t m` also is. This can be expressed through the following type class:

```
class (Monad m, Monad (t m)) => MonadTrans t m where
  lift :: m a -> t (m a)
```

That is, a monad transformer is a type constructor `t` such that, if `t m` for any monad `m` is itself a monad, it is capable of embedding computations given in `m` into `t m`. Furthermore, in order to guarantee that this monad transformer effectively *combines* features of monads without changing the nature of an existing computation, the following laws must be followed for whatever instance of the `MonadTrans` type class (we have used subscripts to indicate to which monad the functions refer):

i. $\texttt{lift . return}_{\texttt{m}} = \texttt{return}_{\texttt{tm}}$

ii. $\texttt{lift (bind}_{\texttt{m}}\texttt{ f m)} = \texttt{bind}_{\texttt{tm}}\texttt{ (lift . f) (lift m)}$

That is, the first law ensures that, if an empty (or pure) computation is lifted to operate in the monad `t`, then it should yield an empty computation in `t`. The second law guarantees that embedding a sequence of computations into `t` is the same as first lifting them individually, and then sequencing them in the lifted `t` monad.

Example 4.3.1. As an example of the previous definition, recall the example of an application that must account for failures while having an internal state. If this application fails, its internal state must be rendered inaccessible, that is, the whole application enters in a failure state. In this way, we may want that the `Maybe` monad used to express potentially failing computations be, in some way, the most basic monad in this application. Thus, we can define a monad transformer based on it:

```haskell
newtype MaybeT m a = { runMaybeT :: m (Maybe a) }

instance Monad m => Monad (MaybeT m) where
  return a = MaybeT (return (Just a))
  bind f (MaybeT m) = MaybeT $ do
    maybeA <- m
    case maybeA of
      Nothing -> return Nothing
      Just a -> runMaybeT (f a)
```

The `MaybeT` type constructor is defined in such a way that it simply encapsulates a `Maybe` value inside another monad `m`. Monad transformers are usually defined in a way that the monad `m` it embeds into itself encapsulate, in some way, its own computations.

The `Monad` instance for `MaybeT m` guarantees that a pure computation from a value of type `a` can be constructed by simply encapsulating a pure computation of type `Maybe` inside the monad `m`. Furthermore it ensures that sequencing (through the use of `bind`) computations given in the `MaybeT m` monad is done by sequencing the encapsulated computations given in the `m` monad while retaining the potentially failing behaviour of the original `Maybe` monad.

We are, then, only left to define a way of embedding (through the use of `lift`) computations in the `m` monad into the `MaybeT m` monad:

```haskell
instance MonadTrans MaybeT where
  lift ma = MaybeT (fmap Just ma)
```

That is, lifting a computation from the `m` monad into `MaybeT m` means simply transforming the original computation in `m` into a computation that yields a successful computation of type `Maybe`.

Thus, by embedding computations in a `State s` monad into the `MaybeT` monad transformer, we can obtain the desired behaviour. Suppose, then, an application which has as its internal state the representation of second degree polynomial (its a, b, and c coefficients, such that the polynomial is given by $ax^2 + bx + c$). We want a simple computation in this application that is able to calculate real roots from the state of the program. That is, if the polynomial described in the current state of the program has real roots, the computation succeeds, the state is cleared and the calculated roots are returned. But in the case it does not have real roots, the whole computation fails.

In this way, we are left to define functions which may, then, retrieve the internal state of a computation in the `State` monad, and turn a `MaybeT m` computation into a failing one:

```haskell
get :: State s s
get = State (\s -> (s, s))

fail :: MaybeT m a
fail = MaybeT (return Nothing)
```

Hence, the computation described above can be written as:

```haskell
data Equation = Equation { _a :: Float, _b :: Float, _c :: Float, }

calculateRealRoots :: MaybeT (State Equation) (Float, Float)
calculateRealRoots = do
  Equation a b c <- lift get
  let delta = b*b - 4*a*c
  if delta < 0 || a == 0
    then fail
    else do
      put (Equation 0 0 0)
      return (((-b) + sqrt delta) / (2 * a), ((-b) - sqrt delta) / (2 * a))
```

Comonad transformers

If monad transformers are used to combine monadic values (or computations) given by different monads while preserving the structure of the computations in both monads, comonad transformers may be used to combine comonadic values (or spaces, the way they are used in this thesis) in such a way that the structure of the spaces described by both monads is preserved.

Comonad transformers are not found in the literature, as their use is much more restricted, perhaps a consequence of the restricted use of comonads as well. Nevertheless, Kmett & Menendez [2008] defines in a package for the Haskell language a definition of a type class for comonad transformers.

Definition 4.3.2. As of Kmett & Menendez [2008], a **comonad transformer** is any type constructor `d` such that, if `w` is a comonad, then `d w` also is. This can be expressed through the following type class:

```haskell
class (Comonad w, Comonad (d w)) => ComonadTrans d w where
  lower :: d (w a) -> w a
```

By using the notion of comonads as spaces, it is possible to think of a comonad transformer as a type constructor `d` such that, if `d w` for any comonad `w` is itself a

comonad, it is capable of extracting values out of subspaces given by `w` contained in the space defined by `d w`.

Moreover, in order to guarantee that this comonad transformer effectively *combines* the spaces defined by both monads without changing their nature and properties, the following laws must be followed for whatever instance of the `ComonadTrans` type class (we have used subscripts to indicate to which monad the functions refer):

i. $\texttt{extract}_{\texttt{m}}\ \texttt{.}\ \texttt{lower} = \texttt{extract}_{\texttt{tm}}$
ii. $\texttt{duplicate}_{\texttt{m}}\ \texttt{.}\ \texttt{lower} = \texttt{lower}\ \texttt{.}\ \texttt{extend}_{\texttt{tm}}\texttt{lower}$

Accessing child components from parents

By using comonad transformers for combining comonads, we may compose spaces of user interfaces, that is, components. It is possible, for instance, to define a parent component which, given any child component of arbitrary comonad `w` and monad `m`, encapsulates this child in its own space as a subcomponent. In order to do this, a comonad transformer must be used for representing the parent component's space.

A simple example of a comonad transformer is the `StoreT` comonad below, 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
```

Listing 4.3.1: The `StoreT` comonad transformer.

This data type allows for us to combine the space described by the `Store` comonad with that of any other comonad `w`. Also, its instance for the `ComonadTrans` type class lets us have access to the subspace in the current position:

```
instance ComonadTrans StoreT where
  lower (StoreT wf s) = fmap (\f -> f s) wf
```

From what has been said so far, it is known that, in order to use this comonad as a space of user interfaces, we need a pairing monad to describe user actions in this space. Hence, if the `State` monad pairs with the `Store` comonad, we may define a `StateT` monad transformer that pairs with the `StoreT` comonad transformer (its instance for the `Monad` type class has been omitted):

```haskell
data StateT s m a = StateT (s -> m (a, s))

instance Pairing m w => Pairing (StateT m) (StoreT w) where
  pair f (StateT get) (StoreT wg state) =
    pair (\(a, state') g -> f a (g state')) (get state) wg
```

Listing 4.3.2: The `StateT` monad transformer and its `Pairing` instance with the `StoreT` comonad transformer.

Example 4.3.2. Suppose we want to use the `listComponent` component given in Example 4.1.3 as the parent of another component given by a comonad `w`. In this way, we need to redefine `listComponent` in order to have it make use of the `StoreT` comonad defined above and to accept its child component as an argument:

```haskell
listComponent
  :: Component IO w m Console
  -> Component IO (StoreT [String] w) (StateT [String] m) Console
listComponent childComponent = Store renderComposed []
```

The main problem now is: how do we define `renderComposed`? From the definition of the `StoreT` data type in Listing 4.3.1, we know that it must have type `w (s -> UI IO (StateT [String] m))`, that is, it must be the `render` function from the original implementation of `listComponent` (in Example 4.1.3) encapsulated into the `w` comonad.

As we have access to the child component, which is a value of the `w` comonad, we may use it in order to have our desired `renderComposed` function. This could be done by applying the `extend` function from the `Comonad` type class to the `child` value.

Recall from Section 2.5 that the `extend` function takes as parameters a value `w a`, where `w` is a comonad, and a function of type `w a -> b`. This function represents a *comonadic computation*, that is, a computation that, given a comonadic context, is able to produce a value. And that's exactly what we need in order to have access to the child component while rendering the parent component:

```haskell
listComponent
  :: Component IO w m Console
  -> Component IO (StoreT [String] w) (StateT [String] m) Console
listComponent childComponent = Store renderComposed []
  where
    renderComposed = extend render child

    render child list = \send -> ...
```

If a comonad represents the space of all possible states of a UI component, extending a comonadic computation over this space makes it possible to capture the whole space withing this computation, that is, the computation may have access to all future states of this component. This can be seen also in the fact that the `render` function has access to the child component.

Besides simply using having access to the current interface of a child component, it is often desirable to be able to retrieve or modify the current state of a child component. By making use of pairings (as of Section 3.1) one is able to perform monadic actions in the context defined by a comonad, that is, we can apply user actions to components and retrieve their result. What leads us to the following definition:

Definition 4.3.3. To **select** a monadic action over a comonadic context means pairing both values (the monadic action and the comonadic context) and retrieving the value produced by the execution of the monadic action.

Thus, selecting a part of a comonadic space means retrieving the result of the application of a monadic action to a comonadic context by using the following function:

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

Example 4.3.3. With the `select` function we are able now to select specific parts of the space of states of a component by applying monadic actions to it. As an example, consider the `listComponent` from Example 4.1.3, whose type is:

```
listComponent :: Component IO (Store [String]) (State [String]) Console
```

If at some point of the code we may have access to this component, we are able to retrieve its current state (of type `[String]`) by applying the following `get` action:

```
get :: State s s
get = State (\s -> (s, s))

...

select listComponent get :: [String]
```

4.4 User interface architectures

The main contributions of the work of Freeman [2016b] is the proposition of the correspondence between comonads and user interfaces. The correspondence, however, goes even further: Freeman proposes that specific comonads may correspond to specific architectures (or paradigms) of user interfaces.

Some libraries or models for user interfaces define distinct architectures for handling and organizing user interfaces. These architectures define not only the way UI components may be organized or may communicate, but also define how their own internal state may be managed, accessed or modified (by the components themselves or by external components).

Handling these UI architectures with comonads allows us to abstract specific details of the implementation of user interfaces and talk about them in a higher level, only in terms of their architecture and, in turn, of the comonads that define them.

In this section we explain some of these correspondences between specific comonads and well-defined architectures for user interfaces.

4.4.1 A general model with the Store comonad

The `Store` comonad seen throughout this chapter (defined in Example 4.1.3) has been said to describe a very general model for user interfaces. Recall from its definition that it may be defined as a data type having a current state dictated by a type `s` and a way to obtain a user interface of type `a` based on the current state:

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

instance Comonad (Store s) where
  extract (Store f s) = f s
  duplicate (Store f s) = Store (Store f) s
```

In addition to that, its instance of the `Comonad` type class may be interpreted as making possible for extracting the current user interface (based on the current state) as well as for duplicating the space and, thus, obtaining in each point of the original space (the possible future user interfaces) a new space with the original point as its current interface and its own possible future interfaces.

A visualization of this comonad as a space where, from every point a user interface can be extracted is given in the figure below:

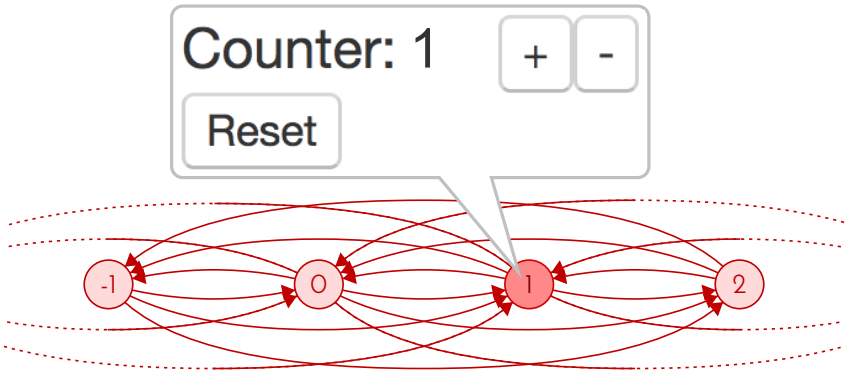


Figure 4.1: The `Store` comonad represents a pointed space indexed by a state type – in this case, the `Int` type. Movement in this space is free, restricted only by the state type.

In Figure 4.1, every point of the space (indicated by a circle with its corresponding state inside) contains a possible user interface, and can be accessed from any other point (represented by the pointed arrows). This is so because movements in this space can be expressed in terms of the `State` monad, which provides operations for replacing the current state for whichever value of the state type one wants, as well as for retrieving the current state:

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

put :: s -> State s ()
put newState = State (\_ -> ((), newState))

get :: State s s
get = State (\s -> (s, s))
```

In this sense, as seen in Section 4.3, specifically in Definition 4.3.3 and Example 4.3.3, if a component given by a `Store` comonad is a child component – that is, if another component has access to this component –, it is possible then, to retrieve the current state of this component (by `select`ing the `get` monadic action over its space) as well as to modify its current state (by `select`ing the `put` monadic action over its space).

Moreover, a parent component in this general architecture can be defined by using the `StoreT` comonad from Listing 4.3.1.

4.4.2 The Elm architecture with Moore machines

The Elm language [Czaplicki, 2012b] is a pure functional domain-specific language for building interactive applications for the Web. It defines its own architecture where the whole application, that is, the whole user interface is defined in terms of a state type, a type of user actions and two functions: one for rendering the current interface given the current state of the program, and one for updating the current state when of a user action in response to user interaction.

As proposed by Freeman [2016b], this architecture may be defined in terms of a Moore machine [Moore, 1956]. Disregarding the output of a Moore machine, that is, taking it as classic deterministic finite automata with no output alphabet, a Moore machine is a finite-state machine that can be defined by a set of states S, an initial state $S_0 \in S$, a finite set of inputs Σ called the input alphabet and a transition function $T : S \times \Sigma \to S$, mapping a state and an input value to the next state.

This can be modelled as the following recursive data type:

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

Listing 4.4.1: The `Moore` type of Moore machines.

Where `i` is the type of inputs corresponding to the finite set of inputs Σ and `a` is the type of states corresponding to the set of states S. The `Moore` constructor builds a Moore machine from an initial state of type `a` and a transition function, which, given an input value of type `i`, returns a new machine with the next state.

However, what most matters in this section is the fact that the `Moore` type admits a comonad:

```
instance Comonad (Moore i) where
  extract (Moore a _) = a
  duplicate w@(Moore a t) = Moore w (duplicate . t)
```

When this comonad is interpreted as a space of user interfaces, we can visualize this space as an infinite tree of interfaces, where every node of the tree contains an interface. Branching in this tree is given by the `i` type of inputs, as in the following figure:

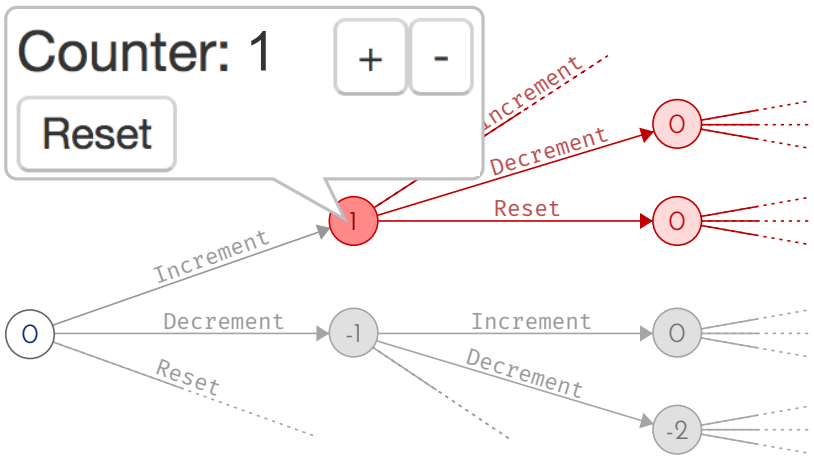


Figure 4.2: The `Moore` comonad represents a space such as an infinite tree. There is a root point and one can navigate to the child nodes with branching given by a type of inputs.

Movements in this space, then, can be thought of as sequences of values of the input type `i`. This analogy yields the following data type which is a monad and pairs with the `Moore` comonad (its monad instance is ommited):

```
data Actions i a = NoAction a | Action (i, Actions i a)

instance Pairing (Actions i) (Moore i) where
  pair f (NoAction a) (Moore b _) = f a b
  pair f (Action (i, r)) (Moore _ t) = pair f r (t i)
```

Listing 4.4.2: The `Actions` monad and its pairing with the `Moore` comonad.

Producing a single action given a value of the input type `i` may be done through the use of the following function:

```
action :: (i, a) -> Actions i a
action (input, a) = Action (input, NoAction a)
```

Listing 4.4.3: The `action` function embeds a single action of the input type `i` into the `Actions` monad

Example 4.4.1. In order to define a user interface component using the `Moore` comonad, we must, first, define a set of user actions by defining an input type. Suppose we want

to create the counter component illustrated in Figure ??. It can be seen that it defines three user actions, namely: incrementing, decrementing or resetting the counter. Thus, our input data type can be defined as:

```
data Input = Increment | Decrement | Reset
```

Now, the only task left is to define an infinite tree of user interfaces, given by a value of the `Moore Input` type. For that we shall use a helper function that, given an initial state of type `s`, and a function that, given a state of type `s`, produces a value of type `a` and transition function, which, given an input of type `i`, produces the next state:

```
unfoldMoore :: s -> (s -> (a, (i -> s))) -> Moore i a
unfoldMoore state next = Moore a (\input -> unfoldMoore (transition input) next)
  where
    (a, transition) = next state
```

With this function, we are now able to define our counter component using the `Moore` comonad:

```
mooreComponent :: Component base (Moore Input) (Actions Input) a
mooreComponent = unfoldMoore 0 (\model -> (render model, update model))
  where
    update :: Int -> Input -> Int
    update counter Increment = counter + 1
    update counter Decrement = counter - 1
    update _ Reset = 0

    render :: Int -> UI base (Actions Input) a
    render counter = \send -> ...  -- declarative specification of the interface
```

From the definition of the user interface of the previous example, it can be noticed that the internal state of that component (of type `Int`) is private. That is, no external component can have access to it. In this sense, components defined in terms of the `Moore` comonad cannot be composed.

So, in order to build complex applications, one must write a single large state machine, just as in the Elm architecture. The similarities with that architecture go further: there is a type of user actions and an update and render functions. In this sense, it can be said that the `Moore` comonad simulates the Elm architecture.

5 Implementing a simple application

The improvements made in this thesis to the original model of Freeman [2016b] have been implemented in order to allow for the verification of the suitability of the formal model to the building of real-world applications. Thus, we have extended the original model to support user interfaces which produce side effects as well as a hierarchical organization of UI components through the use of monad transformers (as proposed in Freeman [2016b]).

Therefore, in order to validate the developed improvements, a Web application for task management has been developed Xavier [2017]. This application produces side effects in order to save the tasks created by the user in the Web browser's local storage and has been developed with two hierarchically organized components.

5.1 The application

A real-world application with user interfaces is typically comprised of multiple UI components (or modules). These components abstract specific self-contained parts of the interface and may be hierarchically organized. This organization of components allow for great flexibility and growth of complexity with ease. Furthermore, for real utility, complex applications must interact with the real world through side effects, be them a connection with a database, a remote HTTP call or a simple local logging of actions.

We propose and implement in this chapter a contrived but complete example of what a real-world application might be. Moreover, we implement two versions of the same application – that is, with exactly the same outcome for the end user both in terms of interaction and experience – in order to verify the correspondences between comonads and user interface paradigms detailed in Section 4.4.

A task management application. The application here detailed is a fabricated version of a simple task management application. As such, it must allow for the end user to view, insert, remove or mark tasks as completed. In addition to that it must display a counter for how many tasks have already been marked as completed.

We have chosen to separate the application in two distinct components, namely the `App` and the `Tasks` component. As we have not yet been able to devise a comonad representing a list of other comonads, we are not able to construct a list of tasks by

combining multiple small hypothetical `Task` components. Thus, the `Tasks` component must be the smallest fragment in our application.

In this sense, the `App` component shall contain an input field where the user may type text for the insertion of a new note. By hitting a confirmation key (the return key in physical keyboards), the text inserted must be used to create a new task (placed as the first one in the list), and the input field must be cleared. The `App` component must also display a counter with the number of completed tasks. In this way, the `App` component must, somehow, have access to this data, which shall be contained in the `Tasks` component, which shall be, itself, a child of the `App` component.

In its turn, the `Tasks` component must simply contain a list of tasks in its internal state. It must display each task separately, as well as a button next to them that, when clicked, removes its accompanying task. In addition to that, when the user clicks on a task, its status of completion must be toggled: if the task is marked as done, it is, then, unmarked, or else, it is, then, marked as done. This component must, in this sense, have parts of its state exposed or encapsulated, such as the number of tasks marked as done, which may be accessible from external components. Moreover, it must provide ways for an external component to modify its state, so that new tasks can be added.

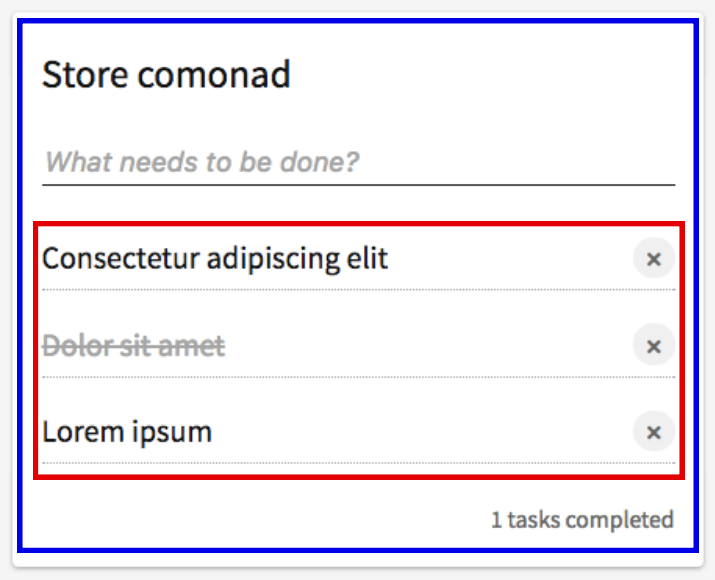


Figure 5.1: Components of the application: the `App` component framed in blue and the `Tasks` component in red.

.

5.2 The PureScript language

The language chosen for the implementation of this application is the PureScript language [PureScript, 2013]. PureScript is a pure functional language that is deeply inspired and similar to Haskell. It generates JavaScript code and can, then, be executed in a Web browser and, consequently, be used for the development of applications with user interfaces for the web.

The ecosystem of the PureScript language contains several useful libraries for the development of UI-based applications, as well as bindings for well-known JavaScript libraries. In order to be able to render our user interface in a Web browser, we have used the `purescript-react` library which exposes bindings for the *React* library [Facebook, 2013], a library for declarative rendering of user interfaces. This library is used solely for this purpose of having a way to declaratively render our user interface.

In this chapter, however, we shall display source code in the Haskell language. The API for the `purescript-react` library will be shortly explained in Section 5.4 with Haskell code.

5.3 The React library

React [Facebook, 2013] is described as a *"declarative, efficient, and flexible JavaScript library for building user interfaces"*. In this thesis, we make use of this library in order to be able to declaratively render HTML elements to the Web browser, and to be able declaratively specify responses to user actions.

Being a component-based library, our application will be created as a single React component (one component for each one of the two versions of the application). A component in the React library is simply a self-contained piece of interface that has an internal state, which is maintained by the library's runtime. When this internal state is modified, a user-defined `render` function is called. This `render` function takes the new internal state of the component and produces a declarative specification of what to render to the interface – in this case, HTML elements given by a `ReactElement` type.

The important details of this library – which are common to most libraries for declarative rendering – are that each component is defined in terms of an internal state and a rendering function. However, in the declarative specification of what to render to the user (produced by the rendering function), there may be properties of elements which provide ways to modify the component's state in reaction to user inputs. We call these

properties *event handlers*.

In the `purescript-react` library, values of the `ReactElement` type are constructed through the use of functions from the `React.DOM` module. These functions take a list of properties for the element as their first argument and a list of child `ReactElement`s as second argument. It may be noticeable, from the example below, that these functions are named after the HTML tags to which they correspond, e.g. `button` for the `<button>` tag, etc.

Example 5.3.1. An example of a simple component defined using the `purescript-react` library is that of a *counter* component. It contains a single button, which, when clicked by the user, increments the number that is displayed as its label. The code below is written in pseudo-Haskell and mimics the API of the `purescript-react` library (unimportant details were left omitted):

```
import qualified React as R
import qualified React.DOM as D
import qualified React.DOM.Props as P

counterComponent :: R.ReactClass
counterComponent = R.createClass (R.spec initialState render)
  where
    initialState = 0

    render :: Int -> R.ReactElement
    render state =
      D.button
        [ P.onClick (R.writeState (state + 1)) ]
        [ D.text (show state) ]
```

One can observe that the rendering function (`render`) takes a value of the type of the state of the component and returns a value of the `ReactElement` type. Also noticeable is the fact that event handlers (such as the `onClick` function) take `IO ()` actions as arguments, which are executed when the corresponding event is received. In this example, then, when the user clicks the button, the state of the component is updated.

In this sense, React components may be interpreted as a `Store` comonad, as seen in Section 4.4.1.

Exploring comonadic user interfaces with React

Given this way of creating React components from a specification (an initial state and a rendering function), we may use this library to explore comonadic user interfaces, that is, user interfaces defined as a comonad representing a space.

As seen in Section 4.4, a general model for a function that allows for the exploration of a space of user interfaces (defined by a comonad) is given by:

```
explore :: (Comonad w, Pairing m w) => Component IO w m a -> IO ()
```

The return type `IO ()` is used as an abstraction for the act of presenting the generated interface to the user. In the case of the React library, we can use the very definition of a React component (through the `ReactClass` type) in order to perform this presentation.

Hence, a function for exploring a comonadic space of user interfaces by rendering the generated interfaces through the React library may be given by a simple adaptation of the function used to create a React component (in Example 5.3.1) and of the function used to explore a comonadic space of console-based interfaces (in Example 4.2.1):

```
import qualified React as R

type ReactComponent w m = ComponentT IO w m R.ReactElement
type ReactUI m = UI IO m R.ReactElement

exploreReact :: (Comonad w, Pairing m w) => ReactComponent w m -> R.ReactClass
exploreReact space = R.createClass (R.spec space render)
  where
    render state = extract state (send state)

    send state baseAction = do
      action <- baseAction
      R.writeState (move state action)
```

5.4 Implementation on two different architectures

As seen in Section 4.4, using different comonads in the definition of user interface components may yield different architectures. For example, by using the `Store` comonad one may obtain a general architecture for user interfaces, similar to that of the React library. Or by using the `Moore` comonad (as of Section 4.4.2), an simple architecture similar to the Elm architecture [Czaplicki, 2012a] may be obtained.

A common point of the comonads mentioned in Section 4.4, though, is that they all provide means for structuring user interfaces with the help of a function which, given the current internal state of the component, produces an interface. By using the React library for rendering the interfaces as of Section 5.3, this function is equivalent to the following `render` function:

```
render :: state -> ReactUI m
```

Where `state` is the type of the component's internal state and `m` is the monad of user actions produced by the component – or movements inside its space of states. In this way, defining a component can be thought of as implementing values equivalent to the `component` value below:

```
component :: forall w m. ReactComponent w m
component = ...
  where
    render :: state -> ReactUI m
```

Listing 5.4.1: The general form of a component given in any of the architectures here presented.

Where the definition of the component itself is a value of the comonadic type `w` representing the space of user interfaces which has access to the local `render` function.

In this sense, every user interface component defined in this thesis will be of the form of the `component` function above, including the two versions of the `App` and `Tasks` components, which shall be presented in the following sections.

The `App` component, however, does present a peculiarity in the sense that it is a parent component; that is, its space of interfaces is a combination of its own space of user interfaces and that of its child component, namely the `Tasks` component.

For the case of a parent component, its child component (in its own current state) must be available to the parent's `render` function, just as its state is. This can be done by applying the `extend` function (from the `Comonad` type class) to the child component, which allows for performing, over all possible UIs this component may produce, a comonadic computation, that is, a computation which produces a value out of some context; in this case, a user interface out of a space of states:

```
parentComponent :: ReactComponent childW childM -> ReactComponent w m
parentComponent childComponent = ...
  where
    renderComposed :: state -> ReactUI m
    renderComposed = extend render childComponent

    render :: ReactComponent childW childM -> state -> ReactUI m
```

Listing 5.4.2: The general form of a parent component given in any of the architectures here presented.

The intuition behind extending the render function (with the `extend` function) over the child component's space, is that it makes it possible to capture all future states of the child component, that is, extending a function over a component yields access to its whole possible future. This can be seen also in the fact that the `render` function has access to the child component.

The domain. The application was created as a Web page containing two React components, namely the two versions of the application, each one having a distinct comonad and, therefore, architecture. These two versions do share, however, the same domain, and, in spite of this, we have defined a few data types to contain the definitions for the domain of the applications: a data type for a list of tasks and a data type for the rest of the state of the application (namely the state of the input field for new tasks and the id of the last task inserted).

```
data Task = Task { _taskId :: Int, _taskDescription :: String, _taskDone :: Bool }
type TasksModel = [Task]

data AppModel = AppModel { _appField :: String, _appLastId :: Int }
```

We have also defined the initial value for the state of the `App` component to be the same throughout the three versions. The initial value for this component depends on an initial list of asks and has its input field cleared:

```
appInit :: TasksModel -> AppModel
appInit tasks = AppModel "" (length tasks)
```

We shall not detail in this section the machinery for saving and retrieving tasks from the browser's local storage, as these implementation details are somewhat unrelated to

the main subject of this work. It is already known from Section 4.2 how side effects may be produced in response to user interaction.

The reader can refer to the source code for the application in the reference to Xavier [2017] in the Bibliography section.

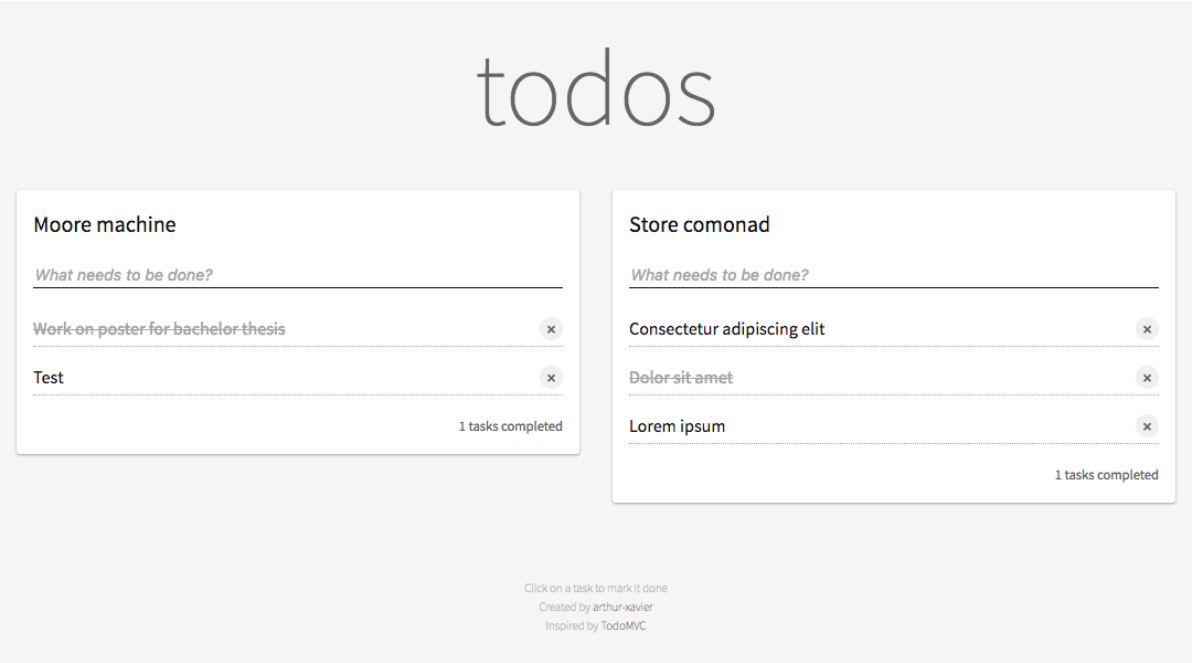


Figure 5.2: Screen capture of the developed applications where every card represents an architecture associated with a specific comonad.

5.4.1 React architecture

As stated in Section 4.4.1, the architecture of the React library – that is, an architecture where every possible state is accessible from any other state and where every component may freely access or modify the state of any other – is described with the `Store` comonad and the `State` monad, both defined in Section 4.4.1.

The `Tasks` component. In this architecture, the space of states for the `Tasks` component is, then, a `Store` comonad, and movements (or actions) in this space are given by the `State` monad:

```
type TasksSpace = Store TasksModel
type TasksAction = State TasksModel
```

As the space of user interfaces for the `Tasks` component is given by the `Store` comonad, it may be defined as just a `Store` comonad with an initial state and a rendering function, which takes the current state and produces a declarative description of the interface. That is, the proper `Tasks` component, may, then, be defined following the general form for a component defined in Listing 5.4.1:

```
type TasksComponent = ReactComponent TasksSpace TasksAction

tasksComponent :: TasksModel -> TasksComponent
tasksComponent initialState = store render initialState
  where
    render :: TasksModel -> ReactUI TasksAction
    render model = \send -> ...  -- declarative description of the interface
```

The user actions (or movements inside `TasksSpace`) in this component are values of the `TasksAction` type, which is, in fact, the `State` monad. These actions are dispatched through the use of the `send` function (as seen in Section 4.1) inside event handlers of React elements (as in the use of the `onClick` function in Example 5.3.1). When an action is dispatched and handled, the state of the application is, then, updated, by pairing the monadic action with the comonadic space (as shown in Section 5.3).

As an example of actions in the `Tasks` component, we have below the definition of the action for for removing a task:

```
removeTask :: Int -> TasksAction ()
removeTask id = modify (filter ((/= id) . _taskId))
```

The `App` component. The outermost component of the application, the `App` component is the parent of the `Tasks` component, that is, its space may be defined as a composition of both spaces and its actions may be defined as a composition of the action types of both components. For that we may use the `StoreT` and `StateT` monad transformers defined in Section 4.4.1. Thus, the type for the space and the type for user actions in the `App` component can be defined as:

```
type AppSpace = StoreT AppModel TasksSpace
type AppAction = StateT AppModel TasksAction
```

The `App` component itself, may, then, be defined as a value of the `StoreT` comonad containing a state of type `AppModel` and a function for retrieving user interfaces (of type `ReactUI AppAction`) out of the current state. By following the general model for a parent component defined in Listing 5.4.2:

```
appComponent :: TasksModel -> ReactComponent AppSpace AppAction
appComponent tasks = StoreT (extend render (tasksComponent tasks)) (appInit tasks)
  where
    render :: TasksComponent -> AppModel -> ReactUI AppAction
    render child model = \send -> ...  -- declarative description of the interface
```

The reader can observe that, by simply extending – with the `extend` function – the `render` function over the `Tasks` component, it is possible to capture all its future internal states, that is, extending a component yields access to its whole possible future. This can be seen also in the fact that the render function has access to this component.

However, some internal machinery (abstracted through a very simple `liftComponent` function) must be used in order to have the state of the child component updated alongside the state of its parent. This function will be omitted in this report, but the reader may have access to it through the source code of Xavier [2017].

An example of communication between both components can be given with the user action for creating a new task:

```
createTask :: AppAction ()
createTask = do
  model <- get
  lift $ modify (++ [Task (_appLastId model) (_appField model) False])
  put (AppModel "" (_appLastId model + 1))
```

By making use of the `MonadTrans` instance for the `StoreT` type, we can lift actions from the parent component (`App`) to the child component (`Tasks`) with the `lift` function.

5.4.2 Elm architecture

As detailed in Section 4.4.2, the architecture of the Elm library is an architecture where the whole application must be defined as a single user interface component. Furthermore, the application is defined in terms of a type of user inputs, an update function (for changing the application state given the current state and a user input), and a render function, which produces the current interface given the current state.

It has been shown that the `Moore` comonad (defined in Listing 4.4.1) simulates the Elm architecture by defining a tree-like space of user interfaces. Furethermore, the `Actions` monad (defined in Listing 4.4.2) can be used to express movements (or user actions) in the space of interfaces defined by the `Moore` comonad.

From the fact that, in the architecture described by the `Moore` comonad, there is no way to use multiple components in an application, a type for the global state of the

application, that is, a type which contains all the state information of the application, has to be defined:

```
data GlobalModel = GlobalModel
  { _globalField :: String
  , _globalLastId :: Int
  , _globalTasks :: TasksModel
  }
```

And, given an initial list of tasks, the initial state of the application is given by the following function:

```
globalInit :: TasksModel -> GlobalModel
globalInit tasks = { field: "", uid: length tasks, tasks }
```

The `Tasks` component. In this architecture, the space of states for the `Tasks` component is, then, a `Moore` comonad, and movements (or actions) in this space are given by the `Actions` monad. As the whole application is given by a single component, we shall not define a type for the space of this component.

```
data TasksInput = AddTask Task | RemoveTask Int | ToggleDone Int
type TasksAction = Actions TasksInput
```

In fact, the `Tasks` component will not be a component itself, but only provide abstractions for separation of concerns, thus, defining a way of rendering its own interface based on a part of the global model, as well as a function for updating a part of the state of the whole application:

```
tasksComponent :: TasksModel -> ReactUI Action
tasksComponent model = \send -> ...  -- declarative specification of the interface

tasksUpdate :: TasksModel -> TasksInput -> TasksModel
```

The `App` component. The `App` component, in the architecture given by the `Moore` comonad functions as the whole application itself, that is, its space define the space of states of the whole application (including that of the abstracted components).

```
type AppInput = ChangeField String | IncrementLastId | TasksAction TasksInput
type AppAction = Actions AppInput

type AppSpace = Moore AppInput
```

```haskell
appComponent :: TasksModel -> ReactComponent AppSpace AppAction
appComponent tasks = unfoldMoore (globalInit tasks) step
  where
    step model = (render model, update model)

    update :: GlobalModel -> AppInput -> GlobalModel
    update model input = ...

    render :: GlobalModel -> ReactUI AppAction
    render model = \send -> ...  -- declarative specification of the interface
```

In order to embed the `tasksComponent` interface inside the interfaces given by `appComponent`, the user actions dispatched by it must be encapsulated in the `TasksAction` constructor of the `AppInput` type by using a function of type:

```haskell
mapAction :: (TasksInput -> AppInput) -> TasksAction a -> AppAction a
```

That is, given a way to convert values of type `TasksInput` to values of type `AppInput`, we may transform movements in the subspace of the `tasksComponent` interface (given by the `TasksAction` monad) into movements in the global space of `appComponent`.

6 Conclusion

A task management application for the Web browser has been created using an extension of model of Freeman [2016b] here devised (see Chapter 4). This extension provides support for the production of side effects – saving the tasks a user has created in the Web browser's local storage – as well as for composition and communication between components – the shell of the application has access to the data contained by the component that manages the tasks and may display, for instance, how many tasks have already been completed.

Furthermore, the application has been developed in three separate versions, whereby each version is developed using a different user interface architecture (under the same general model), thus demonstrating the some of the correspondences between UI paradigms and comonads proposed in Freeman [2016b].

Moreover, apart from the high complexity of the concepts involved in understanding and developing this model, given a set of helper functions and combinators, the development process of this application has shown itself to be comfortable, practical and, above all, pragmatic.

6.1 Results and discussion

As aforementioned, the main contribution of this work is an extension of the model of Freeman [2016b] for representing user interface components as comonads to support side-effecting user interfaces as well as composition (with communication) of components.

The original model uses the concept of *comonads* in order to represent user interface components as a space of all possible user interfaces, where the shape of this space is given by the component's structure and internal state. Furthermore, user actions in this UI component are represented as movements in this space of interfaces, where the current position in the space defines the interface being presented to the user. These movements are, in turn, defined by *monads*. Furthermore, the concept of a *pairing of functors* has been explained in Section 3.1 as a means to properly navigate in this space of user interfaces (Section 3.2) by combining specific comonads (defining spaces) with specific monads whose structure can be used to define movements in these spaces.

The complexity of the concepts involved in this model, however, may impose a great difficulty on its adoption for the development of real-world applications. One limitation

of the technique, in the way it is presented in this thesis, is that it is considerably difficult to generate or discover monads that may be used to describe movements in a space defined a comonad; that is, it is difficult to find *pairings* between comonads and monads.

Side-effecting user interfaces. The addition to the original model of the possibility for a component to produce side effects in response to user interaction has been made through the encoding of user actions (or movements in the space of user interfaces) in a base side-effecting monad. This concept is further detailed in Section 4.2.

A problem identified with this simplistic approach to side-effecting UIs is that user actions (defining state changes in the component) cannot be interleaved with side-effecting computations. That is, a user action is only produced at the very end of a side-effecting computation. This disables the possibility, for instance, to have multiple (asynchronous) state changes in a component in reaction to user interaction.

Composing user interfaces. Already proposed by Freeman [2016b], the use of *comonad transformers* (allied to the used of *monad transformers*) was used in order to hierarchically compose user interface components (in Section 4.3). Apart from that, we have discovered ways in which a parent component may communicate with a child component by accessing parts of its internal state or even modifying it or the user actions dispatched by it.

Moreover, this communication between components is limited by the comonad used in their definition. That is, the comonad must provide operations which allow for accessing or modifying of its state. An example of this lies in the fact that the `Moore` comonad (detailed in Section 4.4.2) can be used to define UI components with private, unaccessible state. This comonad was, then, used as a representation of the Elm architecture [Czaplicki, 2012a], which exhibits the same kind of behaviour.

However, a consequence of the difficulty in finding monads or pairings for specific comonads (such as one for a list of comonads) prevents the use and experimentation with more complex forms of composition.

Comonadic interfaces for real-world applications. In addition to the previously mentioned contributions, this work properly validates the suitability of this general abstract model of user interfaces for the development of real-world complex applications. By developing a small but sufficiently complex application that makes use of the extensions to the original model here developed, we have been able to verify that the correspondences

between distinct UI architectures (or paradigms) and specific comonads proposed by Freeman [2016b] still hold when an application grows in complexity and size.

6.2 Future work

There is very much room for improvement and further research in this subject matter, perhaps due to the inherent complexity of the theme. While the model here developed has been proven useful and practical for the development of applications with user interfaces, there are a number of possible improvements and further developments, ranging from the most theoretical to the most practical ones.

Generate monads for any comonad. As a first and more immediate possibility for future work, we plan on further improving the expressiveness and ergonomics of the technique here presented. By using a technique proposed by Kmett [2011] (and used by Freeman [2016b] in the original model), we shall be able to to eliminate the necessity of finding or developing specific monads and pairings for given comonads, something that has been shown an extremely laborious task. This change in the model would also provide the means needed to correct two limitations of the current model, namely: the inability to express non-hierarchical composition of components (through a comonad that describes a list of comonads), and the inability of interleaving side effects with user actions.

A library for building user interfaces. The model here developed has its practicality and utility praised mainly due to the abstractions devised on top of it to the development of the small application from Chapter 5. We believe that collecting abstractions, helper functions and user interface combinators in the form of a software library could not only be fairly straightforward but produce an artifact that could be of remarkable use to the community and industry for the development of UI-based applications.

Implementation with other rendering libraries. Furthermore, another possibility to further verify the potential ubiquity of this model is to implement other applications that abstract their user interfaces with it. However, for better results we suggest the use of different ways, libraries or techniques for presenting the interfaces to the user, as in this thesis we have made use of standard `IO` actions (for writing to the console) and of the React library (for rendering HTML based interfaces on a Web browser).

Composition of components with Day convolution. Freeman [2016a] has made use of the representation in the Haskell language of a category-theoretical structure called the *Day convolution* in order to horizontally (that is, non-hierarchically) combine user interface components. The use of this structure can lead to the development of a comonad for describing lists of UI components, as of Freeman [2017].

Formalization of pairings from a proper theoretical viewpoint. The definition of a *pairing of functors* given in Section 3.1 is somewhat informal. It is given in terms of a Haskell type class and provides no laws or whatever non-contextualized interpretation. In spite of this, a properly theoretical work should be done over this structure in order to clarify its meaning and definition.

Bibliography

Asperti, A. & Longo, G. (1991). *Categories, types, and structures: an introduction to category theory for the working computer scientist*. MIT press.

Barendregt, H., Dekkers, W., & Statman, R. (2013). *Lambda calculus with types*. Cambridge University Press.

Bird, R. & Wadler, P. (1988). *Introduction to functional programming*, volume 1. Prentice Hall New York.

Bragge, M. et al. (2013). Model-view-controller architectural pattern and its evolution in graphical user interface frameworks.

Brookes, S. & Geva, S. (1991). *Computational comonads and intensional semantics*. Carnegie-Mellon University. Department of Computer Science.

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

Czaplicki, E. (2012b). Elm programming language. `http://elm-lang.org`. Accessed: 2017-08-16.

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

Freeman, P. (2016a). Comonads and day convolution. `http://blog.functorial.com/posts/2016-08-08-Comonad-And-Day-Convolution.html`. Accessed: 2017-08-16.

Freeman, P. (2016b). Comonads as spaces. `http://blog.functorial.com/posts/2016-08-07-Comonads-As-Spaces.html`. Accessed: 2017-08-16.

Freeman, P. (2017). Comonads for optionality. `http://blog.functorial.com/posts/2017-10-28-Comonads-For-Optionality.html`. Accessed: 2017-11-01.

Hughes, J. (1989). Why functional programming matters. *The computer journal*, 32(2):98–107.

Kmett, E. (2008). The Cofree comonad and the expression problem. `http://comonad.com/reader/2008/the-cofree-comonad-and-the-expression-problem/`. Accessed: 2017-08-16.

Kmett, E. (2011). Monads from comonads. `http://comonad.com/reader/2011/`

monads-from-comonads/. Accessed: 2017-08-16.

Kmett, E. A. & Menendez, D. (2008). comonad: Haskell 98 comonads. https://hackage.haskell.org/package/comonad-5.0.2.

Krasner, G. E., Pope, S. T., et al. (1988). A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49.

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.

Meyerovich, L. A., Guha, A., et al. (2009). Flapjax: a programming language for Ajax applications. In *ACM SIGPLAN Notices*, volume 44, pages 1–20. ACM.

Moggi, E. (1991). Notions of computation and monads. *Information and computation*, 93(1):55–92.

Moore, E. F. (1956). Gedanken - experiments on sequential machines. *Automata studies*, 34:129–153.

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.

Psaila, G. (2008). Virtual DOM: An efficient virtual memory representation for large xml documents. In *Database and Expert Systems Application, 2008. DEXA'08. 19th International Workshop on*, pages 233–237. IEEE.

PureScript (2013). PureScript programming language. http://www.purescript.org.

Russel, A. (2011). Web components and model driven views. *fronteers.nl*.

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

Wadler, P. (1995). Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer.

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