To be Decided?

Bla bla bla...

By

EDDY BERTOLUZZO

Faculty of Electrical Engineering, Mathematics and Computer Science
DELFT UNIVERSITY OF TECHNOLOGY

A dissertation submitted to Delft University of Technology in accordance with the requirements of the MASTER OF SCIENCE degree in the Faculty of Electrical Engineering, Mathematics and Computer Science.

OCTOBER 2016

Abstract

In the last few years the *** of Reactive Programming has gained much traction in the developer's community, especially with the general trend for programming languages to embrace functional programming and the shifting of today's applications towards an asynchronous approach to modeling data. The goal of this work is to shed some light to the sea of more or less informed opinions regarding what the meaning of Reactive Programming is thorough the use of Mathematics...

Dedication and acknowledgements

Here goes the dedication.

Table of Contents

CHAPTER 1

INTRODUCTION

The cold winds are rising in the North... Brace yourselves, winter is coming.

— George R.R. Martin, *A Game of Thrones*

With the evolution of technologies brought in by the new millennium and the exponential growth of Internet-based services targeting millions of users all over the world, the Software Engineers' community has been continuously tested by an ever growing number of challenges related to management of increasingly large amounts of user data[2].

This phenomena is commonly referred to as Big Data. A very popular 2001 research report[3] by analyst Doug Laney, proposes a definition of big data based on its three defining characteristics:

- *Volume*: the quantity of data applications have to deal with, ranging from small - e.g. locally maintained Databases - to large - e.g. distributed File Systems replicated among data centers.
- *Variety*: the type and structure of data, ranging from classic SQL-structured data sets to more diversified and unstructured ones such as text, images, audio and video.
- *Velocity*: the speed at which data is generated, establishing the difference between pull-based systems, where data is synchronously pulled by the consumer, and push-based

systems, more suited for handling real-time data by asynchronously pushing it to its clients.

Each of these traits directly influences the way programming languages, APIs and databases are designed today. The increasing volume calls for a declarative approach to data handling as opposed to an imperative one, resulting in the developer's focus shifting from how to compute something to what it is to be computed in the first place[1]. The diversification of data, on the other hand, is the main drive for the research and development of noSQL approaches to data storage. Lastly, the increase in velocity fuels the need for event-driven, push-based models of computation that can better manage the high throughput of incoming data[4].

In this context, the concept of *reactive programming* has gained much traction in the developer's community as a paradigm well-suited for the development of asynchronous event-driven applications[?]. Unfortunately, reactive programming has been at the center of much discussion, if not confusion, with regards to its definition, properties and principles that identify it[?].

The goal of our work is to shed light on this much discussed topic by providing a formalization of the reactive programming paradigm. We are going to do so by means of a mathematical approach in the derivation of the reactive types. We will then continue with the development of an API which builds upon the previously derived theoretical foundations and discuss how this relates to the already existing, commercial reactive libraries.

1.1 Contributions

To the best of our knowledge, we are not aware of any previous work which analyses reactive programming from a mathematical and theoretical perspective..

Many attempts have been conducted in order to formalize this concept, some more business-driven than others, such as the Reactive Manifesto, Reactive Streams. None of these precisely defines reactive programming, but in most cases result in a nice document to be presented to a company's management for adoption - lots of buzzwords, not much substance.

Write contributions

1.2 Overview

Chapter 2 starts with ... bla bla bla

Write overview

1.3 Notation & Conventions

In the exposition of our work we will use Haskell as the reference programming language.. because it's close to mathematical notation..

Write notation and conventions

CHAPTER 2

INTO THE RABBIT HOLE

"It was much pleasanter at home," thought poor Alice, "when one wasn't always growing larger and smaller, and being ordered about by mice and rabbits. I almost wish I hadn't gone down the rabbit-hole – and yet – and yet – ..."

— Lewis Carol,
Alice in Wonderland

The use of the term reactive programming in scientific literature is dated back to the mid-sixties[?]. A relevant and insightful definition was given by G. Berry in 1991[?] as he describes reactive programs in relation to their counterparts, interactive programs:

> "*Interactive programs* interact at their own speed with users or with other programs; from a user point of view, a time-sharing system is interactive. *Reactive programs* also maintain a continuous interaction with their environment, but at a speed which is determined by the environment, not by the program itself."

Interactive programs concretize the idea of a pull-based model of computation, where the program - the consumer in this case - has control over the speed at which data will be requested and handled. A perfect example of an interactive program is a control-flow structure such as a for-loop

iterating over a collection of data: the program is in control of the speed at which data is retrieved from the containing collection and will request the next element only after it is done handling the current one.

Reactive programs, on the contrary, embody the idea of a push-based - or event-driven - model of computation, where the speed at which the program interacts with the environment is determined by the environment rather than the program. In other words, it is now the producer of the data - i.e. the environment - who determines the speed at which events will occur whilst the program's role reduces to that of a silent observer that will react upon receiving events. Standard example of such systems are GUI applications dealing with various events originating from user input - e.g. mouse clicks, keyboard button presses - and programs dealing with stock market, social media or any other kind of asynchronous updates.

The following write up is meant as a summary of what my thesis will expand upon. When thinking about effects and how they combine with different types of computations, we can come up with the following diagram that nicely summarizes them.

	One	**Many**
Sync	f :: () -> a	Iterable a
Async	callback :: a -> ()	???

Find duality in the first column to justify the use in the second? Does what I wrote up there work?

When our goal is to retrieve a single value of type `a` synchronously, we will make use of a function whose return type is `a` itself; one the function is called, we will block until a value is computed and returned. In an asynchronous setting, on the other hand, our function will return a `Future a`, i.e. a computation that at a certain point in the future will result in a value of type `a`.

When we are dealing with multiple values, we will typically return a collection which will contain our results. An `Iterable a` is an abstraction over collections which generalizes the concept of a data structure containing multiple values which are synchronously retrievable.

The purpose of this thesis is to explore the design space left empty in the above diagram and derive a solution to the problem of dealing with multiple asynchronous values in our programs.

2.1 Iterables

We begin by introducing the concept of `Iterable`, made famous by the Gang of Four pattern of the same name and widely adopted by most recent programming languages.

An Iterable is a programming structure which enables the user to traverse a collection of data, abstracting over the underlying implementation.

The interface and semantics of Iterables are fixed across programming languages.

Add C# version

```haskell
class Iterable a where
    getIterator :: () -> IO (Iterator a)

-- Java-like Iterator
class Iterator a where
    next    :: () -> IO a
    hasNext :: () -> Bool

-- C#-like Iterator
class Iterator a where
    current  :: () -> a
    moveNext :: () -> IO Bool
```

Note that different programming languages might expose slightly different interfaces for the Iterator construct. These differences affect the way the API is used by programmes, but do not affect the essence of the Iterator pattern and therefore are not relevant. For this reason and without loss of generality, we will make use of the Java-like definition of Iterator in the reminder of the discussion.

We will now make use of a few mathematical concepts in order to explore the design space around collections of asynchronous values: duality, continuations and CPS, products and coproducts and lastly currying and uncurrying. See Appendix A for a quick introduction to these concepts.

Let's start by stripping down the Iterable interface of all the unnecessary API/imperative design features so that we can derive it's essence and easily reason about it's properties.

The first step is simplifying the Iterator interface: when we take a close look, we can easily see that an Iterator is nothing but a function which returns either a value - when hasNext returns true and next is called - or nothing - when hasNext returns false. Additionally, and Iterator can also throw an exception in the case that next is called when the underlying collection has been exhausted, something that was not explicit in the C# definition.

Merging these consideration with the notion of coproducts and the maybe types, we obtain the

following definition of Iterable.

```
class Iterable a where
    getIterator :: () -> IO (Iterator a)
class Iterator a where
    moveNext :: () -> IO (Either Error (Maybe a))
```

Note how the moveNext function still produces an `IO` computation.

Let's now go one step further and simplify our definition even more, with the purpose of understanding the real essence of an Iterable. In order to achieve this, let's forget for a moment about errors and no values returned from an iterator, i.e. let's ignore operational concerns such as exceptions and termination.

```
type Iterable a = () -> IO (() -> IO a)
type Iterator a = () -> IO a
```

An Iterable is simply a function which, when invoked, returns an Iterator and an Iterator is itself a function which returns a computation that will eventually result in a value of type a. In some way, we can think of an Iterator as a getter function, and of an Iterable as a getter of a getter.

2.2 Into the rabbit hole: Deriving the Observable

As we noticed before, the way we deal with a single value in an synchronous environment is dual to the asynchronous method, flipping the arrows brings us from a function resulting in a value of type `a` to one accepting an `a`.

It seems reasonable now to apply the same concept of duality to the Iterable in order to find a possible solution to our original problem, dealing with a collection of asynchronous values.

```
type Iterable a = () -> IO (() -> IO a)
type Iterator a = () -> IO a

type Observable a = (a -> IO ()) -> IO ()
type Observer a = a -> IO ()
```

It is straightforward to see how duality plays out in the derivation of the new types.

This dualisation leaves us with an interface which is interesting under many point of views. First of all, to link back to the observation that an Iterable is a getter of a getter, we can observe that the Observable plays exactly the opposite role, that is, a setter of a setter, the Observer.

This brings us to another interesting observation, the Iterable embodies the idea of pull collections, the Iterable will give us a new Iterator whenever we ask for it and the Iterator, in turn, will provide us with the next value whenever we decide to pull one from it. Dually, the Observable interface embodies the idea of push collections: we push a callback, the Observer, inside the Observable, which, in turn, will push a value into the Observer whenever one becomes available.

From here we can take two roads, one is to analyze further the type that we obtained and see what it's properties are and what we can understand from it; the second road is to un-simplify its definition and augment it so as to arrive to a usable API for handling asynchronous collections. We will start with the first.

For those who are familiar with functional programming it will not be hard to see how the Observable type resembles a function written in continuation passing style.

```
1 cont       :: (a -> r) -> r
2 observable :: (a -> IO ()) -> IO ()
```

We can easily observe how an Observable is nothing more than a CPS function where the result type `r` is instantiated to `IO ()`.

This observation opens up a great deal of mathematics properties and laws that we can prove about observables.

First of all, let's convince ourselves that a CPS function, and therefore an Observable, truly embodies a push based model of computation. The very definition of continuation tells us that a continuation represents the "rest of the computation"; when looking at it from an observable prospective, the continuation, i.e. the observer, is the function that specifies what needs to happen to a value, whenever this becomes available, that is, whenever the observable pushes it to the observer. Since the continuation can be called multiple times, it is easy to see how this structure lets us deal with multiple values that are might come in at different times in the future.

Note how an observable is "paused" until it receives an observer to which it can push the values it produces. This comes precisely from the definition of CPS function, i.e. a suspended computation which, given another function as argument, the continuation, will produce the final result.

Therefore an observable is a suspended computation that will start producing values once we pass

in an observer, the continuation that will specify what happens to a value once it is computed. The observable will call the observer every time it produces a value.

2.3 Observables are Continuations

Talk about the interface that we want to get, even a minimal one now and explain how this relates to the cont monad, e.g. subscribe = ContT. Introduce the ContT monad and show the minimal reference implementation.

Our goal is now to start from this theoretical explanation of a push based collection and build a usable API. In the previous section we observed how and Observable is a particular case of the continuation monad where the result type is a computation which has unit as a result type. The Haskell language provides a monad construct for expressing continuations as well as a monad transformer in order to stack continuations on top of other monads. A monad transformer is exactly what we need in order to express our continuations resulting in IO.

```
type Observable a = ContT () IO a
type Observer a = a -> IO ()
```

Our API will also need to provide functions to create as well as start the Observable stream.

```
-- Simply wraps the function :: (a -> IO ()) -> IO ()
-- inside the Observable datatype
observable :: (Observer a -> IO ()) -> Observable a
observable = ContT

-- Runs the Observable by providing the continuation - i.e. the Observer -
-- that will handle the asynchronous data.
subscribe :: Observable a -> Observer a -> IO ()
subscribe = runContT
```

We can already notice how This should come as no surprise after our discussion regarding the connection between Observables and the Continuation monad. Conversely, these equivalences should justify even more the use of this model of computation for reactive programming.

At this point we have all the necessary tools to create and run an Observable

```
let obs = observable $ \obr ->
    do obr 1
       obr 2
       obr 3

subscribe obs print

output>
1
2
3
```

The example above is a toy example, let's try with a more realistic one such that we can show that our basic implementation of rx based on continuations works just as well as a full blown one in terms of handling asynchronous data.

Put the actual demo that listens to keyboard presses. In the example, press 3 times and see that all the events are handled and none are lost.

```
obs = observable $ \obr -> do
    passiveMotionCallback $= Just (\p -> obr p)

main :: IO ()
main = do
  (_progName, _args) <- getArgsAndInitialize
  _window <- createWindow "Hello World"
  subscribe obs print
  mainLoop
```

Elaborate better this part. Ask Erik for input.

It is worth noting that rx in itself is not at all async in handling data unless we use schedulers, although it doea handle asynch data. This is a common misconception, even if you have a single thread that doesn't mean you cannot handle async data, actually you are async because the control flow isn't linear. Naturally, the thing is that certain queries (the ones that don't use schedulers) will simply block your single thread and prevent other things from happening.

This demo shows exactly this, even though the processing of the data is synchronous, the data itself, being mouse movements, is inherently async.

At this point in the discussion we have a working implementation of a push based collection purely derived from the underlying theory of duality and continuations. The next step in augmenting out library is to note that continuations are monads and, being the Observable an instance of the continuation monad, it is itself a monad. This observation comes with great benefits, we can get mathematical laws - the monad laws - proven for free for our structure and, from a more practical point of view, we get functions defined for free for the Observable: fmap (from Functor), flatmap and return.

Say more on the monad laws, even if they are trivially proved.

We will soon see how the implementation of these functions will change and move from the standard one the moment we start moving towards a more operational implementation for real world use.

2.4 Out of the rabbit hole: Towards a usable API

Start from the minimal reference implementation, add exceptions and termination, explain how flatmap changes and lastly add subscription and talk about cancellation and the conctract.

Up until now we have analyzed the essence of the Observable, leaving out the operational concerns that would come up when our goal is to design a usable API. Now we will slowly and step by step re-introduce these concerns in order to go from a theoretical definition of Observables to a more operational and therefore usable one.

Remember how, at the beginning of our discussion, we greatly simplified the Iterable interface in order to derive a type that represents the essence of an Iterable. We later applied the duality principle from category theory in order to derive the Observable. Now, we want to walk the semplification path backwards and, step by step, re-introduce all that we simplified before in order to arrive to a usable API.

2.4.1 Adding Termination and Error handling

The first thing we want to re-introduce is handling exceptions and termination of a stream. Where an Iterable can return a value, terminate or throw an exception when we ask for a value, and Observable, being it's dual, can produce one or more values, terminate or throw an exception when it is subscribed to.

A more appropriate type for our interface is then the following.

```
4 type Observable a = ContT () IO (Either Error (Maybe a))
5 type Observer a = Either Error (Maybe a) -> IO ()
```

Remove this Event shit and move to the custom type directly? Not sure, I like the discussion on bind.

Now, this code is not exactly the definition of readable; let's apply some good design skills to make it more pleasant to the eye without changing it's meaning.

```haskell
-- datatype representing Either Error (Maybe a)
data Event a =
      OnNext a
    | OnError SomeException
    | OnCompleted

type Observable a = ContT () IO (Event a)
type Observer a = Event a -> IO ()
```

Although this might not look like a big change, it greatly influences the design of our API. We are, in fact, changing our instantiation of the continuation monad to an input type that is not `a` anymore, but `Event a`. On the other hand, out type variable for Observable is still `a`. This is not an issue per se, but it has one big consequence: the flatmap function that we inherit from the continuation monad is not the one that we want to expose from out API anymore. The types differ like so.

```haskell
-- bind inherited from the Continuation monad
(>>=) :: Observable a -> (Event a -> Observable b) -> Observable b

-- bind that we would like to expose from our API
flatmap :: Observable a -> (a -> Observable b) -> Observable b
```

This has many implications, first of all, we are gonna need to implement flatmap by ourselves.. see todo below...

It has now come the time to move away from an implementation of Observable as a type synonym. We have already seen how the current implementation using `Event a` does not allow for a correspondence between `>>=` operations; this will only create confusion in the future. The next step is then to define our own observable tybe, which will clearly be really similar to the Continuation monad and subsequently prove that it is itself a monad.

```haskell
newtype Observable a = Observable { subscribe :: Observer a -> IO () }
data Observer a = Observer
    { onNext       :: a -> IO ()
    , onError      :: SomeException -> IO ()
    , onCompleted  :: IO ()
    }
```

With this implementation we have eliminated the materialisation of the event types. The Observer is now not a single function from `Event a -> IO ()` but a collection of 3 continuations that will be used inside the observable depending on the type of the event. It is clear that this implementation of Observable has not changed in functionality from the previous one using the Continuation Monad, it has just dematerialized the 3 types of events in 3 functions which handle them.

The next step is to make Observable a monad

```haskell
instance Monad Observable where
        return a = observable (\obr -> onNext obr a)
        o >>= f = ...
```

The return function is the exact same as in the continuation monad, with the only difference that we have now 3 continuations to chose from instead of a single one.

Bind, on the other hand, is completely different from the Cont monad implementation; in this case ...

finish the discussion

The only thing left to do now is to prove the monad laws to show that Observable really is a monad.

```haskell
-- Monad Laws
-- return a >>= k  =  k a
-- m >>= return  =  m
-- m >>= (x -> k x >>= h)  =  (m >>= k) >>= h
```

We mentioned before how the bind from Cont differs from our in the Observable. Below I will show that in this implementation it corresponds to a function lift that ...

Talk about lift = »= in Cont.

By using lift we can transform streams and implement operators...

Modify keyboard press example from before to handle errors and termination. Point to later discussion regarding the rx contract, since now we can detect termination and errors but there is no guarantee that nothing will come after we receive them, i.e. that we abide the contract.

PUT THE CONTRACT HERE????

2.4.2 Adding Schedulers

Up until now our discussion on push based collections has not mentioned time. This might seem strange, especially when coming from and FRP background, where continuous time and functions are at the foundations of the theory.

- Elaborate on orthogonality of time. - Discuss how rx is synch by itself and how we need to add concurrency in order not to have it blocking. - Discuss the different possible levels of concurrency - Discuss how this affects the implementation of operators: levels of safety - Discuss how threadpool scheduler breaks monad laws.

Ask Erik about this.

2.4.3 Adding Subscriptions

With error handling and termination, our implemnentation starts getting more and more usable. We now want to add a mechanism that will allow us to stop an observable stream from the outside (as opposed to waiting for an onCompleted) whenever we don't require it's data anymore.

In order to achieve this we will use a new datatype, Subscription. The idea is that the subscribe function will return a Subscription to the user who will later be able to call unsubscribe on it and stop the stream associated to it from producing any more values.

Talk about the best effort in canceling work and eventual consistency with the contract.

A Subscription in itself is nothing more than an IO action to perform once we stop an observable stream. Naturally, to make it usable in a real setting, we are gonna need to augment it more information: first, we will need a way to test whether the subscription is unsubscribed or not. The easiest way to do this is to use mutable state and store a boolean value with every Subscription. Moreover, some operators

discuss childern subscriptions

Note that from now on, in order to allow our implementation to be usable, we will make use not only of functional language features, but of imperative ones as well. This will be done in situations in which it makes sense from an understandability point of view. There's no shame in using all our tools and being a purist is not always the best way.

Probably better to put what follows somewhere else, right after we leave the theory for example.

Note that, even though the Observable's theoretical foundation is strictly functional, the road to make it usable is full of obstacles that are better tackled using imperative features, i.e. state. As much as I personally prefer a functional approach to programming, I will favor the solution that most clearly and easily solves the problem, be that functional of imperative.

```
1 type Subscription = IO ()
```

Our goal now is to implement a cancellation mechanism that would stop the Observable. This will be achieved by calling a function unsubscribe, which will prevent, from that moment on, any events to be signaled to any subscribed observer.

This is achieved by wrapping the user supplied observer to the subscribe function with an internal one which adds this functionality and forwards all accepted events to the supplied one.

the code for safe observer.

```
25      OnNext a
26    | OnError SomeException
27    | OnCompleted
28
29  --we can remove returning the subscription to the ouside
30  --since the behaviour is reproducible with operators
31  --e.g. subject+takeUntil
32  subscribe :: Observable a -> Observer a -> IO Subscription
33  subscribe obs obr = do
34      s <- createSubscription (print "unsubscribed")
35      let
36          safeObr = observer safeOnNext safeOnError safeOnCompleted
37          safeOnNext a = do
38              b <- isUnsubscribed s
39              when (not b) $ obr (OnNext a)
40          safeOnError e = do
```

As a design decision, when unsubscribe is called on an observable subscription, the observable sequence will make a best effort attempt to stop all outstanding work. This means that any queued work that has not been started will not start. Any work that is already in progress might still complete as it is not always safe to abort work that is in progress. Results from this work will not be signaled to any previously subscribed observer instances.

Motivation for children subscriptions: some operators will create inner observables and therefore will need to unsubscribe from them when the outer observable is unsubscribed from.

On why using Cont () (StateT Subscription IO) (Event a) wouldn't work: it would be perfect in order to thread Subscriptions throughout execution making it usable inside operators, since the call to the continuation would return a state which would then be sequenced by »=. This method fails with schedulers, in particular newThread since the state of the action executed on the new thread would be disconnected from the threading mentioned above. The other thread would get a state but it would not know what to do with it and would not have any ways to connect it to the original one passed by the subscribe. Another way is to use mapStateT to map the IO action that will result from the state to an action on the other thread. Again, this method won't work,

---

On the motivation for a Subscription: it is needed so the user can cancel work at any time. This implies that a scheduler is used. If this is not the case the subscription will be returned

synchronously after the execution of the whole stream. It will therefore be already unsubscribed and calling unsubscribe will be a NoOp. In the case that we actually use schedulers then we can unsubscribe from anywhere in our program.

Now the question is the following: can we reproduce the behaviour of unsubscribe with operators so that we can eliminate subscriptions altogether? That is, we can hide them to the outside and use them only inside the stream, we still need them but we don't necessarily need to return them when we subscribe. The behavious can be easily replaced by the use of takeUntil(Observable a) where we pass in a subject that will be signaled when we want to stop the stream. The takeUntil operator fires events from the upstream up until the point in which we signal the subject, then stops the stream and unsubscribes.

With this approach we seemingly lose one thing, i.e. the ability to specify what happens at unsubscription time; this functionality can simply be regained by a subscribe function that takes the unsubscribe action as a parameter and runs it when the stream is unsubscribed.

2.5 The Reactive Contract

Begins a section.

2.5.1 Subsection

Begins a subsection.

TODO LIST

- Modify keyboard press example from before to handle errors and termination. Point to later discussion regarding the rx contract, since now we can detect termination and errors but there is no guarantee that nothing will come after we receive them, i.e. that we abide the contract. 15
- Ask Erik about this. 15
- Talk about the best effort in canceling work and eventual consistency with the contract. 15
- discuss childern subscriptions 15
- Probably better to put what follows somewhere else, right after we leave the theory for example. 16
- the code for safe observer. 16
- Motivation for children subscriptions: some operators will create inner observables and therefore will need to unsubscribe from them when the outer observable is unsubscribed from. 17

APPENDIX A

Begins an appendix

A.1 Duality

A.2 Continuation Passing Style

A.3 Product and Sum Types

A.4 Currying and Uncurrying

BIBLIOGRAPHY

[1] D. FAHLAND, D. LÜBKE, J. MENDLING, H. REIJERS, B. WEBER, M. WEIDLICH, AND S. ZUGAL, *Declarative versus imperative process modeling languages: The issue of understandability*, in Enterprise, Business-Process and Information Systems Modeling, Springer, 2009, pp. 353–366.

[2] B. FURHT AND A. ESCALANTE, *Handbook of cloud computing*, vol. 3, Springer, 2010.

[3] D. LANEY, *3d data management: Controlling data volume, velocity and variety*, META Group Research Note, 6 (2001), p. 70.

[4] E. MEIJER, *Your mouse is a database*, Queue, 10 (2012), p. 20.