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

"Well, here at last, dear friends, on the shores of the Sea comes the end of our fellowship in Middle-earth. Go in peace! I will not say: do not weep, for not all tears are an evil."

— J.R.R Tolkien,
The Return of the King

Here goes the dedication.

Table of Contents

Page

Introduction

"Lasciate ogne speranza, voi ch'intrate."

— Dante Alighieri, *Divina Commedia*

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[6].

This phenomena is commonly referred to as Big Data. A very popular 2001 research report[10] 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[5]. 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[11].

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[2]. 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[12].

The goal of our work is to use mathematics as a tool to analyze today's commercial reactive libraries and understand how they relate to the theoretical concept of reactive programming. We are going to do so by utilizing concepts and ideas from functional programming and category theory with the purpose of deriving the reactive types, and will subsequently continue with the development of a reference reactive library which builds upon the previously derived theoretical foundations.

Contributions

To the best of our knowledge, we are not aware of any previous work which analyses reactive programming from a theoretical point of view or derives its types and interfaces though the use of mathematics.

The most known attempt at defining reactive programming is the Reactive Manifesto[**?**], a document that tries to describe reactive systems in terms of design principles and conceptual architecture. Although certainly insightful, we feel the document targets a more managing-focused audience as opposed to software developers and engineers.

Our work, on the other hand, aims at providing types and interfaces that describe the real essence of the reactive paradigm, aiding engineers that wish to use or develop reactive libraries in understanding and taking more informed decisions in the matter.

Overview

Chapter 2 starts with ... bla bla bla

Write overview

Notation & Conventions

In the exposition of our work we will make use of Haskell as the reference language. This decision is motivated by the language's strong connection with mathematics as well as it's clean syntax.

All the code presented in this report, a minimal complete theoretical implementation and a full blown library implementation of a reactive library can be found at the associated code repository on Github ().

Github link

CHAPTER 1

REACTIVE PROGRAMMING

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

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

In this chapter we are going to introduce the concept of reactive programming and motivate it's importance and relevance with regards to modern applications and the type of problems developers have to face nowadays. We are then going to introduce the most popular commercial libraries that claim to solve the reactive problem, with the purpose of giving the reader some context for our discussion and motivating the need for a mathematical formalization that abstracts over the class of problems these implementations set out to address.

1.1 The Essence of Reactive Programs

The use of the term reactive program in scientific literature is dated back to the mid-sixties[1]. A relevant and insightful definition was given by G. Berry in 1991[3] as he describes reactive programs in relation to their dual 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 itself. 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.

1.2 Why Reactive Programming Matters

Considering the definition and examples of reactive programs we analyzed in the previous section, let's now try to formalize the class of problems the reactive programming paradigm is specifically well-suited for.

The table below provides a collection of types offered by common programming languages for handling data, parameterized over two variables: the size of the data, either one or multiple values, and the way data is handled, either by synchronous or asynchronous computations[13].

	One	**Many**
Sync	`a`	`Iterable a`
Async	`Future a`	*Reactive Programming*

The first row shows that synchronous functions come in two flavors: classic functions that return a single value of type `a` and functions that produce a collection of results of type `a`, abstracted through the `Iterable a` interface (See section 2.1). These types of functions embody the standard imperative, pull-based approach to programming, where a call to a function/method synchronously blocks until a result is produced.

Moving on to the second row, we encounter `Future a`, an interface representing an asynchronous computation that, at a certain point in the future, will result in a value of type `a`. Futures are generally created by supplying two callbacks together with the asynchronous computation, one to be executed in case of success and the other one in case of error.

Programming languages, however, are not as well equipped when it comes to handling asynchronous computations resulting in multiple values - i.e. push-based collections. The issue lies in the fact that the program's control flow is dictated by the environment rather than the program itself - i.e. inversion of control -, making it very hard to model such problems with commonly known control structures, which are optimized for sequential models of computation. Traditional solutions typically involve developers manually trying to compose callbacks by explicitly writing CPS (continuation passing style) code[13], resulting in what it's commonly referred to as *Callback Hell*[4].

The aforementioned class of problems reflects the definition of reactive programs we analyzed in the previous section, where the environment asynchronously - i.e. at its own speed - pushes multiple events to the program. The reactive programming paradigm sets out to provide interfaces and abstractions to facilitate the modeling of such problems as push-based collections.

However, interfaces are only as good as the implementations that back them up. In the next section we are going to discuss and analyze various APIs and libraries that claim to embody the reactive paradigm, motivating our need for a mathematical formalization to aid in unifying these different approaches under a single set of interfaces, eliminating doubts and confusion when it comes to the essence of reactive programming.

1.3 Reactive Programming in the Real World

Talk about FRP, Reactive Streams, Rx and other stuff. CIte survey paper Claim that Observable is dual of Iterator in order to justify the why.

1.3.1 Reactive Extensions

1.3.2 Reactive Streams

Try to define itself as reactive programming, mathematics prove the claim wrong, they implement asynchronous iterables. This does not make them useless, yet they are not a good solution for the problem we are trying to solve.

Probably put the derivation of Reactive Streams in the appendix, it's not really the focus, I guess.

1.3.3 Functional Reactive Programming

Much researched topic, beautiful in theory, yet based on continuous time, un-achievable with computers where time is inherently discrete. Connel Elliot, the inventor of FRP, says that commercial implementations are far from the original theory and do not really represent the essence of FRP

Chapter 2

Into the Rabbit Hole: *Deriving the Observable*

> *"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*

In this chapter we are going to derive the `Observable` interface starting from its dual counterpart, the `Iterable`, which, as we have seen in Chapter 1, embodies the idea of a pull-based model of computation and is the commonly adopted solution to dealing with synchronous computations resulting in multiple values.

2.1 Iterables

An `Iterable` is a programming construct which enables the user to traverse a collection of data, abstracting over the underlying implementation[7].

The interface and semantics of `Iterable`s were first introduced by the Gang of Four though their Iterator pattern[7]; today's most used programming languages expose the `Iterable` as the root interface in standard Collections APIs.

The `Iterable` interface is generally fixed across programming languages, with the exception of naming conventions - C# and related languages call it IEnumerable - and slight differences in the types, as we can see from the following definitions.

Should I use a datatype instead of class?

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

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

---------------------------------------------

-- C# IEnumerable
class IEnumerable a where
    getEnumerator :: () -> IO (IEnumerator a)

class IEnumerator a where
    moveNext :: () -> IO Bool
    current  :: () -> a
```

Although the essence of the pattern is preserved in both definitions, we claim that the C# version more clearly and accurately reflects the way side effects play a role in the usage of the interface: `moveNext` contains all the side effects of walking down the underlying collection and retrieving the next value while `current` can inspect the retrieved value multiple times in a pure way. The Java version, on the other hand, embeds the side effect in the `next` function, making it impossible to inspect the current value multiple times. For this reason and without loss of generality, we will make use of the C# definition - modulo naming conventions - in the reminder of the discussion.

Mention Subscription here?

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

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

The derivation that follows will require the use of a number mathematical concepts such as categorical duality, continuations, (co)products, (un)currying, covariance, contravariance and functors. We suggest the reader to get familiar with these topics before diving into the derivation. An accessible introduction to each can be found in Appendix A.

2.2 The Essence of Iterables

The first step in deriving the `Observable` is to simplify our `Iterable` definition to a type that reflects its very essence; we are gonna do this by stripping the interface presented in the previous section of all the unnecessary operational features that only clutter our definition.

Let's start by taking a closer look at the `Iterator` interface; we can observe that the definition of the functions `moveNext` and `current` is equivalent to a single function which returns either a value - analogous to a `moveNext` call returning true and a subsequent invocation to `current` - or nothing - analogous to a call to `moveNext` returning false.

Before we formalize this observation with a proper type, let us notice another effect that is hidden in the current definition of `moveNext`, but not made explicit by the its type: the possibility for an exception to be thrown by the function's body.

By merging these considerations with the notion of coproducts and Haskell's `Either` and `Maybe` type, we obtain the following definition.

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

Note how, theoretically, `getIterator` could also throw an exception. We assume here, without loss of generality, that the function will never throw and will always be able to return an `Iterator` instance.

The next step is to forget about class instances and express our interfaces as simple types.

```
type Iterable a = () -> IO (Iterator a)
type Iterator a = () -> IO (Either Exception (Maybe a))
```

At this point, we want to put aside the operational concerns regarding exceptions and termination and assume the `Iterator` function will always return a value of type `a`.

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

We have now reached a point where no simplification is possible anymore. The obtained types reflect the essence of the Iterator patter: an `Iterable` is simply a function which, when invoked, produces an `Iterator` and an `Iterator` is itself a function producing a value of type `a` as a side effect.

Reformulate this part better, ask Erik for advice.

These types present some interesting properties; let's start by analyzing the `Iterator`, a function which, given nothing, will produce a value of type `a`. This type should sound familiar to the reader acquainted with object oriented programming as it precisely describes the notion of a getter function, i.e. a lazy producer of values. When looking at the relation between the `Iterator` type and its base component, `a`, we can observe how they are bound by a covariance relation:

vending machine example?

$$\frac{A <: B}{() \to A <: () \to B}$$

The `Iterable`, on the other hand, is nothing more than a getter of another getter, the `Iterator` and therefore abides by the covariance relation as well.

To formally prove this intuition of a covariant relation, we instantiate the `Iterable` / `Iterator` types to a covariant `Functor`.

```haskell
{-
    Using a simplified syntax, hiding IO and Haskell's newtypes,
    in order to clearly show the idea:

    iterator :: () -> a
    iterable :: () -> () -> a

    fmapi ::  (a -> b) -> (() -> a)       -> () -> b
    fmapi         f            ia          = () -> f (ia ())

    fmapii :: (a -> b) -> (() -> () -> a) -> () -> () -> b
    fmapii        f            iia         = () -> fmapi f (iia ())
-}

newtype Iterator a = Iterator { runIterator :: () -> IO a }
newtype Iterable a = Iterable { getIterator :: () -> IO (Iterator a) }

instance Functor Iterator where
    fmap f ia  = Iterator $ \_ -> liftM f (runIterator ia ())

instance Functor Iterable where
    fmap f iia = Iterable $ \_ -> liftM (fmap f) (getIterator iia ())
```

We will see in the next section how these concepts are relevant in expressing and motivating the relevance of the duality between Iterables and Observables.

2.3 Applying Duality

By now, the reader should be somehow familiar with the concept of duality, as it has appeared many times throughout our discussion, in concepts such as pull and push models of computation or interactive and reactive programs. Duality is, in fact, a very important general theme that has manifestations in almost every area of mathematics[8] (See Appendice A for an introductory discussion on the topic).

Starting from the fact that the `Iterable` interface embodies the idea of interactive programming, let's use the principle of duality to derive the `Observable` interface and see how it relates to the concept of reactive programming. In practice, this translates to the simple task of flipping the function arrows in the `Iterable` interface, taking us from a function resulting in a value of type `a` to one accepting an `a`.

```
{-
    () -> (() -> a) -- iterable
    () <- (() <- a) -- apply duality
    (a -> ()) -> () -- observable
-}

type Iterator a = () -> IO a
             -- = () IO <- a
type Observer a = a  -> IO ()

type Iterable   a = () -> IO (() -> IO a)
               -- = () IO <- (() IO <- a)
type Observable a = (a -> IO ()) -> IO ()
```

Not how the side effects are bound to function application rather than values, hence their flipped position in the `Observable` type.

The reader acquainted with functional programming will easily see the resemblance between the `Observable` type and a CPS function (See Appendice A).

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

It is clear how an `Observable` is nothing more than a CPS function where the result type `r` is instantiated to `IO ()`. To convince ourselves of this equivalence, let's think about the definition of a CPS function, i.e. a suspended computation which, given another function - the continuation - as argument, will produce its final result. This definition suits perfectly the idea behind `Observable` discussed in Section 1.3.1: a function which will do nothing - i.e. is suspended - until it is subscribed to by an `Observer`.

A continuation, on the other hand, represents the future of the computation, a function from an intermediate result to the final result[14]; in the context of `Observable`s, the continuation represents the `Observer`, a function specifying what will happen to a value produced by the `Observable`, whenever it will become available, that is, whenever it will be pushed into the `Observer`. Since a continuation can be called multiple times within the surrounding CPS context, it is easy to see how this mathematical concept allows us to easily deal with multiple values produced at different times in the future.

In the previous section we have discussed many properties associated with `Iterable`s. Let's analyze now how these properties translate under dualisation and how they affect our new derived interface, the `Observable`.

First, moving from 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 type `Observer :: a -> IO ()` represents, in fact, the essence of a setter function, whereas the `Observable` consists in nothing more than the simple task of applying the observer function to itself, producing a setter of setters.

While the discussion about `Iterable`'s covariance was quite intuitive, things get a little bit more complicated when analyzing `Observable`s. Let's start by informally introducing the notion of positivity and negativity of types: we can interpret a function of type `f :: a -> b` as a way for us to produce a value of type `b`. In this context, `b` is considered to be positive with respect to the type `a -> b`. On the other hand, in order to apply the function, we are going to need a value of type `a`, which we will need to get from somewhere else; `a` is therefore considered to be negative w.r.t. the function type, as the function introduces a need for this value in order to produce a result. The point of this distinction is that positive type variables introduce a covariant relation between base and function type whereas negative type variables introduce a contravariant relation.

Probably will need to riformulate this.

Analyzing `Iterable` within this framework is easy, the `Iterator` function contains a single type parameter found in a positive position, therefore resulting in a covariant relation; being the `Iterable` the result of applying the `Iterator` function to itself, we again result in a covariant relation w.r.t. the type parameter `a`.

The `Observer` function, on the contrary, introduces a need for a value of type `a`, resulting in a contravariant relation w.r.t. `a`. Again, the `Observable` function is the result of applying `Observer` to itself; surprisingly, this results in `a` being in a positive position. The intuition is easily understood by thinking about the rules of arithmetic multiplication: `a` is in negative position w.r.t. the `Observer` function, whereas the `Observer` is in negative position w.r.t. the `Observable`. This leads to `a` being negated twice, ultimately resulting in a positive position within the `Observable` function. Before we formalize this intuition, let's convince ourselves that `Observable`s effectively produces a value of type `a` by looking at an example:

```haskell
{-

f   ::  a ->  b
     = -a -> +b

g   ::  ( a ->  b) ->  c
     = -(-a -> +b) -> +c
     =  (+a -> -b) -> +c

observer
    ::  a -> ()
     = -a -> ()

observable
    ::  ( a -> ()) -> ()
     = -(-a -> ()) -> ()
     =  (+a -> ()) -> ()

-}

randomValueObs :: Observable Int
randomValueObs = Observable $ \observer -> do
    int <- randomRIO (1, 10)
    observer int
```

It is clear from this implementation that `randomValueObs` indeed produces a value of type `Int`, whereas the `Observer` introduces a need for such value in order to be applied. For more details on the positivity and negativity of functions and type variables, see[15][9].

Just as we did with `Iterable` we can formally prove the covariant and contravariant relations by instantiating `Observable` to a `Functor` instance and `Observer` to a Cofunctor one (Haskell uses the name `Contravariant` instead).

```haskell
import Data.Functor.Contravariant

newtype Observer   a = Observer   { onNext :: a -> IO () }
newtype Observable a = Observable { subscribe :: Observer a -> IO () }

instance Contravariant Observer where
    contramap f ob = Observer $ onNext ob . f

instance Functor Observable where
    fmap f ooa = Observable $ subscribe ooa . contramap f

```

2.4 Observables are Continuations

In the last section we mentioned how the `Observable` interface is equivalent to a CPS function. We are now going to formally prove this claim by providing a basic implementation of `Observable` as a type alias of Haskell's continuation monad. The Haskell language provides a monad construct for expressing continuations, as well as a monad transformer which allows the user to stack the functionality of continuations on top of other monads. A monad transformer is exactly what we need in order to express our `Observable` function producing a result in the IO monad.

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

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

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

```haskell
obs = newObservable $ \observer ->
    do observer 1
       observer 2
       observer 3

main :: IO ()
main = subscribe obs print

{-
output>
    1
    2
    3
-}
```

Notice how an observable only pushes values once subscribed to and acts as a suspended computation otherwise.

The code abobe, being a toy example fails to show some fundamental properties associated with this new interface; in particular, it fails to show how `Observable`s can actually handle asynchronous sources of data. The following snippet of code contains a more realistic and meaningful example of an `Observable` handling keyboard presses, asynchronous events by nature. It is worth noticing how our basic implementation of `Observable` based on continuations works just as well as a full blown one in terms of its core capability 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
```

This example brings us to the following observation: `Observable`s are capable of handling asynchronous data sources, yet the means by which the data is handled are not asynchronous by default. This is a common misconception and source of much confusion among the community: an `Observable`, by default, synchronously handles it's incoming data, blocking the next incoming events whilst processing the current one. This behavior is not fixed though: as we will see in Chapter 3, it is possible to make use of `Scheduler`s in order to alter the control flow of the data processing and dispatch the work to other threads.

At this point in the discussion we have arrived to a working implementation of a push based collection purely derived from mathematical and categorical concepts such as duality and continuations. In the next Chapter we will take the necessary steps to transform this basic implementation into a full blown API that can be used in real world applications.

Chapter 3

Out of the rabbit hole: *Towards a usable API*

> *TBD*
>
> — Lewis Carol,
> *Alice in Wonderland*

So far we focused our analysis on the essence of the `Observable` interface, setting aside the many operational concerns that would come up when trying to implement these concepts into a usable, commercial API. In this chapter we are going to adopt a step-by step approach that will guide us from our theoretical definition of `Observable` to a concrete and usable library, to which we will refer to as Rx.

When discussing the `Iterable`, we progressively simplified the starting interface in order to derive a type which would represent its essence. We later applied the notion of duality from category theory in order to derive the `Observable` interface. We are now going to walk the simplification path backwards and re-introduce the operational components we set aside before.

The following helps clarifying the steps we will undertake in the remainder of our discussion: first, think back to Section 2.3, when we compared `Observable` to a setter of setters, the result of applying the `Observer` function to itself. Starting from this notion let's define a function `(!)`, representing an `Observer`:

```
!a :: a -> IO ()
```

When we apply the function to itself - i.e. substitute `a` for `!a` - we obtain the `Observable`, which, as we saw previously, presents the same signature and can be implemented by the continuation monad.

```
!!a :: (a -> IO ()) -> IO ()
```

Next, we are going to define two new functions, `(?)`, representing a continuation that can deal with termination and errors as inputs, and `(%)`, a modified version of `(!)` which hands over a `Subscription`, a datatype used, as we will discuss, to implement premature stream termination.

```
?a :: Either Error (Maybe a) -> IO () -- error handling
%a :: a -> IO Subscription            -- stream subscription
```

In this chapter we are conceptually going to augment our initial continuation-based definition of `Observable` by plugging in these two functions, obtaining the following:

```
%?a :: (Either SomeException (Maybe a) -> IO ()) -> IO Subscription
```

This will have many implications on our definition of `Observable`, as moving away from an implementation based on the continuation monad will mean giving up on free class instances for `Observable` such as `Functor, Applicative, Monad`. This will not be a problem as we will easily be able to provide such instances by our own means.

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.

3.1 Termination and Error Handling

Let's start reshaping our interfaces by introducing a way to handle exceptions and termination of a stream. Informally, an `Observable` stream does not only produce one or more values, but it can terminate at a certain point in time or it can throw an exception while processing values.

A more appropriate type is then the following:

```haskell
type Observer   a = Either SomeException (Maybe a) -> IO ()
type Observable a = ContT () IO (Observer a)
```

Sticking with our `ContT` transformer in the definition of `Observable` starts now to become hairy and less and less readable. From a functional point of view, what we really need is for our original CPS function to accept two additional continuations, one dealing with completion and one with exceptions. This is unfortunately not expressable using `ContT`, as our hands are tight to one single input parameter. For this reason, let's implement our own datatypes for `Observable` and `Observer`.

```haskell
newtype Observable a = Observable
    { subscribe :: Observer a -> IO ()
    }

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

We will use these types as the base for the rest of our discussion.

On a side note, it is worth mentioning how the above code is not the only way we could have implemented termination and exceptions; by *materializing* the three different events to the datatype `Event a`, we could have provided the following code that is still tight to the continuation monad transformer.

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

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

Although this version is insightful for theoretical discussions on the properties and relations of `Observable` and the continuation monad, its use in a full fledged API is impractical. See Appendice B for a full implementation of the Rx Library based on this idea, which is meant to highlight the strong connection between `Observable` and other already existing functional structures from which it composes.

3.2 Operators

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.

```haskell
type Observer   a = Either SomeException (Maybe a) -> IO ()
type Observable a = ContT () IO (Observer a)
```

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 SomeException (Maybe a)
data Event a =
      OnNext a
    | OnError SomeException
    | OnCompleted

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

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 ()
```

```haskell
    , 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.

3.3 Concurrency with Schedulers

At the end of Section 2.4 we discussed how `Observable`s, by default, handle data by means of a synchronous pipeline, blocking the processing of successive elements via the call stack. It is worth mentioning again how this synchronous processing does not affect the ability of `Observable`s to handle asynchronous data.

However, this synchronous behavior might not always be the best solution, especially in real world applications, where we might want to have a thread dedicated to listening to incoming events and one which processes them. Enter the `Scheduler` interface, an orthogonal[?] structure w.r.t. `Observable` which allows us to introduce concurrency into our reactive equation.

`Scheduler`s allow us to to alter the control flow of the data processing within an observable expression, introducing a way to dispatch the work of any number of operators to be executed within the specified context, e.g. a new thread.

The `Scheduler` interface looks like the following [1].

```haskell
data Scheduler = Scheduler
    { _schedule      :: IO () -> IO ()
    , _scheduleDelay :: TimeInterval -> IO () -> IO ()
    }
```

`Scheduler`s expose two functions which are essentially equal, modulo arbitrary delays in time. Both of these functions take an `IO` action as input and dispatch it to the appropriate execution context, producing a side effect.

To better understand `Scheduler`s, let us present the implementation of one of them, the newThread scheduler, which allows us to dispatch action to a new, dedicated thread.

[1] The interface presented in this section is the result of a simplification of the actual one, which involves `Subscription`s. We will discuss the impact of `Subscription`s on `Schedulers` in the next section; suffices to know that the version presented here has no negative effects w.r.t the generality of our discussion.

```haskell
newThread :: IO Scheduler
newThread = do
    ch  <- newTChanIO
    tid <- forkIO $ forever $ do
        join $ atomically $ readTChan ch
        yield
    return $ Scheduler (schedule_ ch) (scheduleDelay_ ch)
        where
            schedule_ ch io = atomically $ writeTChan ch io
            scheduleDelay_ ch delay io = do
                threadDelay $ toMicroSeconds delay
                atomically $ writeTChan ch io
```

The `newThread` function gives us a side effectful way of creating a `Scheduler` by generating a new execution context - i.e. a new thread - and setting up the necessary tools for safe communication with it. The `Scheduler` functions we are provided, on the other hand, simply write the input `IO` action to the channel and return, effectively dispatching the execution of those actions to the new thread.

Up to this point we haven't mentioned `Observable` s at all. This is the reason why we previously claimed that `Scheduler` and `Observable` are connected by an orthogonal relationship: the two interfaces are independent from one another, yet, when used together within an observable expression, they provide the user with greater expressive power w.r.t. concurrency.

The only thing missing now is a way for us to combine the functionality of these two interfaces: `observeOn` and `subscribeOn` are the operators that will aid us on this task. The former will allow us to dispatch any call to an observer continuation on to the specified execution context, whereas the latter will allow us to control the concurrency of the `Observable` subscribe function.

For the sake of completeness and understandability, the following snippet contains a simple implementation of the `observeOn` operator together with a sample usage.

```haskell
observeOn :: Observable a -> IO Scheduler -> Observable a
observeOn o sched = Observable $ \obr -> do
    s <- sched
    subscribe o (f s obr)
        where
            f s downstream = Observer
                {   onNext      = void . _schedule s . onNext downstream
                ,   onError     = void . _schedule s . onError downstream
                ,   onCompleted = void . _schedule s $ onCompleted downstream
                }
```

```
obs = Observable $ \obr ->
    do onNext obr 1
       onNext obr 2
       onNext obr 3
       onCompleted obr

obr :: Observer Int
obr = Observer on oe oc
    where
        on v = do
            tid <- myThreadId
            print (show tid ++ ": " ++ show v)
        oe = print . show
        oc = print "Completed"

main :: IO ()
main = do
    hSetBuffering stdout LineBuffering
    subscribe obs’ obr
    tid <- myThreadId
    putStrLn $ "MainThreadId: " ++ show tid
    where
        obs’  = obs ‘rxmap‘ (+1) ‘observeOn‘ newThread ‘rxmap‘ (+10)
        rxmap = flip fmap

{-
output>
    ThreadId 2: 12
    ThreadId 2: 13
    ThreadId 2: 14
    Completed
    MainThreadId: 1
-}
```

3.3.1 A note on the concept of time

Our discussion on push-based collections so far has not once mentioned the concept of time. This might appear strange, especially to the reader familiar with Functional Reactive Programming, where functions over continuous time are at the foundations of the theory. As we mentioned in Section 1.3.3, this dependency on continuous time comes at a great cost: commercial FRP libraries fail to successfully implement the concepts found in the theory as they cannot avoid simulating continuous time and approximating functions operating over it, being this concept inherently discrete in the context of computers.

Rx, on the other hand, completely sheds the notion of time from the notion of reactivity[?], shifting its focus, with the help of `Scheduler` s, to concurrency instead. Time still plays a role, although indirect, within the library: events are processed in the order they happen, and operators make such order is maintained, ultimately handing over to the user a stream of time-ordered events. See Section 3.5 for a broader discussion on this matter.

3.3.2 Impact on Operators

Previously we discussed how introducing concurrency is an orthogonal concept w.r.t. our reactive library. This statement is only partly true and falls short of its promises when we are dealing with combiner operators such as `(»=)` or `combineLatest` . These operators will not work at their full potential in a synchronous setting, due to the fact that subscribing to a stream will consume it entirely - or forever process, in the case of an infinite stream - before allowing an operator to subscribe to a different one, effectively making interleaving of events impossible.

The problem is gracefully solved with the introduction of `Scheduler` s, which, by allowing for `Observable` s to be executed on different contexts, indirectly make it possible for interleaving to happen and for combiner operators to work at their full potential. This comes at a - even though small - cost, which motivates our claim of partial orthogonality: the internal state of a combine operator needs to be synchronized and carefully handled as, with the introduction of concurrency, it can now be modified from different execution contexts.

3.4 Subscriptions

- now purely practical concern: stop stream while processing from anywhere, tell observable I dont want to observe anymore, i.e. the observer is done. - what is it: IO action/s | in theory can be seen as readerT, the subscription is always the same one, but you can add stuff to it. - who does it belong to: the observer, you get one for each subscribe. - how does it work: call unsubscribe, best effort stop work - impact on operators: a need for children -> now you can free resources -

impact on schedulers: kill the thread - write that it only works now that we have schedulers - edge enforcement - subject + takeuntil - why cannot be monoid, we need remove

---

With error handling, termination and schedulers, our implementation of Rx starts getting more and more usable. The next step is to add a mechanism that will allow us to stop an observable stream from anywhere in our program whenever we don't require it's data, i.e. a mechanism to tell the `Observable` that one of it's `Observer`s is not interested in receiving it's events anymore.

This is a purely practical concern and, as opposed to schedulers, adding a cancellation mechanism doesn't add expressive power to our reactive implementation. Yet, such feature is very useful when using Rx in commercial applications where resource handling gains much greater importance as opposed to a theoretical context.

Before diving into the details of the implementation let's try to better understand the intended functionality. The general idea is that we will introduce a new datatype, `Subscription` which will be returned from the subscribe function. The user will later be able to use it to call a newly introduced function, `unsubscribe` which will stop the associated stream from producing any more values.

It is important to understand that the notion of subscription is tight to the `Observer` rather than the `Observable`. The latter can, in fact, be subscribed to by various `Observer`s and our goal is to provide the user with the power to stop observing from any of them at any point in time.

So, what is a `Subscription` then? Conceptually, it could be a simple boolean recording whether an `Observer` is unsubscribed from the stream. In practice it is useful to augment it with some additional information. First of all, `Subscription`s can be used for resource cleanup at the termination - either forced or natural - of a stream. It is therefore useful to associate an `IO ()` action to the datatype, which will represent the computation to be executed once the stream is unsubscribed from. Another improvement is making the type recursive, i.e. allowing `Subscription`s to contain, add and remove children subscriptions. This will be extremely useful for internal coordination of operators such as `(>>=)` and for resource management in the context of schedulers - more on this in the next sub-subsections.

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.

We could just chain IO actions, but then we cant remove, which can lead to stack overflow with huge thunks in flatmap.

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

```
data Subscription = Subscription
    { onUnsubscribe   :: IO ()
    , _isUnsubscribed :: IORef Bool
    , subscriptions   :: IORef [Subscription]
    }
```

Now that we have a datatype to represent our subscriptions, we need to insert it into our reactive equation; the following snipped shows the impact of subscriptions on our reactive interface.

```
newtype Observable a = Observable
    { onSubscribe :: Observer a -> IO Subscription
    }

data Observer a = Observer
    { onNext       :: a -> IO ()
    , onError      :: SomeException -> IO ()
    , onCompleted  :: IO ()
    , subscription :: Subscription
    }
```

As we can see, `subscribe` now returns a `Subscription` and `Observer` contains one in its datatype.

The key functionality of the subscription mechanism lies in the `subscribe` function:

```haskell
import Data.Functor.Contravariant

newtype Observable a = Observable
    { onSubscribe :: Observer a -> IO Subscription
    }

data Observer a = Observer
    { onNext       :: a -> IO ()
    , onError      :: SomeException -> IO ()
    , onCompleted  :: 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.

```haskell
      OnNext a
    | OnError SomeException
    | OnCompleted

--we can remove returning the subscription to the ouside
--since the behaviour is reproducible with operators
--e.g. subject+takeUntil
subscribe :: Observable a -> Observer a -> IO Subscription
subscribe obs obr = do
    s <- createSubscription (print "unsubscribed")
    let
        safeObr = observer safeOnNext safeOnError safeOnCompleted
        safeOnNext a = do
            b <- isUnsubscribed s
            unless b $ obr (OnNext a)
        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.

3.5 The Reactive Contract

Ask Erik about this.

- Discuss how threadpool scheduler breaks monad laws. - Remember best effort cancellation.

Todo list

APPENDIX A

APPENDIX A

Begins an appendix

A.1 Categorical Duality

A.2 Continuations

A.3 (Co)Products

A.4 (Un)Currying

A.5 (Covariance & Contravariance

A.6 Functors

APPENDIX B

APPENDIX B

Begins an appendix

Formal RX Implementation

Bibliography

[1] *Scopus search: Reactive programming 1960-2016*.

[2] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter, *A survey on reactive programming*, ACM Computing Surveys (CSUR), 45 (2013), p. 52.

[3] A. Benveniste and G. Berry, *The synchronous approach to reactive and real-time systems*, Proceedings of the IEEE, 79 (1991), pp. 1270–1282.

[4] J. Edwards, *Coherent reaction*, in Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications, ACM, 2009, pp. 925–932.

[5] 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.

[6] B. Furht and A. Escalante, *Handbook of cloud computing*, vol. 3, Springer, 2010.

[7] E. Gamma, *Design patterns: elements of reusable object-oriented software*, Pearson Education India, 1995.

[8] T. Gowers, J. Barrow-Green, and I. Leader, *The Princeton companion to mathematics*, Princeton University Press, 2010.

[9] C. H. Group, *Covariance, contravariance, and positive and negative position*.

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

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

[12] E. Meijer, *Reactive streams keynote*.
LambdaJam, 2014.

[13] E. MEIJER, K. MILLIKIN, AND G. BRACHA, *Spicing up dart with side effects*, Queue, 13 (2015), p. 40.

[14] J. NEWBERN, *All about monads*.

[15] TEL, *What's up with contravariant?*