The Essence of Reactive Programming

A Theoretical Approach

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

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*

As we saw in Chapter 1, the `Iterable` interface 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. In this chapter we are going to formalize the intuition that there exists a duality relation between interactive and reactive programs, as well as between pull and push models of computations, by deriving the `Observable` interface starting from its dual counterpart, the `Iterable`.

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.

2.1 Iterables

An `Iterable` is a programming interface 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 introduce the `Iterable` as the root interface for standard pull collections APIs - exposing concrete implementations such as maps, sets, indexed sequences and so on.

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.

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

data Iterator a = Iterator
    { hasNext :: () -> Bool
    , next    :: () -> IO a
    }

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

-- C# IEnumerable
newtype IEnumerable a = IEnumerable
    { getEnumerator :: () -> IO (IEnumerator a)
    }

data IEnumerator a = IEnumerator
    { moveNext :: () -> IO Bool
    , current  :: () -> a
    }
```

Although the essence of the pattern is preserved by 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 traversing 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:

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

data Iterator a = Iterator
    { moveNext :: () -> IO Bool
    , current  :: () -> 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` and 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.

```haskell
newtype Iterable a = Iterable
    { getIterator :: () -> IO (Iterator a)
    }
newtype Iterator a = Iterator
    { moveNext :: () -> IO (Either SomeException (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 data types and express our interfaces as simple types.

```haskell
type Iterable a = () -> IO (Iterator a)
type Iterator a = () -> IO (Either SomeException (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`.

```haskell
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, theoretically, 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.

When looking at the `Iterator` type from an object oriented perspective, the reader should notice a strict similarity to a *getter* function - i.e. a lazy producer of values: iterators are, in fact, nothing more than getters of values of type `a`. The `Iterable`, on the other hand, is a function that enables the user to get an `Iterator`, i.e. a getter of a getter of `a`.

This correspondence will turn out to be very insightful later on in our discussion, where we will observe that `Observable` is nothing more that a setter of setters, another instance of duality in our formalization.

When looking at the relation between the `Iterator` type and its base component, `a`, we can observe how they are bound by a covariant relation:

```
A <: B
-----------------------------------
() -> IO A <: () -> IO B
```

The intuition can be easily understood when we think of an iterator as a drink vending machine, i.e. a function which, whenever called, will give back a drink:

```
Coke <: Drink
-------------------------------
VendingM Coke <: VendingM Drink
```

If coke is a subtype of drink, then whenever I am asked for a drink vending machine, I can hand out a coke vending machine without incurring in any troubles with the person who asked, as that machine will correctly provide drinks - even though they will always be coke - whenever prompted for one.

With `Iterator` being a getter itself, it should be clear how covariance plays the same role as with `Iterable`.

To formally prove the intuition of a covariant relation, we instantiate the `Iterable` / `Iterator` types to a covariant `Functor` and prove its laws in the following snippet.

```
{-
    Using a type synonim insead of Haskell's newtypes,
    in order to avoid clutter in our proofs:

    type Iterator a = () -> IO a

    fmap :: (a -> b) -> Iterator a -> Iterator b
    fmap f ia = \() -> ia () >>= return . f

    identity:
          fmap id                                          -- eta abstraction
        = \ia -> fmap id ia                                -- definition of fmap
        = \ia -> \() -> ia () >>= return . id              -- application of id
        = \ia -> \() -> ia () >>= return                   -- IO monad right identity*
        = \ia -> \() -> ia ()                              -- eta reduction
        = \ia -> ia                                        -- definition of
        = id

    composition:
          (fmap p) . (fmap q)
        = \ia -> ((fmap p) . (fmap q)) ia
        = \ia -> fmap p (fmap q ia)
        = \ia -> fmap p (\() -> ia () >>= return . q)
        = \ia -> \() -> (\() -> ia () >>= return . q) () >>= return . p
        = \ia -> \() -> ia () >>= return . q >>= return . p
        = \ia -> \() -> ia () >>= \a -> (return . q) a >>= return . p
        = \ia -> \() -> ia () >>= \a -> return (q a) >>= return . p
        = \ia -> \() -> ia () >>= \a -> (return . p) (q a)
        = \ia -> \() -> ia () >>= \a -> (return . p . q) a
        = \ia -> \() -> ia () >>= \a -> return ((p . q) a)
        = \ia -> fmap (p . q) ia
        = fmap (p . q)

    * monad right identity:
        m >>= return = m

      monad left identity:
        return a >>= f = f a

      defintion of (.):
        (.) :: (b -> c) -> (a -> b) -> a -> c
        (f . g) a = f (g a)
-}
```

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

    fmap :: (a -> b) -> Iterable a -> Iterable b
    fmap f iia = \() -> iia () >>= return . fmap f

    identity:
          fmap id                                       -- eta abstraction
        = \iia -> fmap id iia                           -- definition of fmap
        = \iia -> \() -> iia () >>= return . fmap id    -- Iterator identity law
        = \iia -> \() -> iia () >>= return . id         -- application of id
        = \iia -> \() -> iia () >>= return              -- IO monad right identity
        = \iia -> \() -> iia ()                         -- eta reduction
        = \iia -> iia                                   -- definition of id
        = id

    composition:
          (fmap p) . (fmap q)                                                   --
        = \iia -> ((fmap p) . (fmap q)) iia                                     --
        = \iia -> fmap p (fmap q iia)                                           --
        = \iia -> fmap p (\() -> iia () >>= return . fmap q)                    --
        = \iia -> \() -> (\() -> iia () >>= return . fmap q) () >>= return . fmap p    --
        = \iia -> \() -> iia () >>= return . fmap q >>= return . fmap p         --
        = \iia -> \() -> iia () >>= \ia -> (return . fmap q) ia >>= return . fmap p    --
        = \iia -> \() -> iia () >>= \ia -> return (fmap q ia) >>= return . fmap p      --
        = \iia -> \() -> iia () >>= \ia -> (return . fmap p) (fmap q ia)         --
        = \iia -> \() -> iia () >>= \ia -> (return . fmap p . fmap q) ia         --
        = \iia -> \() -> iia () >>= return . fmap p . fmap q                     --
        = \iia -> \() -> iia () >>= return . fmap (p . q)                        --
        = \iia -> fmap (p . q) iia                                              --
        = fmap (p . q)
-}

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

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

For the sake of completeness, it is worth mentioning that `Iterable` is, among others, also an instance of Applicative Functor and Monad. Although certainly interesting from a theoretical perspective, showing these instances and proving the associated laws goes beyond the scope of this work. Nontheless, we will see in the next section how these concepts are relevant in expressing and motivating 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 ()
```

Note how the side effects are bound to function application rather than values, hence their flipped position in the `Observable` type.

The newly derived types are relatively easy to read and understand: `Observer` is simply a function

that, given a value of type `a` will handle it somehow, producing side effects; the `Observable`, on the other hand, is responsible for producing such values of type `a` and feeding them to the `Observer` it has been given as an argument.

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. Referring back to our previous example involving cokes and drinks, we can now think of the `Observer` as a a recycling machine:

```
1                    Coke <: Drink
2          ----------------------------------
3          RecyclingM Drink <: RecyclingM Coke
```

Our intuition tells us that this time, a recycling machine that can only handle coke cannot be used in place of one that needs to handle any type of drinks, as it would fail at its task whenever a drink that is not a coke is fed into it. On the other hand, a recycling machine that works for any type of drink can be safely used in place of one that needs to handle cokes. This intuition bounds `Observer` and its base type `a` by a contravariant relation:

```
1                        A <: B
2          ----------------------------------
3          A -> IO () <: B -> IO ()
```

A more theoretical take on the matter involves the notion of type's positivity and negativity: 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.

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.

```
3  f   ::  a ->  b
4       = -a -> +b
5
6  g   ::  ( a ->  b) ->  c
7       = -(-a -> +b) -> +c
8       =  (+a -> -b) -> +c
9
10 observer
11     ::  a -> ()
12      = -a -> ()
13
14 observable
15     ::  ( a -> ()) -> ()
16      = -(-a -> ()) -> ()
17      =  (+a -> ()) -> ()
```

Before we formalize this claim, let's convince ourselves that `Observable`s effectively produces a value of type `a` by looking at an example:

```haskell
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/Iterator`, we can formally prove the covariant and contravariant relations between `Observable/Observer` and their base type `a` by instantiating them to `Functor` and `Contravariant (Functor)` respectively, as well as proving the laws.

```haskell
{-
    type Observer a = a -> IO ()

    contramap :: (a -> b) -> Observer b -> Observer a
    contramap f ob = ob . f

    identity:
          contramap id
        = \ob -> contramap id ob
        = \ob -> ob . id
        = \ob -> ob
        = id

    composition:
          (contramap p) . (contramap q)
        = \ob -> ((contramap p) . (contramap q)) ob
        = \ob -> contramap p (contramap q ob)
        = \ob -> contramap p (ob . q)
        = \ob -> (ob . q) . p
        = \ob -> ob . (q . p)
        = \ob -> contramap (q . p) ob
        = contramap (q . p)
-}

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

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

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

    fmap :: (a -> b) -> Observable a -> Observable b
    fmap f ooa = \ob -> ooa (contramap f ob)

    identity:
          fmap id
        = \ooa -> fmap id ooa
        = \ooa -> \ob -> ooa (contramap id ob)
        = \ooa -> \ob -> ooa ob
        = \ooa -> ooa
        = id

    composition:
          fmap p . fmap q
        = \ooa -> (fmap p . fmap q) ooa
        = \ooa -> fmap p (fmap q ooa)
        = \ooa -> fmap p (\ob -> ooa (contramap q ob))
        = \ooa -> \oc -> (\ob -> ooa (contramap q ob)) (contramap p oc)
        = \ooa -> \oc -> ooa (contramap q (contramap p oc))
        = \ooa -> \oc -> ooa ((contramap q . contramap p)  oc)
        = \ooa -> \oc -> ooa (contramap (p . q) oc)
        = \ooa -> fmap (p . q) ooa
        = fmap (p . q)
-}

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

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

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 special case of 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 deal with multiple values produced at different times in the future.

We can prove our claim by implementing the `Observable` interface using Haskell's Continuation Monad Transformer and observing how the unwrapping function `runContT` effectively hands us back our original type:

```
newtype ContT r m a :: * -> (* -> *) -> * -> *
runContT :: ContT r m a -> (a -> m r) -> m r

type Observable a = ContT () IO a
runContT :: ContT () IO a -> (a -> IO ()) -> IO ()
```

which

2.4 Termination and Error Handling

We began this chapter by progressively simplifying the `Iterable`'s interface in order to derive a type that would theoretically represent its very essence. One of the most important steps was setting aside concerns regarding termination an error handling of a collection. We are now going to reshape our reactive interfaces in order to address these concerns and appropriately describe the potential side effects directly in the types.

Informally, an `Observable` stream might not only produce one or more values, but it might gracefully terminate at a certain point in time or throw an exception and abruptly terminate whilst processing values. A more appropriate type for `Observer` is then the following:

```
newtype Observer a = Observer
    { onNext :: Either SomeException (Maybe a) -> IO ()
    }
```

Just as with `Iterable`, the introduction of `Either SomeException` allows us to express that the `Observer` can handle unexpected exceptions, while the `Maybe` reflects the possibility for a stream to end and propagate no more values.

Unfortunately, this type is very hard to read as well as understand for someone new to the topic. Looking at the matter from a functionality point of view, what we really need is for our CPS function - i.e. the `Observable` - to accept two additional continuations, one dealing with completion and one with exceptions. We can achieve this by first noticing that our type is nothing more than a coproduct - the same that we introduced previously for `Iterable` - of three base types: `a + SomeException + ()`. By utilizing the notion of product - the dual of coproduct - we can split the function handling the initial type into three different ones. This brings us to the final version of our reactive interfaces for push-base collections[1]

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

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

The `Observable` is now a special version of a CPS function accepting three continuation functions - embedded inside the `Observer` -, one for each effect an `Observable` can propagate: value,

[1] The two definitions are equivalent also from an implementation point of view, the first simulating the second though the use of pattern matching.

termination or exception.

2.5 Formalizing Observables

Throughout our discussion, we have mentioned multiple times the connection between `Observable` and CPS functions. In this section we are going to explore this matter and see how we these two concepts are effectively the same thing.

We are going to start from the notion that `Observable` is, at its essence, nothing more than a setter of setters, the result of applying the `Observer` function to itself. We can than express the `Observer` as a function `(!)` [2] that negates its type argument and results in a side-effectful computation.

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

When we apply the function to itself - i.e. substitute `a` for `!a` - we obtain our first definition of `Observable`, a CPS function that instantiates the result to `IO ()`.

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

As we have seen in the previous section, this definition is not expressive enough when we want to make explicit all the effects that are involved when dealing with push-base collections. It is therefore necessary to deviate from the standard definition of continuation and replace the inner application of `(!)` with a new function `(?)`, whose type embed the involved effects:

```
1 ?a  ::  Either Error (Maybe a) -> IO () -- termiantion and error handling
2 !?a :: (Either Error (Maybe a) -> IO ()) -> IO ()
```

Note how this definition is equivalent to the one used in the previous section, where we used the notion of products to unwrap the `(?)` function into three different ones, each addressing one of the possible effects.

Finally, we can now translate this pseudo-code into real Haskell, proving the equivalence between

[2] Regard the code used in this explaination as pseudo-Haskell.

`Observable` and CPS functions. We are going to do so with the help of Haskell's continuation monad transformer, a construct which allows to stack the functionality of the continuation monad on top of other monads. With it, we can define the `Observable` as a continuation transformer producing an empty result in the IO monad.

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

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

newObservable :: (Observer a -> IO ()) -> Observable a
newObservable = ContT

subscribe :: Observable a -> Observer a -> IO ()
subscribe = runContT
```

The code above uses a slightly different approach to expressing the three types of side effects an `Observer` has to deal with. Insead of using `Either` and `Maybe` from Haskell's libraries, we utilize our own custom datatype `Event`, a coproduct of values of type `a + SomeException + ()`; although the two definitions are equivalent in every aspect, the adopted one offers more clarity in terms of code readability.

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

```
18 obs = newObservable $ \observer ->
19     do observer (OnNext 1)
20        observer (OnNext 2)
21        observer OnCompleted
22
23 main :: IO ()
24 main = subscribe obs print
25
26 {-
27 output>
28     OnNext 1
29     OnNext 1
30     OnCompleted
31 -}
```

Notice how, being a continuation, an `Observable` only pushes values once subscribed to and acts as a suspended computation otherwise.

The code above, 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: whenever the user presses a key, an event containing the corresponding character is propagated to the `Observer` and will eventually be printed on the command line. 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.

```haskell
obs :: Observable Char
obs = newObservable $ \observer -> do
    keyboardCallback $= Just (\c p -> observer (OnNext c))

display :: DisplayCallback
display = do
  clear [ ColorBuffer ]
  flush

main :: IO ()
main = do
  (_progName, _args) <- getArgsAndInitialize
  _window <- createWindow "Observable Keyboard"
  subscribe obs (print . show)
  displayCallback $= display
  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: the `Observable` interface is not opinionated w.r.t. concurrency and therefore, 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 to orthogonally introduce concurrency in our reactive systems, altering the control flow of the data processing allowing the user to 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 spite of being very insightful for theoretical discussions on the properties and relations of `Observable` and the continuation monad, this implementation of the reactive types is impractical in the context of a full fledged API. In the next Chapter we will take the necessary steps to build the bridge between theory and practice, providing a reference implementation of `Observable` more adapt to be utilized in real world applications.

For a full implementation of a Reactive Library based on the ideas presented in this section, aimed at highlighting the strong connection between `Observable` and other already existing functional structures from which it composes, see Appendice B.

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 build the bridge between our theoretical definition of `Observable` and a concrete and usable implementation of a reactive library, to which we will refer to as Rx.

In the remained of the discussion, we are going to introduce the *Reactive Contract*, a set of assumptions on the reactive types our library is going to build upon, *Schedulers*, which will allow us to bring concurrency into our reactive equation, *Subscriptions*, used to implement a mechanism for premature stream cancellation and, finally, *Operators*, the means with which we will make our reactive streams composable.

For the sake of clarity and completeness, the following set of interfaces represents the starting point for our discussion:[1]

[1] `subscribe` has been renamed to `_subscribe` in order to both avoid naming conflicts later on in the discussion and reflect the fact that it should not be used directly by the user.

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

It is worth noting that even though the Observable's theoretical foundations lie in the realm of functional programming, the road to make it usable is full of obstacles that are often better tackled using imperative programming features, such as state. As much as I personally prefer a functional and pure approach to programming, I will favor, in the rest of the discussion, the solution that most clearly and easily solves the problem, be that functional or imperative.

3.1 The Reactive Contract

The `Observable` and `Observer` interfaces are somewhat limited, in they expressive power, to only argument and return types of they functions. The reactive library we are going to build is going to make more assumptions than the ones expressible by the type system. Although limiting, in a sense, the freedom with which the reactive interfaces can be utilized, this set of assumptions - the *Contract* - greatly facilitates reasoning about and proving correctness of reactive programs[?].

In later sections, we will refer back to these assumptions when discussing the actual implementation of our reactive library.

3.1.1 Reactive Grammar

The first assumption we are going to introduce involves restrictions on the emission protocol of an `Observable`. Events propagated to the `Observer` continuation will obey the following regular expression:

```
onNext* (onError | onCompleted)?
```

This grammar allows streams to propagate any number - 0 or more - of events through the `onNext` function, optionally followed by a message indicating termination, be that natural - through `onCompleted` - or due to a failure - through `onError`. Note how the optional nature of a termination message allows for the existence of infinite streams.

This assumption is of paramount importance as it guarantees that no events can follow a termination message, allowing the consumer to effectively determine when it is safe to perform resource cleanup operations.

3.1.2 Serialized Messages

Later in this chapter we will see how we can introduce concurrency in our reactive library through the use of the `Scheduler` interface. From a practical point of view, this means that it will be possible for different messages, to arrive to an `Observer` from different execution contexts. If all `Observer` instances would have to deal with this scenario, the code in our library would soon become cluttered with concurrency-related housekeeping, making it harder to maintain and reason about.

For this reason, we assume that messages will always arrive in a serialized fashion. As a consequence, operators that deal with events from different execution contexts - e.g. combiner operators - are required to internally perform serialization.

3.1.3 Best Effort Cancellation

The next assumption involves premature stream cancellation. We will later introduce the `unsubscribe` function, used to activate this mechanism and stop observing an `Observable`; the assumption here is that whenever `unsubscribe` is invoked, the library will make a best effort attempt to stop all the ongoing work happening in the background. The reason is simple: it is not always safe to abort work that is being processed - e.g. database writes. Although the library might still complete the execution of pending work, its results are guaranteed not to be propagated to any `Observer` that was previously unsubscribed.

This may leak, if not properly used, e.g. db writes on different schedulers. Do we care?

3.1.4 Resource Cleanup after termination

3.1.5 Observers are use and throw

3.2 Concurrency with Schedulers

At the end of Section 2.5 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 [2].

```
31 data Scheduler = Scheduler
32     { _schedule      :: IO () -> IO ()
33     , _scheduleDelay :: IO () -> TimeInterval -> IO ()
34     }
```

`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 actions to a new, dedicated thread.

```
36 newThread :: IO Scheduler
37 newThread = do
38     ch  <- newTChanIO
39     tid <- forkIO $ forever $ do
40         join $ atomically $ readTChan ch
41         yield
42     return $ Scheduler (schedule ch) (scheduleD ch)
43         where
44             schedule  ch io =
45                 atomically $ writeTChan ch io
46             scheduleD ch io d = do
47                 threadDelay $ toMicroSeconds d
48                 schedule ch io
```

The `newThread` function gives us a side effectful way of creating a `Scheduler` by generating a new

[2] 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.

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.

```
50 observeOn :: Observable a -> IO Scheduler -> Observable a
51 observeOn o sched = Observable $ \obr -> do
52     s <- sched
53     subscribe o (f s obr)
54         where
55             f s downstream = Observer
56                 {   onNext      = void . _schedule s . onNext downstream
57                 ,   onError     = void . _schedule s . onError downstream
58                 ,   onCompleted = void . _schedule s $ onCompleted downstream
59                 }
```

```haskell
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
-}
```

3.2.1 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 sure such order is maintained, ultimately handing over to the user a stream of time-ordered events.

3.2.2 A Note on Orthogonality

Previously we discussed how concurrency is an orthogonal concept w.r.t. Rx - i.e. introducing concurrency does not affect or pollute the definition of our reactive interfaces. This statement is only true from a abstract point of view, falling short of its promises when looking from an implementation perspective, in particular, when dealing with combiner operators (see Section 3.4) 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 the 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 cost: combiners operators are required to perform message serialization (see assumption 3.1.2) as well as internal state synchronization as, with the introduction of concurrency, messages and state changes can now originate from different execution contexts.

3.3 Subscriptions

With schedulers, we are now able to handle observable streams from different execution contexts. The next step in making Rx ready for a production environment is to add a mechanism that will allow us to stop a stream from anywhere in our program, whenever we don't require it's data anymore - i.e. a mechanism that will allow the user to communicate to the `Observable` that one of it's `Observer`s is no longer interested in receiving its events.

Is the following clear?

We discussed in the previous section how schedulers effectively boost the expressive power of our reactive expressions by introducing concurrency and interleaving among events originating from different streams. Introducing a cancellation mechanism, on the other hand, is a purely practical concern: although very useful from a practical perspective, especially in the context of resource management, it doesn't impact expressive power from a *reactive* point of view.

The means by which we are going to introduce a cancellation mechanism inside our reactive equation is though the `Subscription` datatype. From a functionality point of view, what we are aiming for is for the `_subscribe` function to hand back a `Subscription` whenever invoked; users will later be able to use this `Subscription` in order to prematurely cease the observation of a stream.

The first step in designing a new feature is to understand how the already existing interfaces will be affected by the newly introduced one; starting from our informal definition of `Observable` from section 2.5, let's now define a new function `(%)`, which incorporates the notion of returning a `Subscription` and see how this is going to affect our types:

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

With this change, each execution of the `Observable` function now returns a `Subscription`, a means for the user to prematurely terminate the processing of the stream.

The next question is the following: to whom does a `Subscription` belongs to? The key observation in addressing this question is that an `Observable` can be subscribed to by multiple `Observer`s; our goal is to provide a mechanism that will allow for a fine-grained control over which `Observer` is supposed to stop receiving events. The answer is then straightforward: the notion of subscription is tight to that of observer. The following snippet reflects this observation:

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

Let's quickly summarize what we have discussed so far: a subscription is some object which will allow us to prematurely stop observing a specific stream; since any stream can be subscribed to by

multiple observers, we need to associate subscriptions to observers as opposed to observables. Lastly, a subscription is returned every time an observer is subscribed to a stream through the `_subscribe` function. The following modifications to our reactive interfaces reflect these ideas:

_subscribe should return IO () no IO Subscription

```
12 newtype Observable a = Observable
13     { _subscribe :: Observer a -> IO Subscription
14     }
15
16 data Observer a = Observer
17     { onNext       :: a -> IO ()
18     , onError      :: SomeException -> IO ()
19     , onCompleted  :: IO ()
20     , subscription :: Subscription
21     }
```

do not subscribe an observer to 2 different streams

So far we have talked a lot about `Subscription`s, yet we haven't clarified what the type really looks like. The general idea is to have `Subscription` record the state of the `Observer` w.r.t. the `Observable` it is subscribed to - be that subscribed or unsubscribed. This can be easily achieved with a variable `_isUnsubscribed :: IORef Bool` initialized to `False`, indicating that the associated `Observer` is initially not unsubscribed.

From a practical point of view, it is useful to augment `Subscription` with some additional functionality. The following code shows a definition of `Subscription` which incorporates an `IO ()` action to be executed at unsubscription time. This is particularly useful when we want to associate resource cleanup actions to the termination - be that forced or natural - of a stream observation. Additionally, it is useful to make the type recursive, allowing `Subscription`s to contain other values of the same type. This will be extremely useful for internal coordination of operators such as `(>>=) :: Monad m => m a -> (a -> m b) -> m b`, where each value will spawn and subscribe a new `Observable`, whose subscription should be linked to the original one. Section 3.4 will extensively discuss this matter.

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

We will see, later on, how we can further augment this interface in order to incorporate additional functionality useful for internal operator coordination as well as management of schedulers.

It's now time to introduce the two functions at the hearth of the whole cancellation mechanism: `unsubscribe` will take care of modifying the state carried by the `Subscription` - i.e. setting `_isUnsubscribed` to `True` - as well as execute the associated `IO ()` action, whereas `subscribe` will simply act as a proxy for the original `_subscribe` function from the `Observable` interface.

```haskell
subscribe :: Observable a -> Observer a -> IO Subscription
subscribe obs obr = _subscribe obs safeObserver
    where
        safeObserver = Observer safeOn safeOe safeOc s
        s            = subscription obr
        safeOn a     = ifSubscribed $ onNext obr a
        safeOe e     = ifSubscribed $ finally (onError obr e) (unsubscribe s)
        safeOc       = ifSubscribed $ onCompleted obr >> unsubscribe s
        ifSubscribed = (>>=) (isUnsubscribed s) . flip unless

unsubscribe :: Subscription -> IO ()
unsubscribe s = do
    writeIORef (_isUnsubscribed s) True
    onUnsubscribe s
```

The `safeObserver` utilized by the `subscribe` function is of crucial importance to the functionality of our library: its implementation, in fact, embeds two of the reactive contract assumptions introduced previously. The safe onNext/onError/onCompleted functions implement the subscription mechanism, preventing, through the `ifSubscribed` function, events from propagating to the underlying `Observer`, once the related `Subscription` has been unsubscribed. By doing so, it is easy to see how assumption 3.1.3 is satisfied: unsubscribing from a stream does not force the stop of any outstanding work, yet it is made sure that any result produced after unsubscribing, if any, will not

be delivered to the downstream `Observer` - i.e. the `Observer` supplied by the user. Additionally, `safeObserver` allows the enforcement of the reactive grammar seen in assumption 3.1.1; this is done by calling `unsubscribe` as soon as the first termination message - be that `onError` or `onCompleted` - arrives, effectively preventing any additional event from being propagated.

Now that we have a clear idea of how the subscription mechanism is supposed to work and how it is integrated into our library, let's take a look at a few observations and concerns that involve it.

3.3.1 Impact on Schedulers

In the previous sections we discussed a simplified version of the `Scheduler` interface that was glossing, without loss of generality, over details regarding `Subscription`s. In practice, it is useful to associate `Subscription`s not only to `Observer`s but to `Scheduler`s as well.

```
95 data Scheduler = Scheduler
96     { _schedule      :: IO () -> IO Subscription
97     , _scheduleDelay :: IO () -> TimeInterval -> IO Subscription
98     , subscription   :: Subscription
99     }
```

With this version of the interface, each scheduled action returns a `Subscription`, offering fine grained control over the actions to be executed; at the same time, a `Subscription` is also associated to the `Scheduler` as a whole, allowing the user to perform cleanup actions on the `Scheduler` itself once `unsubscribe` is called. This is best shown with a new example implementation of the newThread scheduler and observeOn operator:

```
101 newThread :: IO Scheduler
102 newThread = do
103     ch  <- newTChanIO
104     tid <- forkIO $ forever $ do
105         join $ atomically $ readTChan ch
106         yield
107     sub <- createSubscription (killThread tid)
108
109     return $ Scheduler (schedule ch) (scheduleD ch) sub
110         where
111             schedule  ch io =
112                 atomically $ writeTChan ch io
113                 emptySubscription
114             scheduleD ch io d = do
115                 threadDelay $ toMicroSeconds d
116                 schedule_ ch io
117
118 observeOn :: Observable a -> IO Scheduler -> Observable a
119 observeOn o schedIO = Observable $ \obr -> do
120     sched <- schedIO
121     sub   <- subscription obr
122     liftIO $ addSubscription sub (subscription sched)
123     _subscribe o (f s obr)
124         where
125             f s downstream = Observer
126                 {   onNext       = void . _schedule sched . onNext downstream
127                 ,   onError      = void . _schedule sched . onError downstream
128                 ,   onCompleted  = void . _schedule sched $ onCompleted downstream
129                 ,   subscription = subscription downstream
130                 }
```

The code is mostly equal to the one presented in section 3.2. The most relevant change can be found at line 107, where we create a subscription for the newThread scheduler with an action that simply kills the thread[3]. On line 122 we then add this subscription to the one carried by the downstream observer. In this way, unsubscribing from the downstream subscription will trigger a waterfall effect that will eventually unsubscribing the scheduler's one as well, effectively killing the thread associated

[3]Absolutely not safe, but it's good enough for the sake of our example.

to it.

On a last note regarding the relationship between schedulers and subscriptions, it is worth mentioning how the subscription mechanism only works in the presence of schedules. As we mentioned before, in fact, Rx is synchronous by default in the processing of its data. This means that the program would return from the invocation of the `subscribe` function only after it has fully processed the stream, effectively rendering the subscription mechanism ineffective, as it would not be possible to invoke `unsubscribe` whilst the `Observable` is active. With the introduction of schedules and different execution contexts, this problem disappears and the mechanism works as intended.

3.3.2 Formalizing Subscriptions

In Section 2.5 we saw how an `Observable`, at its essence, is nothing more than an instance of the Continuation Monad, where the three types of events that can occur are materialized into a single datatype, `Event a`, as opposed to being handled by three different continuations.

In the discussion that follows we are going to try and understand what the essence of a subscription is and how it relates to our formal definition of observable as a continuation.

As we mentioned before, a subscription is strongly tight to the notion of observer, as an observable can be subscribed-to multiple times. Although, we can be more specific than this and notice that a subscription is actually tight to the execution of an observable. These two takes on subscriptions are effectively the same thing: an observer can only be subscribed a single time to a single observable (see assumption 3.1.5), creating the unique link between the subscription and a single execution of the observable function. This perspective is very insightful, as it hints to the fact that a subscription should be immutable within the context of an observable execution. Another observation is that the subscription needs to be retrievable from an observable for a number of reasons, the most important of which being to check whether the subscriber is unsubscribed before pushing any additional events.

The properties of subscription that we just discussed are very similar to the idea of environment variables, shared by computations yet immutable in their nature. The Haskell programming language exposes a monad construct for such computations, the Reader - Environment - Monad: in the remained of this section, we are going to formalize the essence of subscriptions as a Reader monad transformer on top of our previous definition of observable as a continuation.

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

subscribe :: Observable a -> Observer a -> IO Subscription
subscribe obs obr = do
    subscription <- emptySubscription
    safeObserver  = enforceContract obr
    runContT (runReaderT obs s) safeObserver
    return s

enforceContract :: Observer a -> Observer a
enforceContract obr = ...
```

This formalization is very insightful under many points of view: first of all, in the same way as schedulers, it is completely orthogonal to the definition of observable we previously had, the definition did not change, but the mechianism was, in a way stuck on top of it. THis is very clear when looking at line 8 in the subscribe function: we first run the reader transformer, getting a continuation monad out of it, which will have the environment variable available to use. Notice how each call to subscribe will effectively create a new subscription and pair it to the execution of the observable.

Notice how, even if the subscription is immutable by itself. the defintiion of the datatype still allows us to

A natural question now is: why didn't we implement subscriptions in this way also before, in the "real world" implementation? there we had to change the interfaces, making it not orthogonal. We did this only for clarity, the interfaces look more clear than reading readerT, especially if we want to use that implementation as a reference for other languages. On the other hand, here we want to focus on the essence of the subscription mechanism, therefore all the fancy FP stuff.

Notice how this formalization only takes care of the aspect of stitching subscriptions to observables, the business logic of the mechanism is still the same as before, embedded in the `enforceContract` function.

As a final point, this is obviously not the only way we can formalize this mechanism, many other definitons have been tried out, yet this one has turned out to be the best one. We could have used stateT:

```
type Observable a = StateT Subscription (ContT () IO) (Event a)
```

but this would have given too much power, and the possibility to mutate the subscription. Another version was:

```
Cont () (StateT Subscription IO) (Event a)
```

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,

---

- edge enforcement

- subject + takeuntil: do we really need subscriptions?

- why cannot be monoid, we need remove

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.

3.4 Operators

- what are operators? ways to transform a stream

- introduce lift, we can define operators on underlying type (make connection with continuation bind? formalObserver val = matchF val observer)

- functor, applicative, monad

- overview of other operators

- how does it work with subscriptions?

- formalizing operators: stateful objects, nothing to do about that. Drawing of continuations called

---

With schedulers and subscriptions we can handle asynchronous streams of data from different execution context and cease our observation whenever at any point in time. The last feature before we can consider our reactive library ready for use by developers is the introduction of a set of higher order functions that, given one or more `Observable`s, will allow access to its underlying elements - providing ways to transform, compose and filter them - while abstracting over the enclosing data structure. These higher order functions, also referred to as operators, are a common technique when it comes to abstractions over data structures; examples can be found in many commonly known programming languages, where data structures such as iterables, lists, trees, sets, ..., offer a wide range of functions - map, filter, flatmap, concat, ... - that allow the user to operate on its internal elements without any knowledge of the structure that contains them.

Back in section 2.3, we defined the Functor instance for the Observable type, effectively augmenting our reactive type with our first operator, `map :: (a -> b) -> Observable a -> Observable b`, allowing the user to transform streams of elements of type `a` into streams of elements of type `b` by applying the input function of type `a -> b` to each incoming element in the input stream.

The next operator we are going to introduce is `lift :: (Observer b -> Observer a) -> Observable a ->` This operator takes a function defined on the Observer level and lifts it to a more general context, that of Observable. This pattern is very common in the field of Functional Programming, and it is often used in order to provide an abstraction that facilitates accessing values nested inside container structures such as Functors, Applicatives or Monads. In the context of Observables, this function will

result very useful as we will be able to define operators on the Observer level and later make them accessible in the context of Observables simply by lifting the operation.

revise

```
lift :: (Observer b -> Observer a) -> Observable a -> Observable b
lift f ooa = Observable $ \ob -> onSubscribe ooa (f ob)
```

As an example, we repropose here the implementation if the Functor instance for Observable, this time using the `lift` function:

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

instance Functor Observable where
    fmap f = lift (contramap f)
```

The first thing we want to do then is explore those operators that would derive from call instances in the Haskell language. For the sake of completeness, here we repropose the implementation of functor for observable, this time using lift.

```
 functor code using lift
```

Next, we are going to implement an instance of the class Applicative. From a semantics point of view, this is nothing too hard, the first function, pure, needs to create and observ? out of ..., while (the other function) is a way to combine...

```
 applicative code
```

Now let's go one step further and note that we can also flatmap. Show implementaiton and prove the laws, how do I prove the laws?

```
19 -- Monad Laws
20 -- return a >>= k  =  k a
21 -- m >>= return  =  m
22 -- m >>= (x -> k x >>= h)  =  (m >>= k) >>= h
```

Operators can theoretically be infinite, as infinite are the transformations that can be done to elements of an observable stream. Practice shows, though, that a relatively small subset of operators and the composition of those, suffices to express the majority of the use cases encountered by users. This leverages the power of composability of operators, coming straight from FP, and is advantageous from a API design, since the library won't have to explode with new operators for each use cases.

Operators can be grouped into categories, by looking at their characteristics; it is not the purpose of this work to list every possible operator and its semantics, yet, for the sake of completeness, here are the most important categories[4]

put a table or something

1. Creation
2. Transformation
3. Filtering
4. Combining
5. Error Handling
6. Utility
7. Combining

3.4.1 About Lift

3.4.2 Formalizing Operators

3.5 Enforcing the Contract

[4] An implementation of these operators can be found at the repository associated with this work.

TBD

— Lewis Carol,
Alice in Wonderland

Bla bla bla

Future Work

Conclusion

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?*