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.

APRIL 2017

Abstract

What does it mean to be Reactive? The concept of Reactive Programming has gained much traction in the last few years as a paradigm well-suited for the development of asynchronous event-driven applications. Unfortunately, Reactive Programming has been at the center of much discussion, if not confusion, with regards to its definition, properties and identifying principles. In this work we are going to wield the most powerful tool available to software engineers, mathematics, in order to formally derive the reactive types and bring clarity to this much opinionated topic.

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 Engineering community has been continuously tested by an ever growing number of challenges related to management of increasingly large amounts of user data [17].

This phenomena is commonly referred to as Big Data. A very popular 2001 research report [28] 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 [16]. 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 [30].

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 [6]. 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 [32].

The goal of our work is to use mathematics as a tool to formalize the concept of reactive programming from a theoretical perspective. We are going to do so by utilizing constructs and ideas from functional programming and category theory with the purpose of formally deriving a set of types and interfaces embedding the essence of reactive programming. We will then continue with the development of a reference reactive library which builds upon the previously derived theoretical foundations.

Motivation

As we mentioned above, reactive programming's steep increase in popularity in the last few years [6] has come with a number of issues with regards to it's defining properties. Individual people, as well as industries, have been trying to push their own definition of reactive programming to the community, often placing their own interests before objectivity [32].

We find the current state of things to be unacceptable as it undermines the scientific foundations and reputation of our community and field. This lack of a scientific and formal analysis of the concepts involved in reactive programming gives motivation to the work and research presented in this report.

To the best of our knowledge, we are not aware of any previous work which analyses reactive programming from a theoretical standpoint or derives its types and interfaces though the use of mathematics. Our research will take a strictly formal and mathematical approach to the derivation of a theory around reactive programming, reinstating objectivity as the main protagonist in this much opinionated field.

Goals & Contributions

The goal of this work is to provide 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 on the matter.

This goal is achieved by providing a mathematical derivation of the reactive types, starting from their interactive counterparts and making use of theoretical concepts from category theory. These derived types are then used in the implementation of a formal reactive library where the purpose is showing how the theoretical definitions given to the various components can effectively be translated into working code.

Together with the formal definition of the paradigm, this work contributes to the the field of reactive programming with a reference implementation for a production level reactive library, as well as an highlight of the issues and challenges encountered when bridging from the theoretical foundations of reactive programming to a concrete implementation of a reactive API.

With the help of this report and the associated code repository, any software engineer interested in the topic should be able to understand the theoretical foundations behind the reactive paradigm and develop a reactive library in any language of choice.

Research Questions

The work presented in this report will focus on answering the following research questions:

- **Which class of problems does reactive programming solve? How does this relate to the real world libraries that claim to be reactive?**

  Before any attempts at a formalization can be carried out, we need to clearly identify the class of problems the reactive paradigm is fit for solving, understanding what are the issues and concerns such problems present, thus setting the basis for a formalization to be defined. Additionally, we are going to analyze the current libraries and APIs that claim to be reactive, and see how they relate to our definition.

- **How can we use existing mathematical and computer science theory in order to formally derive a definition for reactive programming?**

  Once we have a clear definition of the meaning of reactive programming and the class of problems it solves, we are going to look at existing theories in mathematics and computer science that would allow us to derive a set of types/interfaces representing the essence of reactive programming. In order to make our work sound, we will then need to prove the connection between the derived types and the definition resulting from the first research question.

- **How can we bridge from the derived theoretical foundations of reactive programming to a concrete API that, whilst maintaining its mathematical roots, is fit for use in a production environment?**

  Although appealing under multiple aspects, a set of interfaces is not concrete enough to have an impact in our daily lives as software engineers. The last step of our work will focus on building a reactive API directly from the theory discussed in the previous point, providing a reference point and set of good practices applicable to the implementation of a reactive library in any language of choice.

Related Work

This work mainly builds on top of the 2010 paper *Subject/Observer is Dual to Iterator* by Erik Meijer [29]. In his work, the author introduces the `Observable` interfaces and informally provides its derivation starting from the `IEnumerable` interface in `C#`. Together with a small number of related recorded talks and presentations [31–33], this paper is the only source that discusses the theoretical aspects tight to reactive programming.

Nevertheless, much related work and theories was found on the formalization of the semantics of a program, language or API that has aided in the development of the research presented in this thesis. The most popular approaches to formalizing computations are known as Operational and Denotational Semantics and were introduced by Christopher Strachey and Dana Scott in the late 1960s [38]. Operational semantics describes a program in terms of the computational steps needed in order to interpret the program, whereas a denotational semantics attaches a mathematical meaning to the components of the programs and abstracts away from any operational concern. Although mainly used to formalize the semantics of programming languages, these techniques have been proposed and employed in the development of programming libraries as well; Conal Elliott, promotor of the denotational design approach [12], makes use of denotational semantics in order to provide a precise definition of theory of Functional Reactive Programming [15].

Process Calculus and in particular ACP [8] - Algebra of Communicating Processes - is an algebraic approach to describing concurrent systems in term of communicating processes and their composition. This theory contains interesting aspects and properties that helped reasoning about the formalization of operators in this work. Inspiration for this task was also found in Kowalski and Sergot's Event Calculus [39], different from ACP in that the focus shifts from reasoning in terms of processes to reasoning in terms of events.

Alongside the mentioned theories and calculi, much inspiration for the research and methodology presented in this report was gained though work focused on the formal description of interactive libraries and constructs. To this end, it is worth mentioning Gibbons and Oliveira's formalization of the essence of the Iterator pattern [19], Kiselyov's work on Iteratees [27], Gonzale's work on the Pipes [20] library for the Haskell programming language, with a particular focus on a theory driven development, Capriotti's attempt to a Continuation monad based implementation of FRP [9], as well as the great amount of material on functional programming, Continuation monad, Free monad, monad transformers and so on found on HaskellWiki [23].

Overview

Chapter 1 introduces the scope of our research, providing a definition of reactive programming, the motivation and reasoning behind our research and an overview of the current technologies and APIs that claim to belong to the world of reactive programming. *Chapter 2* presents the mathematical derivation of the reactive types and interfaces, starting from the definition of `Iterable` and ending with that of `Observable`. *Chapter 3* builds the bridge between the formal definition of the reactive types and a production level implementation of the paradigm, highlighting the technical issues as well as analyzing the relations with the previously discussed formal definitions. *Chapter 4* concludes with final thought and future work.

Notation & Conventions

In the exposition of our work we will make use of Haskell as the reference programming language. This decision is motivated by the language's strong connection with mathematics and category theory, as well as it's clean syntax. These features will make the code both easy to read and explicit in the side effects that come into play in the various definitions. A minimal knowledge of Haskell's syntax - type declaration, lambda abstraction and `IO` monad - is assumed to be known by the reader in the exposition of this report.

All the code presented in this report, a minimal complete theoretical implementation and a reference implementation of a reactive library can be found at the associated code repository on Github - `https://github.com/Widar91/Thesis`.

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 [37]. A relevant and insightful definition was given by G. Berry in 1991 [7] 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 markets, 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 [34].

	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 [34], resulting in what it's commonly referred to as *Callback Hell* [11].

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.

1.3 Reactive Programming IRL

Interfaces are only as good as the implementations that back them up. In this section we are going to discuss and analyze the most commonly known 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.

1.3.1 Reactive Extensions

Reactive Extensions - also known as Rx - is the standard library for programming in a reactive way. Originally published by Microsoft as an API for the C# and Javascript languages, it was later ported to the JVM world by Netflix as an open source project, gaining much traction in the developers community and resulting in various implementations for the currently most commonly used programming languages [2].

The intuition and theory on which Reactive Extensions are built originated from the mind of Erik Meijer [29] and will be at the basis of the work developed in this thesis, where we will use mathematical constructs and derivations in order to prove the correspondence between the interfaces exposed by this library and the essence of reactive programming we will derive.

Although originally based on theoretically sound concepts, this polyglot family of libraries diverged from a purely reactive implementation, mainly due to their open source nature, independent development and, most importantly, to the lack of a unifying reference formalization of the reactive paradigm. This last aspect further motivates the need for our research.

Rx defines itself as a a library for composing asynchronous and event based (reactive) programs by using observable sequences [3]. At its core, it expose two interfaces, `Observable` and `Observer`. An `Observable` is the producer of a sequence of events which are pushed to an `Observer`, who will act upon them and produce side effects. Furthermore, the library offers a number of additional constructs such as `Subscription`, `Scheduler` and operators, that facilitate programming with asynchronous events and make the API more appealing for use in a production environment.

1.3.2 Reactive Streams

Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure [1]. As both the name and the description on the website [1] suggest, this API sets out to provide a standard set of interfaces addressing the class of problems identified previously as reactive.

The set of interface exposed by Reactive Streams is nearly identical to the Reactive Extensions' ones, the difference being an additional form of control over the producer of data, non-blocking back pressure. With this term, the promoters of Reactive Streams refer to a way for the consumer of the data to control the speed at which the producer will push its elements downstream.

As great as this sounds on paper, mathematics unfortunately proves it impossible: Reactive Streams are not reactive and back pressure is not applicable to the class of programming problems defined as reactive.

As Erik Meijer proves in his talk "Let Me Calculate That For You" at Lambda Jam 2014 [32], the interfaces exposed by the Reactive Streams initiative are equivalent - modulo naming conventions - to the more familiar `AsyncIterable`, a special version of `Iterable` that returns it's element to the caller in an asynchronous fashion. This allows for the implementation of back pressure, as the underlying model of computation is still pull-based, i.e. interactive.

A last point worth discussing before moving on is the claim that back pressure is not applicable to the class of problems we previously identified as reactive. As the reader might remember from Berry's definition, reactive programs interact with the environment at a speed at which determined by the environment and not by the program itself [7]; this definition makes the two concepts of reactive programs and back pressure incompatible.

From an informal perspective, it is easy to understand why: the speed at which events originated from reactive sources - such as mouse movements, stock ticks, GUI components and hardware sensors - occur is fully determined by the producer of such events - i.e. the environment. It would make no sense - and would be effectively impossible - for a program to ask a user to stop producing mouse movements or the stock market to slow down in producing stock ticks, because it cannot process its events fast enough. In such a context, a program is forced to to handle the overflow on its end, by taking actions such as buffering or dropping events.

The fact that Reactive Streams are ultimately not reactive does not make the API useless, yet it contributes to a general confusion and pollution in the terminology among the field of reactive programming.

1.3.3 Functional Reactive Programming

Functional Reactive Programming - also known as FRP - is a general paradigm for describing dynamic, time-varying information. Introduced by Conal Elliott in 1997 [15], it is precisely defined by a simple set of data types and associated denotational semantics.

As criticized by the author himself, the term has recently been used incorrectly to describe systems like Elm, Bacon, and Reactive Extensions [14]. Albeit the similar names, Functional Reactive Programming and Reactive Programming are two separate theories and differ from each other in certain fundamental aspects: where the former models time-varying values over continuous time, the latter is focused on asynchronous data streams and completely abstracts over the concept of time. For these reasons, we are not going to further discuss FRP in this report.

1.3.4 Reactive Manifesto

Whilst not being an API in and of itself, the Reactive Manifesto is worth a mention in our discussion, as it is often wrongly associated to the context of reactive programming.

The Reactive Manifesto [25] is a document that aims at providing a definition of reactive systems. With this term, the document refers to a set of architectural design principles for building modern systems that are prepared to meet the technical demands that applications face today. [26].

Due to overlapping terminology, the principles outlined by the Manifesto are often mixed or confused with those defining reactive programming, with the former focusing on the higher level of abstraction of architecture and design principles of application - targeting a more management-focused audience and lacking any type of scientificity - and the latter defining a set of interfaces aimed at solving a precisely defined class of problems.

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 [29], as well as between pull and push models of computations, by deriving the `Observable` interface - introduced in Section 1.3.1 - 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 [18].

The interface and semantics of `Iterable` s were first introduced by the Gang of Four though their Iterable/Iterator pattern [18]; 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 - e.g. IEnumerable (C#), Iterable (Java), Iterator/Generator (Python) - and slight differences in the types. Below we show two example definitions of these interfaces and their types.

```haskell
-- 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 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.

```
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, as it operates within the `IO` monad. We assume in the remainder of the discussion that this will never happen and a call to the function will always return an `Iterator` instance. The reason for this assumption is that `getIterator` is nothing more than a factory method for `Iterator`. The only way it could possibly throw an exception is if it fails instantiating the object, which could only happen in extreme cases - e.g. when the runtime does not have any memory left for allocation - hence the omission of `Either` in the type. Note that, even if the underlying collection does not exist, `getIterator` would still correctly return an `Iterator`, which would then throw once `moveNext` is called and access to a non-existing collection is attempted.

The next step is to forget about data types and express our interfaces as simple types. This is a simple simplification of Haskell's syntax which allows us to eliminate the type constructors introduced by the `newtype` and reason about `Iterable/Iterator` without any syntactic clutter.

```
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`. The purpose of this simplification is to make the discussion that follows easier to read and it's justified by the fact that the exceptions and termination play no role in the properties of `Iterable` we are going to analyze next. Note that setting these concerns aside is only temporary, they will be reintroduced once we have derived `Observable` later in the chapter.

```
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`. A proof of the associated `Functor` laws can be found in Appendix C

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

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


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 [21] (See Appendix 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`.

```haskell
{-
    () -> (() -> 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:

```
Coke <: Drink
```

```
-----------------------------------
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:

```
              A <: B
----------------------------------
    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.

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

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

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

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

```
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 [40] [22].

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. Once again, a proof of the associated laws can be found in Appendix C.

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

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


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 Appendix A).

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

The above code shows how `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 [35]; 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:

```haskell
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 ()
```

This equivalence is very important as it allows us to claim an instance of Applicative Functor and Monad for our derived type, `Observable`. These instances are inherited for free from the continuation monad, sparing us the burden of implementing them and proving all related laws.

```haskell
instance Applicative (ContT r m) where
    pure x  = ContT ($ x)
    f <*> v = ContT $ \c -> runContT f $ \g -> runContT v (c . g)

instance Monad (ContT r m) where
    return x = ContT ($ x)
    m >>= k  = ContT $ \c -> runContT m (\x -> runContT (k x) c)
```

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 would like is for our CPS function - i.e. the `Observer` - to be able to accept three continuations, one dealing with a proper value, one with completion and one with exceptions, as these are the three possible effects at play. 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, termination or exception.

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

2.5 Formalizing Observables

In section 2.3 we have shown how the essential type for `Observable` is effectively nothing more than a particular instance of the continuation monad. In this section we are going to explore this relation in further detail, introducing a notation which will help us keep track of the changes we will make to the original `Observable` type, ultimately showing how the resulting interface - that will be used in our final library - will consist in nothing more than a modified version of a CPS function.

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.

```
!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 ()`.

```
!!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:

```
?a  ::  Either Error (Maybe a) -> IO () -- termiantion and error handling
!?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 continuation, each addressing one of the possible effects.

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

In the next chapter we are going to further modify this definition with the inclusion of a cancellation mechanism.

This newer version of `Observable` is still implementable as an instance of the continuation monad, as the code below shows.

```
--   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`.

```
obs = newObservable $ \observer ->
    do observer (OnNext 1)
       observer (OnNext 2)
       observer OnCompleted

main :: IO ()
main = subscribe obs print
```

```
{-
output>
    OnNext 1
    OnNext 1
    OnCompleted
-}
```

Notice how, being a CPS function, 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.

```
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 reference 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 Appendix B.

Chapter 3

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

> *"In theory there is no difference between theory and practice; in practice there is."*
>
> — Nassim Nicholas Taleb, *Antifragile - Things that Gain From Disorder*

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.

The goal of this chapter is to provide the reader with a reference implementation of a reactive library as well as to highlight the challenges and issues that emerge when trying to build the bridge between theory and practice. The implementation choices presented below are in no way prescriptive, instead, they aim to describe the problem in the most clear way, in order to stimulate awareness rather than blindly guide the reader to a solution.

In the remainder 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]

```
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 making 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. As mentioned previously, Appendix B contains a reference implementation of a Reactive Library which implements the features presented in this chapter utilizing existing constructs from functional programming.

3.1 The Reactive Contract

The `Observable` and `Observer` interfaces are somewhat limited, in their 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 [5].

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

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

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 via `Subscription`s and the function `unsubscribe`, used in order to stop the observation of events from an `Observable`; we are going to assume 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.

3.1.4 Resource Cleanup After Termination

As we mentioned in assumption 3.1.1, the guarantee that no events will occur after the first termination message makes it possible to determine when resource cleanup operations are safe to perform. We will now make one step further and assume that resources *will* be cleaned up immediately after termination. This will make sure that any related side-effect will occur in a predictable manner.

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 [36] 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].

```
data Scheduler = Scheduler
    { _schedule      :: IO () -> IO ()
    , _scheduleDelay :: IO () -> TimeInterval -> 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 actions to a new, dedicated thread.

[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.

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

3.2.1 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. This dependency on continuous time comes at a great cost: commercial FRP libraries fail to successfully implement the concepts found in the theory [13] 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 [29], 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.

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. This design is closely related to Dispose pattern utilized in the .NET framework [10].

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` belong 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:

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

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 input 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.

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

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.

```

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
```

The `safeObserver` utilized by the `subscribe` function is of crucial importance to the functionality of our library and its the reason why we need to proxy the original `_subscribe` function: 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.

Note that, with the current implementation of the subscription mechanism, an `Observer` can only be subscribed once and only to a single `Observable`, as, once its `Subscription` is unsubscribed, the `_isUnsubscribed` field is never reset to `False`. This convention is shared by many already existing implementations of reactive libraries such as the onces under the ReactiveX umbrella [4].

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.

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

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:

```haskell
newThread :: IO Scheduler
newThread = do
    ch  <- newTChanIO
    tid <- forkIO $ forever $ do
        join $ atomically $ readTChan ch
        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 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 schedulers. 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

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

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 the subscription mechanism 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, 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 model the subscription mechanism 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

```

```
11 enforceContract :: Observer a -> Observer a
12 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 original definition did not change, yet the mechanism was, in a way, glued on top of it. This observation becomes very clear when looking at line 8 in the above snippet: the `subscribe` function first runs the reader transformer, resulting in continuation monad that will have the environment variable available within its context. Notice how each call to subscribe will effectively create a new subscription and pair it to the execution of the observable.

A natural question now is: why didn't we use this technique for implementing subscriptions in the "real world" implementation from the previous paragraph? There, we had to change the definitions of our interfaces, losing orthogonality as a consequence. The reason is simply 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, the goal of this paragraph is focusing on the essence of the subscription mechanism, hence the use of theory-related constructs such as monads.

Notice how this formalization only focuses on augmenting the definition of observables with subscriptions. The actual logic of the mechanism remains unchanged and is abstracted away through the `enforceContract` function.

As a final point, it is worth noting that what we presented so far is obviously not the only way we can formalize the subscription mechanism. Many other definitions have been tried out during the course of this work, yet all of the others ended up granting too much or too less power to the resulting mechanism and were therefore discarded. Examples include:

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

3.4 Operators

With schedulers and subscriptions we can handle asynchronous streams of data from different execution context and cease our observation 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, etc - that allow the user to operate on its internal elements without requiring any knowledge of the structure that contains them.

3.4.1 Functor, Applicative, Monad

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`. This operator takes a function defined on the Observer level and lifts it to a more general context, that of Observable[24]. 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.

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

As an example, we re-propose here the implementation of the Functor instance for Observable, this time using the `lift` function:

```haskell
instance Contravariant Observer where
    contramap f ob =
        Observer (onNext ob . f) (onError ob) (onCompleted ob) (subscription ob)

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

As we discussed in section 2.3, the `Observable` type, at its essence, is an instance of Applicative Functor and Monad from a cathegory theory perspective. This observation motivates our next effort of trying and defining such instances on the latest version of our interface. We will not bother anymore with proving the associated laws, as it would be a very difficult task, given the complicated nature of the implementation of such functions which now needs to handle subscriptions as well as synchronized state due to schedulers.

```
instance Applicative Observable where
    pure x = Observable $ \obr -> do
        onNext obr x
        onCompleted obr
        return $ subscription obr
    (<*>) = combineLatest ($)

combineLatest :: (a -> b -> r) -> Observable a -> Observable b -> Observable r
combineLatest combiner oa ob = Observable $ \downstream ->
    let
        onNext_ :: TMVar t -> TMVar s -> (t -> s -> IO ()) -> t -> IO ()
        onNext_ refT refS onNextFunc valT = join . atomically $ do
            _ <- tryTakeTMVar refT
            putTMVar refT valT
            maybeS <- tryReadTMVar refS
            return . when (isJust maybeS) $ onNextFunc valT (fromJust maybeS)

        onError_ :: TMVar Bool -> SomeException -> IO ()
        onError_ hasError e = join . atomically $ do
            hasE <- takeTMVar hasError
            putTMVar hasError True
            return . when (not hasE) $ onError downstream e

        onCompleted_ :: TMVar t -> TMVar Bool -> TMVar Int-> IO ()
        onCompleted_ refT hasCompleted hasActive = join . atomically $ do
            emptyT <- isEmptyTMVar refT
            hasC   <- takeTMVar hasCompleted
            active <- takeTMVar hasActive
            putTMVar hasCompleted True
            putTMVar hasActive (active - 1)
```

```
            return . when (emptyT && not hasC || active - 1 == 0) $
                onCompleted downstream
    in do
        active      <- newTMVarIO 2
        refA        <- newEmptyTMVarIO
        refB        <- newEmptyTMVarIO
        hasError    <- newTMVarIO False
        hasCompleted <- newTMVarIO False
        let obrA = Observer (onNext_ refA refB (fa downstream))
                            (onError_ hasError)
                            (onCompleted_ refA hasCompleted active)
                            (subscription downstream)
        let obrB = Observer (onNext_ refB refA (fb downstream))
                            (onError_ hasError)
                            (onCompleted_ refB hasCompleted active)
                            (subscription downstream)
        _subscribe ob obrB
        _subscribe oa obrA
        where
            fa downstream = (\a b -> onNext downstream (combiner a b))
            fb downstream = (\b a -> onNext downstream (combiner a b))
```

The function `pure` does nothing more than wrapping a value into an `Observable` whereas `(<*>)` applies the most recent function emitted by the first `Observable` to the most recent element emitted by the second one. The implementation utilizes the more general `combineLatest` operator, which allows to combine two streams into one by emitting an item whenever either of the two emits one - provided that each of them has emitted at least one.

Last but not least comes the Monad instance for our Observable type:

```
instance Monad Observable where
    return = pure
    (>>=)  = flatMap

flatMap :: Observable a -> (a -> Observable b) -> Observable b
flatMap obs f = Observable $ \downstream ->
```

```
    let
        onNext_ gate activeRef hasError hasCompleted val = do
            atomically $ modifyTVar activeRef (+1)
            s <- emptySubscription
            let inner = Observer (innerOnNext_)
                                 (onError_ hasError)
                                 (innerOnCompleted_ s)
                                 (s)
            addSubscription (subscription downstream) (subscription inner)
            handle (onError_ hasError) . void $ _subscribe (f val) inner
                where
                    innerOnNext_ v = do
                        withMVar gate $ \_ -> onNext downstream v

                    innerOnCompleted_ s = do
                        cond <- atomically $ do
                            c <- readTVar hasCompleted
                            modifyTVar activeRef (subtract 1)
                            a <- readTVar activeRef
                            return (c && a == 0)
                        if cond
                            then onCompleted downstream
                            else removeSubscription (subscription downstream) s

        onError_ hasError e = do
            cond <- atomically $ do
                e <- swapTVar hasError True
                return (not e)
            when cond $ onError downstream e

        onCompleted_ activeRef hasCompleted = do
            cond <- atomically $ do
                c <- swapTVar hasCompleted True
                a <- readTVar activeRef
                return (not c && a == 0)
            when cond $ onCompleted downstream

    in do
```

```haskell
        gate        <- newMVar ()
        activeRef   <- newTVarIO (0 :: Int)
        hasError    <- newTVarIO False
        hasCompleted <- newTVarIO False

        _subscribe obs $ Observer (onNext_ gate activeRef hasError hasCompleted)
                                  (onError_ hasError)
                                  (onCompleted_ activeRef hasCompleted)
                                  (subscription downstream)
```

Note how this implementation if `(>>=)` shows the need for the introduction of children subscriptions that we discussed in section 3.3: whenever the outer observable is unsubscribed, we want to automatically unsubscribe any observable that has previously been created by the function passed to `(>>=)`.

3.4.2 The Essential Operators

Operators can theoretically be infinite in number, 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 these, suffices to express the majority of the use cases encountered by users. Leveraging the power of composability of operators is advantageous towards the design of a simple yet powerful API.

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, table 3.1 presents the most important categories[4] and the associated operators:

3.4.3 Formalizing Operators

So far we have seen which are the most useful operators and how they aid in making our reactive library more useful from a user perspective. Things get more complicated when we try to analyze them from a theoretical point of view: operators can be viewed as state machines, containing an internal state which is modified whenever an event occurs, following the semantics of the specific operator.

For this reason, trying to formalize them as a functional and pure structure becomes very difficult, resulting in more confusion than clarity. This outcome should not come as a surprise: operators

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

Table 3.1: The essential operators

Transformation	buffer bufferWithSkip groupBy fmap scan scanLeft sample throttle window
Filtering & Conditional	filter distinctUntilChanged skip skipUntil skipWhile take takeUntil untilTake
Combining	(»=) (<*>) concat startsWith withLatestFrom zip zipWithBuffer
Error Handling	catchOnError onErrorResumeNext retry
Utility	observeOn subscribeOn publish share ofType doOnNext doOnError doOnCompleted toIterable toList

defined on other more common data structures such as lists or trees are state machines as well, usually hiding their state using function parameters:

```
take :: Int -> [a] -> [a]
take _ []      = []
take 0 _       = []
take n (x:xs) = x : take (n-1) xs
```

As we can see from this example implementation of take on lists, the internal state of the operator - i.e. the number of elements to be taken, `n` - is wired in the definition of the function, eliminating the need for an internal variable as a result.

This type of operator definition works very neatly for any pull-based data structure, where we can define the operators recursively on the structure of the collection. For `Observable` things become a little more complicated since we are dealing with a push-based collection, were elements are never gathered as a concrete collection in memory, not allowing, as a consequence, any type of structural recursion.

We will discuss in section 3.4.3, Future Work, how Event Calculus might be used as a technique to better define the semantics of operators for push-based collections.

Conclusions

> *"... and the mystery clears gradually away as each new discovery furnishes a step which leads on to the complete truth."*
>
> — Sir Arthur Conan Doyle, *Sherlock Holmes - The Adventure of the Engineer's Thumb*

The main research goal of this work was to analyze and formalize what is commonly referred to as the reactive programming paradigm by means of a theoretical and mathematical approach.

We broke down our approach into three research questions, which were answered throughout the discussion presented in this report.

- **Which class of problems does reactive programming solve? How does this relate to the real world libraries that claim to be reactive?**

  Starting from Berry's definition of reactive programs [7], we identified the class of problems reactive programming sets out to solve as those dealing with asynchronous, event based data sources. After showing how such problems require a push based model of computation in order to be solved, we analyzed the most famous libraries and APIs that claim to embody the reactive philosophy and showed how, more often than not, this claim is not true from a theoretical perspective.

- **How can we use existing mathematical and computer science theory in order to formally derive a definition for reactive programming?**

  Starting from the intuition that the definitions and properties of interactive and reactive programs are the opposite of one another, we used the categorical concept of duality, as well as other useful constructs borrowed from mathematics - see Appendix A -, in order to simplify the definition of `Iterable` to its essential type and use this to formally derive the `Observable` type, thus proving our intuition correct. We later proved the connection between the previously mentioned definition of reactive programming and the `Observable` by showing its correspondence with the definition of a special kind of a Continuation Monad, where the result type is `IO ()` and the side effects of its inner workings are made explicit in the type itself.

- **How can we bridge from the derived theoretical foundations of reactive programming to a concrete API that, whilst maintaining its mathematical roots, is fit for use in a production environment?**

  The last part of this research focused on building a reference implementation of a reactive library starting from the derived theoretical definition of `Observable`. In this section of the work, we augmented the `Observable` with features - subscriptions, schedulers, operators - that would make the type both useful and usable in a production environment, effectively resulting in a reactive library. We analyze each of the proposed additional features under both a theoretical - their meaning and impact on the previously derived formal types - and practical - implementation details and related challenges - point of view, with the purpose of stimulating awareness and discussion w.r.t. these features and their related challenges, rather than being prescriptive and forcing a specific solution upon the reader.

To conclude, this research contributes to the field of reactive programming by providing a formal derivation and analysis of the reactive types, a theory-biased implementation of these formal concepts and a production ready reactive library meant as a reference for software engineers interested in implementing a version of the library in their language of choice.

Limitations & Future Work

The work presented in this report does not come without limitations. The main one can be pinpointed to the development of our reference implementation. If the first section of our research is made precise by the use of mathematics and category theory, the bridging between theory and practice, realized by augmenting the reactive interfaces with additional features, cannot be justified in a scientific fashion. Whether a certain feature might or might not result useful for a software engineer in a production environment is not easily quantifiable. In this work, we relied on both common sense and the fact that the introduced features can already be found in wildly used reactive libraries such as the Reactive Extensions family. From this point of view, we have contributed to better understanding these features by providing a theoretical analysis, as well as a discussion on the practical advantages and challenges that would follow from their inclusion in a production reactive API.

Another limitation is represented by the Reactive Contract. Once again, its introduction is justified by wildly spread acceptance and commons sense, yet these reasons are not strong enough to preclude from the formulation of a different contract that would eventually result in an API with different semantics from the one developed in this report.

These limitations sparkle motivation for further research: with regards to the introduction of new features such as subscriptions, schedulers and operators, additional work could focus on better defining such concepts from a theoretical perspective, finding an abstract model to represent their behavior and semantics, which could more easily act as a reference for implementors. To this end, Event Calculus [39] seems to be a promising mathematical language to reason about events and their effects, making it interesting for modeling the behavior of operators.

Moreover, further research could investigate a different set of rules that would constitute the Reactive Contract, analyzing the ways these could affect the resulting reactive library and the use cases where one set could be more useful than another. As an example, we could imagine a set of rules which lifts the constraint that a stream must terminate after an error is produced. This contract could be useful for certain types of applications where errors in the processing of a single element are tolerated.

We hope, with this work, to have sparkled interest towards the field of reactive programming, removing any doubt as to what its definition and properties are, and giving it its right place within the computer science's scientific community.

APPENDIX A

APPENDIX A

A.1 Duality - Category Theory

Duality is an element of paramount importance in research we presented. Up until this moment we have only dealt with an informal definition of duality but its roots go very deep in the field of mathematics and category theory in particular.

Given the formal definition of a category, composed of objects

$$A, B, C, \ldots$$

and arrows

$$f, g, h, \ldots$$

and operations

$$dom(f) \quad cod(f) \quad id(A) \quad g \circ f$$

Given any sentence Σ, we can create its dual Σ^{op} by interchanging

$$dom \rightarrow cod$$

and the order of composition

$$g \circ f \rightarrow f \circ g$$

It is clear how Σ^{op} is a well formed sentence in the language of category theory CT.

The principle of - formal - duality then tells us that for any sentence Σ in the language of category theory, if Σ follows from the axioms defined for categories, then so does Σ^{op}.

$$CT \vdash \Sigma \implies CT \vdash \Sigma^{op}$$

From a visual perspective, this definition boils down to reversing the order of composition of the arrows.

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[dr, "g \circ f"'] & B \arrow[d, "g"] \\
& C
\end{tikzcd}
\qquad
\begin{tikzcd}
A & B \arrow[l, "f"'] \\
& C \arrow[u, "g"'] \arrow[ul, "f \circ g"]
\end{tikzcd}$$

A.2 Products & Coproducts

In category theory, the product of two objects in a category is the most general object which admits a morphism to each of the ones that compose it. The notion of product aims at capturing the essence of more specialized definitions found in various categories and areas of mathematics. The easiest way understand this construct is to start from the cartesian products in the category of sets.

Given sets A and B, let us define the cartesian product as the set

$$A \times B \ = \ \{\, (a, b) \mid a \in A,\ b \in B \,\}$$

There are two coordinate projections

$$A \xleftarrow{\ fst\ } A \times B \xrightarrow{\ snd\ } B$$

where

$$fst\,(a, b) = a \qquad snd\,(a, b) = b$$

It follows that given any element $c \in A \times B$

$$c \ = \ (\, fst\,(c),\ snd\,(c)\,)$$

The following diagram captures the essence of cartesian products.

$$\begin{tikzcd}
 & 1 \arrow[dl, "a"'] \arrow[d, "{(a,b)}"] \arrow[dr, "b"] & \\
A & A \times B \arrow[l, "fst"] \arrow[r, "snd"'] & B
\end{tikzcd}$$

The definition of categorical products can be derived from generalizing the elements in the previous definition.

Coproducts are the dual notion of categorical products, representing the least general object to which the objects in the family admit a morphism. Within the context of set theory, the represent the disjoint union of sets.

Given sets A and B, let us define their disjoint union as the set

$$A + B \; = \; \{\, (a,1) \mid a \in A\} \cup (b,2) \mid b \in B\}$$

There are two injection functions

$$\begin{tikzcd}
A \arrow[r, "left"] & A + B & B \arrow[l, "right"']
\end{tikzcd}$$

where

$$left\,(a) = (a,\ 1) \qquad right\,(b) = (b,\ 2)$$

The essence is captured by the following diagram, where we have simply reversed the arrows.

$$\begin{tikzcd}
 & 1 & \\
A \arrow[ur, "f"] \arrow[r, "left"'] & A + B \arrow[u, "{[f,g]}"'] & B \arrow[ul, "g"'] \arrow[l, "right"]
\end{tikzcd}$$

where

$$[f,g](x,\delta) = \begin{cases} f(x), & \delta = 1 \\ g(x), & \delta = 2 \end{cases}$$

Withing the context of our research and the Haskell programming language, tuples represent products and the Either monad represents coproducts.

A.3 Functors

In category theory a functor is a mapping from one category to another or a homomorphism of categories where certain laws hold.

Functors span thorough a large number of categories and are more and more common in modern programming languages. In Haskell the Functor class and its laws are defined as follows.

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

fmap id      = id
fmap (p . q) = (fmap p) . (fmap q)
```

A.4 Covariance & Contravariance

A.5 Continuations

A.6 Curry & Uncurry

Currying is the technique of transforming a function taking multiple arguments as input in one taking only the first argument and returning a function that takes the remainder of the arguments and returns the result of the initial function. Uncurrying is the opposite of currying, taking a curried function into one that accepts multiple arguments as input.

```
curry   :: ((a, b) -> c) -> (a -> b -> c)
uncurry :: (a -> b -> c) -> (a, b) -> c

f :: a -> b -> c
g :: (a, b) -> c

f = curry g
```

```
g = uncurry f

-- f x y = g (x, y)
-- curry . uncurry = id
```

APPENDIX B

APPENDIX B

B.1 Formal Rx Implementation

The following code presents a continuation based reference Reactive Library where each additional feature is implemented as discussed in the respective formalization section in Chapter 3 .

```
{-# LANGUAGE ScopedTypeVariables #-}

module Main where

import System.IO
import Control.Arrow
import Control.Monad
import Control.Monad.Cont
import Control.Concurrent.MVar
import Control.Concurrent.STM
import Control.Monad.Reader
import Control.Exception
import Data.IORef
import Data.List
import Data.Maybe
import Control.Concurrent (ThreadId, forkIO, forkOS, killThread, yield, myThreadId)
import Control.Concurrent.STM.TChan
```

```


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

data Event a =
      OnNext a
    | OnError SomeException
    | OnCompleted

subscribe :: Observable a -> Observer a -> IO Subscription
subscribe obs obr = do
    s <- emptySubscription
    let
        safeObr = observer safeOnNext safeOnError safeOnCompleted
        safeOnNext a = do
            b <- isUnsubscribed s
            unless b $ obr (OnNext a)
        safeOnError e = do
            b <- isUnsubscribed s
            unless b $ finally (obr $ OnError e) (unsubscribe s)
        safeOnCompleted = do
            b <- isUnsubscribed s
            unless b $ obr OnCompleted >> unsubscribe s
    runContT (runReaderT obs s) safeObr
    return s

observable :: (Observer a -> IO ()) -> Observable a
observable os = do
    s <- ask
    lift . ContT $ \downstream ->
        let
            -- subscription check not necessary but useful
            on a = do
                b <- isUnsubscribed s
                unless b $ handle oe (downstream $ OnNext a)
            oe e = do
                b <- isUnsubscribed s
```

```
                unless b $ downstream (OnError e)
            oc = do
                b <- isUnsubscribed s
                unless b $ handle oe (downstream OnCompleted)
        in
            os $ observer on oe oc

observer :: (a -> IO ()) -> (SomeException -> IO ()) -> IO () -> Observer a
observer on oe oc ev = case ev of
    OnNext v     -> on v
    OnError e    -> oe e
    OnCompleted -> oc

rxmap :: Observable a -> (a -> b) -> Observable b
rxmap o f = o >>= mapCont
    where
        mapCont ev = observable $ \downstream -> case ev of
            OnNext v     -> downstream (OnNext (f v))
            OnError e    -> downstream (OnError e)
            OnCompleted -> downstream OnCompleted

rxtake :: Observable a -> Int -> Observable a
rxtake o n = do
    nRef <- liftIO $ newIORef n
    o >>= takeFunc nRef
    where
        takeFunc nRef ev = observable $ \downstream -> case ev of
            OnNext v     -> do
                n' <- atomicModifyIORef nRef $ pred &&& pred
                when (n' >= 0) $ downstream (OnNext v)
                when (n' == 0) $ downstream OnCompleted
            OnError e    -> downstream (OnError e)
            OnCompleted -> downstream OnCompleted

rxflatmap :: Observable a -> (a -> Observable b) -> Observable b
rxflatmap o f = do
    s <- ask
    gate    <- liftIO $ newMVar ()
```

```
    active <- liftIO $ newTVarIO (0 :: Int)
    err    <- liftIO $ newTVarIO False
    compl  <- liftIO $ newTVarIO False
    let
        flatmapCont ev = observable $ \downstream ->
            let
                onNext v = do
                    atomically $ modifyTVar active (+1)
                    s_ <- emptySubscription
                    addSubscription s s_
                    let
                        inner        = observer onNext_ onError onCompleted_
                        onNext_ v_   = withMVar gate $ \_ -> downstream (OnNext v_)
                        onCompleted_ = do
                            cond <- atomically $ do
                                c <- readTVar compl
                                modifyTVar active (subtract 1)
                                a <- readTVar active
                                return (c && a == 0)
                            if cond
                                then downstream OnCompleted
                                else removeSubscription s s_
                    handle onError $ runContT (runReaderT (f v) s_) inner
                onError e = do
                    cond <- atomically $ do
                        e <- swapTVar err True
                        return (not e)
                    when cond $ downstream (OnError e)
                onCompleted = do
                    cond <- atomically $ do
                        c <- swapTVar compl True
                        a <- readTVar active
                        return (not c && a == 0)
                    when cond $ downstream OnCompleted
            in case ev of
                OnNext v    -> onNext v
                OnError e   -> onError e
                OnCompleted -> onCompleted
```

```haskell
    o >>= flatmapCont


--------------------------------------------------------------------------------
-- Subscriptions
--------------------------------------------------------------------------------

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

instance Eq Subscription where
    s == t = subscriptions s == subscriptions t

createSubscription :: IO () -> IO Subscription
createSubscription a = do
    b  <- newIORef False
    ss <- newIORef []
    return $ Subscription a b ss

emptySubscription :: IO Subscription
emptySubscription = createSubscription (return ())

isUnsubscribed :: Subscription -> IO Bool
isUnsubscribed s = readIORef $ isUnsubscribed_ s

unsubscribe :: Subscription -> IO ()
unsubscribe s = do
    b <- isUnsubscribed s
    unless b $ do
        writeIORef (isUnsubscribed_ s) True
        onUnsubscribe s
        subs <- readIORef $ subscriptions s
        mapM_ unsubscribe subs

addSubscription :: Subscription -> Subscription -> IO ()
```

```haskell
addSubscription s s' = modifyIORef' (subscriptions s) $ \ss -> s':ss

removeSubscription :: Subscription -> Subscription -> IO ()
removeSubscription s s' = modifyIORef' (subscriptions s) $ \ss -> delete s' ss


--------------------------------------------------------------------------------
-- Schedulers
--------------------------------------------------------------------------------

type Scheduler = IO Worker
data Worker = Worker
    { _schedule      :: IO () -> IO Subscription
    , _subscription  :: Subscription
    }

newThread :: Scheduler
newThread = do
    reqChan <- newTChanIO
    tid <- forkIO $ forever $ do
        join $ atomically $ readTChan reqChan
        yield
    subscription <- createSubscription $ killThread tid

    return Worker
        { _schedule = \action -> do
            atomically (writeTChan reqChan action)
            emptySubscription
        , _subscription = subscription
        }

observeOn :: Observable a -> Scheduler -> Observable a
observeOn o sched = do
    s <- ask
    w <- liftIO sched
    liftIO $ addSubscription s (_subscription w)

    let observeOnCont ev = observable $ \downstream -> case ev of
```

```
            OnNext v    -> void . _schedule w $ downstream (OnNext v)
            OnError e   -> void . _schedule w $ downstream (OnError e)
            OnCompleted -> void . _schedule w $ downstream OnCompleted

    o >>= observeOnCont
```

Appendix C

C.1 Proving Functor laws for Iterable

The following snippet presents a proof of the `Functor` laws for the `Iterator` and `Iterable` type.

```
-- Using a type synonym instead 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)
            -- eta abstraction
        = \ia -> ((fmap p) . (fmap q)) ia
            -- definition of (.)*
        = \ia -> fmap p (fmap q ia)
            -- definition of fmap q
        = \ia -> fmap p (\() -> ia () >>= return . q)
            -- definition of fmap p
        = \ia -> \() -> (\() -> ia () >>= return . q) () >>= return . p
            -- eta reduction inner lambda
        = \ia -> \() -> ia () >>= return . q >>= return . p
            -- eta abstraction
        = \ia -> \() -> ia () >>= \a -> (return . q) a >>= return . p
            -- definition of (.)
        = \ia -> \() -> ia () >>= \a -> return (q a) >>= return . p
            -- IO monad left identity*
        = \ia -> \() -> ia () >>= \a -> (return . p) (q a)
            -- definition of (.)
        = \ia -> \() -> ia () >>= \a -> (return . p . q) a
            -- definition of (.)
        = \ia -> \() -> ia () >>= \a -> return ((p . q) a)
            -- definition of fmap
        = \ia -> fmap (p . q) ia
            -- eta reduction
        = 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)

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

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)
        -- eta abstraction
    = \iia -> ((fmap p) . (fmap q)) iia
        -- definition of (.)
    = \iia -> fmap p (fmap q iia)
        -- definition of fmap
    = \iia -> fmap p (\() -> iia () >>= return . fmap q)
        -- definition of fmap
```

```
        = \iia -> \() -> (\() -> iia () >>= return . fmap q) () >>= return . fmap p
            -- eta reduction
        = \iia -> \() -> iia () >>= return . fmap q >>= return . fmap p
            -- eta abstraction
        = \iia -> \() -> iia () >>= \ia -> (return . fmap q) ia >>= return . fmap p
            -- definition of (.)
        = \iia -> \() -> iia () >>= \ia -> return (fmap q ia) >>= return . fmap p
            -- IO monad left identity
        = \iia -> \() -> iia () >>= \ia -> (return . fmap p) (fmap q ia)
            -- definition of (.)
        = \iia -> \() -> iia () >>= \ia -> (return . fmap p . fmap q) ia
            -- eta reduction
        = \iia -> \() -> iia () >>= return . fmap p . fmap q
            -- Iterator composition law
        = \iia -> \() -> iia () >>= return . fmap (p . q)
            -- definiton of fmap
        = \iia -> fmap (p . q) iia
            -- eta reduction
        = fmap (p . q)
```

C.2 Proving Contravariant laws for Observable

The following snippet presents a proof of the `Contravariant` and `Functor` laws for the `Observer` and `Observable` type respectively.

```
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
```

```haskell

-- 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)

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

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
```

BIBLIOGRAPHY

[1] *Reactive Streams.*
`http://www.reactive-streams.org/`.

[2] *ReactiveX, an API for asynchronous programming with observable streams.*
`http://reactivex.io/`.

[3] *Rx (Reactive Extensions) on Codeplex.*
`https://rx.codeplex.com/`.

[4] *RxJava - Wiki.*
`https://github.com/ReactiveX/RxJava/wiki`.

[5] *Rx Design Guidelines.*
`http://go.microsoft.com/fwlink/?LinkID=205219`, October 2010.

[6] 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.

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

[8] J. A. BERGSTRA AND J. W. KLOP, *Process algebra for synchronous communication*, Information and control, 60 (1984), pp. 109–137.

[9] P. CAPRIOTTI, *Continuation-based relative-time FRP*.
`https://www.paolocapriotti.com/blog/2012/06/04/continuation-based-relative-time-frp/`, June 2012.

[10] K. CWALINA AND B. ABRAMS, *Framework design guidelines: conventions, idioms, and patterns for reusable. net libraries*, Pearson Education, 2008.

[11] 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.

[12] C. ELLIOTT, *Denotational design with type class morphisms (extended version)*, tech. rep., Technical Report 2009-01, LambdaPix, March 2009. URL http://conal. net/papers/type-class-morphisms.

[13] ——, *Denotational Design*.
Lambda Jam 2015, `https://www.youtube.com/watch?v=bmKYiUOEo2A`, October 2015.

[14] ——, *The Essence of FRP*.
`https://begriffs.com/posts/2015-07-22-essence-of-frp.html`, July 2015.

[15] C. ELLIOTT AND P. HUDAK, *Functional reactive animation*, in ACM SIGPLAN Notices, vol. 32, ACM, 1997, pp. 263–273.

[16] 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.

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

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

[19] J. GIBBONS AND B. C. D. S. OLIVEIRA, *The essence of the iterator pattern*, Journal of Functional Programming, 19 (2009), pp. 377–402.

[20] G. GONZALES, *Pipes*.
`https://github.com/Gabriel439/Haskell-Pipes-Library`, 2012.

[21] T. GOWERS, J. BARROW-GREEN, AND I. LEADER, *The Princeton companion to mathematics*, Princeton University Press, 2010.

[22] C. H. GROUP, *Covariance, contravariance, and positive and negative position*.
`https://www.schoolofhaskell.com/user/commercial/content/covariance-contravariance`.

[23] HASKELL.ORG, *HaskellWiki*.
`https://wiki.haskell.org`.

[24] ——, *Lifting*.
`https://wiki.haskell.org/Lifting`.

[25] R. K. JONAS BONER, DAVE FARLEY AND M. THOMPSON, *Reactive Manifesto*.
`http://www.reactivemanifesto.org/`, 2014.

[26] V. K. JONAS BONER, *Reactive programming vs. Reactive systems.* https://www.oreilly.com/ideas/reactive-programming-vs-reactive-systems, 2016.

[27] O. KISELYOV, *Iteratees*, Functional and Logic Programming, (2012), pp. 166–181.

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

[29] E. MEIJER, *Subject/Observer is Dual to Iterator*, in FIT: Fun Ideas and Thoughts at the Conference on Programming Language Design and Implementation, 2010.

[30] ——, *Your mouse is a database*, Queue, 10 (2012), p. 20.

[31] ——, *Duality and the End of Reactive.* Lang.NEXT 2014, https://channel9.msdn.com/events/Lang-NEXT/Lang-NEXT-2014/Keynote-Duality, May 2014.

[32] ——, *Let Me Calculate That For You.* Lambda Jam 2014, https://www.youtube.com/watch?v=pOl4E8x3fmw, September 2014.

[33] E. MEIJER AND B. BECKMAN, *Expert to Expert: Brian Beckman and Erik Meijer - Inside the .NET Reactive Framework (Rx).* https://channel9.msdn.com/Shows/Going+Deep/Expert-to-Expert-Brian-Beckman-and-Erik-Meijer-I July 2009.

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

[35] J. NEWBERN, *All About Monads.* https://www.cs.rit.edu/~swm/cs561/All_About_Monads.pdf.

[36] ORTHOGONALITY, *Orthogonality — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/wiki/Orthogonality#Computer_science, 2016.

[37] SCOPUS.COM, *Scopus Search: Reactive Programming 1960-2016.*

[38] D. SCOTT, *Outline of a mathematical theory of computation*, Oxford University Computing Laboratory, Programming Research Group, 1970.

[39] M. SHANAHAN, *The event calculus explained*, in Artificial intelligence today, Springer, 1999, pp. 409–430.

[40] TEL, *What's up with contravariant?* https://www.reddit.com/r/haskell/comments/1vc0mp/whats_up_with_contravariant/.