UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA UNIVERSIDAD DE MÁLAGA

Graduado en Ingeniería del Software

Diseño de Sistemas Funcionales y Reactivos

Functional and Reactive Systems design

Realizado por
Santiago Sánchez Fernández

Tutorizado por
José Enrique Gallardo Ruiz

Departamento
Lenguajes y Ciencias de la Computación

MÁLAGA, junio de 2020

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADUADO EN INGENIERÍA DEL SOFTWARE

Diseño de Sistemas Funcionales y Reactivos

Functional and Reactive Systems design

Realizado por
Santiago Sánchez Fernández

Tutorizado por
José Enrique Gallardo Ruíz

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO DE 2020

Fecha defensa: julio de 2020

Abstract

As today's user demands faster changes on software systems, a programming foundation based on easy to change programs is needed. Functional Programming constructs are highly composable and free of side effects. For this reason, this paper studies it as a paradigm for creating programs that can adapt to new requirements in a fast manner. With application responsiveness being to an increasingly important trait of user-facing applications, Reactive Systems are studied as a frame of reference for creating applications that respond fast to user interactions and are fault-tolerant within the functional scope.

Functional and Reactive systems concepts are not only presented on a theoretical basis but are also accompanied by examples and common patterns that make the most profit from them.

Finally, a functional and reactive web application that makes use of all these concepts is developed putting in practice all the introduced concepts and showcasing a practical application that uses persistent storage and network communication in a purely functional manner using the reactive principles.

Keywords: Functional Programming, Scala, Reactive Systems

Resumen

Dado que los usuarios hoy en día requieren actualizaciones constantes del software que usan, son necesarios unos fundamentos de programación que permitan realizar cambios frecuentes en estos sistemas de una manera lo más simple posible. Los elementos propios de la Programación Funcional son fáciles de componer y están libres de efectos colaterales. Es por ello que este documento estudia la programación funcional como un paradigma para crear programas que pueden adaptarse con facilidad a los cambios que implican la evolución de requisitos. Al mismo tiempo, la capacidad del software de responder rápidamente a las interacciones con los usuarios es cada vez más importante. Los Sistemas Reactivos establecen un marco de referencia para crear aplicaciones que sean capaces de interaccionar con un número elevado de usuarios con una baja latencia y que sean además tolerantes a fallos.

Estos conceptos no son presentados exclusivamente de un punto de vista teórico sino que también se acompañan de ejemplos prácticos y patrones útiles para crear Sistemas Funcionales y Reactivos.

Finalmente, se estudia un caso de uso práctico en el que se utilizan todos los elementos que se han estudiado, implementando una aplicación que utiliza almacenamiento persistente y comunicación a través del protocolo HTTP de una forma puramente funcional, siguiendo además los principios propios de la programación reactiva.

Palabras Clave: Programación Funcional, Scala, Sistemas Reactivos

Contents

1 Introduction

1.1 Motivation

Software systems are demanding nowadays an unprecedented amount of changing requirements. As more people have gained access to personal computers, and more lately mobile devices, the amount of users an application must serve has increased dramatically. In the past, software was developed using a waterfall model. Target users were localized and systems were made to satisfy their needs with their requirements being known a priori.

This scenario is no longer true. Today's users expect software that adapts to their needs and in many cases have plenty of alternatives to choose from. If an application is not able to accommodate to the user demands, they can browse or download another one in matter of seconds to minutes.

With that premise in mind, it is now required more than ever to construct applications that are evolutionary, easy to change and mantain. This is exactly what Functional Programming aims to achieve. Functional Programs are composed of functions that are highly composable and that require low maintenance. These functions are guaranteed to maintain their behavior regardless of external changes. Additionally, as they are free of side effects, they provide a seamless reutilization experience given that they produce no observable result other than producing a result, so that they are safe to use regardless of the context in which they are called.

Due to these characteristics, refactoring is safe because one change in a function cannot produce a side effect that could affect other functions which are not using it.

As users demand new features more frequently, additionally expecting no degradation of already existing behaviours, changes are expected to occur on code often and having such properties of functional programs is very desirable in this context.

At the same time, the non functional requirements of software systems have been growing to be more strict. Nowadays not only human beings consume applications, the so called Internet of Things has been growing massively, starting to be ubiquitously present. From home automation to agriculture there are devices which connect to the internet to do their work. This can make services to be consumed by millions of clients simultaneously.

When developing web applications, they must be able to scale to this circumstances and work under high loads without affecting the user experience and the application responsiveness. The Reactive System term have been coined to describe a series of traits and elements that form software applications which accomplish these characteristics.

1.2 Objectives

Firstly, in this document it is going to be exposed what Functional Programming is, the characteristics of functional programs and the benefits that they provide to software systems. At the same time, it is going to be showcased in a practical way how to create functional programs in the Scala programming language[1] and why this is a suitable language for writting functional programs.

Secondly, it will be introduced what reactive systems are, as a principled way to structure applications that are able to scale and be responsive for users and how to create them in the context of Functional Programming, combining the benefits of systems that are able to change and have good performance.

The end goal will be to create a basic set of principles for Functional Programming which will allow the programmer not only to recognize what makes programs functional but also to get the most benefits from them. At the same time common patterns and elements of functional programs will be introduced to help to earn the greatest profit from this paradigm.

Finally, a simple web application built with purely functional constructs will be developed. This will showcase how Functional Programming can be used to create applications that interact with the real world and that, at the same time, have the reactive traits, thus being both easily adaptable to change and good performant.

1.3 Structure of the document

The structure of the document is aligned with the objectives of this work. Chapter 2 will introduce the Functional Programming paradigm along with its principles and benefits.

Chapter 3 will define the concept of Reactive Systems, how they relate to well performing applications and what makes a system reactive.

Both chapters will thus serve as a study of the state of the art in order to understand the pillars behind both topics to be able to base the rest of the document on these fundamentals.

In Chapter 4, after these fundamentals are known, elements common to both reactive and functional programs are going to be introduced. Some common structures in Functional Programming will be presented as long as elements of programs that enable reactiveness for an application.

With the knowledge acquired in previous chapters, Chapter 5 will explore the companion application to this document. I will be a simple web application based on both functional and reactive traits. We will introduce a way of interacting with the external world in a functional manner and how an application can be structured based on purely functional constructs.

1.4 Technologies used

The whole text will use the Scala programming language for explaining concepts and to present examples. Ammonite [2] will be used as a tool for creating small Scala scripts as it allows importing third party dependencies and other scripts in a simple way.

Then the application which will put in practice the concepts exposed previously will be based on various libraries of the Typelevel ecosystem [3] which are based on the Cats library [4].

2 Concepts of Functional Programming

2.1 Introduction

Functional programming is a programming paradigm in which programs are structured as a composition of pure functions [5].

Pure functions, in contrast to function constructs of programming languages, refer to the mathematical concept of a function. In mathematics, "a function *f* from A to B, where A and B are non-empty sets, is a rule that associates with each element of A (the domain) a unique element of B (the codomain)" [6].

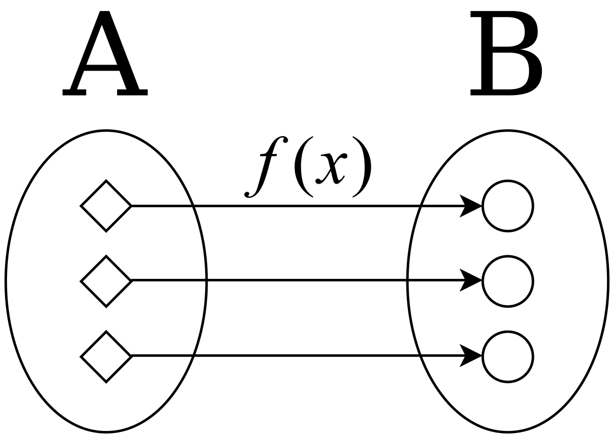


Figure 1: A visual representation of the function f

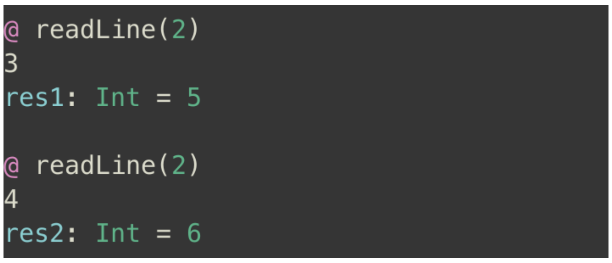
Although this is the mathematical definition, in the domain of programming languages, it could also be stated that "a pure function has no observable effect on the execution of the program other than to compute a result given its inputs" [7].

Function constructs in many programming languages don't have the traits of mathematical functions. One example of this is the one represented in Listing 1. This function reads from the console an integer and returns this read integer added to the parameter n.

```
def readLine(n: Int): Int = {
  val read = scala.io.StdIn.readInt()
  n + read
}
```

Listing 1: An example of an impure function

This function breaks the definition of a pure function. In figure 2, a Scala interactive session is ran with the previous function defined. In this session, it can be observed that for the same function input, `2`, two different outputs are returned, `res1: Int = 5` and `res2: Int = 6`, when the user introduces the inputs 3 and 4 respectively in the console, which is a violation of the definition of pure function given before.



```
@ readLine(2)
3
res1: Int = 5

@ readLine(2)
4
res2: Int = 6
```

Figure 2: An Scala interactive session showing how the `readLine` function is not pure

The elements which make programming language's functions not perform as mathematical functions are called side effects. To make a differentiation, functions without side effects are referred to as pure functions[8] while functions that have side effects are called procedures.

Some side effects that can make a function non-pure are [9]:

- Performing I/O
- Modifying a non-local variable
- Modifying a data structure in place

- Throwing an exception

2.2 Referential transparency

When treating with pure functions there is a one to one relation between a function call and the result it produces. For example, we can see that the expression $2 + 2$ is the same as 4. In mathematics, this is a very important property when solving equations. On the process of solving them, usually both sides of it are simplified by applying operations that reduce the number of elements on each side until we get a simple expression that its trivial to solve. In figure 3, this process is showcased.

$$2x - x = 2^2 + 2 \cdot 2$$
$$x = 4 + 4$$
$$x = 8$$

Figure 3: Solving a equation by simplifying thanks to referential transparency

This property is called referential transparency [10] and is a very important property of functional programming.

The reason why referential transparency is only possible when using pure functions lies in its definition. If a function is not pure, an element of the domain may be related with multiple elements of the codomain, thus, it is not posible to know a priori which is the related element to the input.

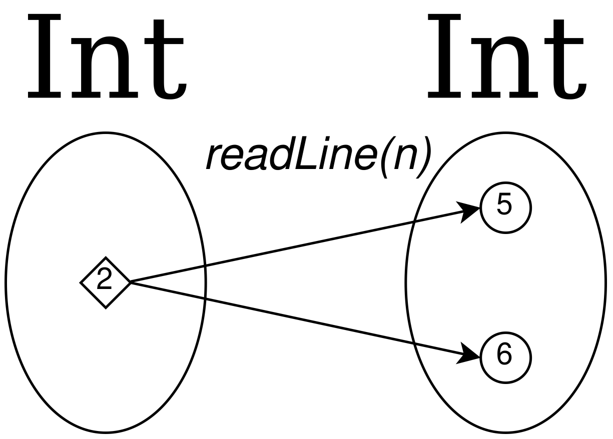


Figure 4: A representation of the relationship introduced by the readLine procedure

Figure 4 represents this indeterminism with the previously mentioned `readLine` procedure. Input 2 has two possible results, 5 and 6, and these are not the only possible results. There are as many possible results as `Int`s. Which one should be the one substituted? The given result will only be resolved at run time and thus referential transparency is lost.

2.2.1 Local reasoning

One of the benefits of referential transparency is local reasoning. In order to understand how a function behaves, it is only necesary to understand the components of the function themselves and not the context in which they are placed.

When referential transparency is not present, this is no longer possible. As an example, consider the function defined in Listing 2 which depends on a global variable `name`, and returns a greeting to that name.

```
var name = "Foo"

def greet() = s"Greeetings $name!"
```

Listing 2: A function which greets `name`

To understand the behaviour of the function, it is needed to know the context in which it is called, i.e. inspect where the variable `name` is assigned and also when assignments to the variable occur previous to the function call.

In the script shown in Listing 3, the output shown to console will be `Greeetings ooF!` as the name is assigned to *Foo* first and then is reversed.

But even though it is possible to analyse the behaviour at a certain point in time, it is not possible to ensure that, without changes to the source code of the function itself, the functionality will keep being the same. Changes in the context of the function call regarding the variable `name` may change the behaviour of the function in the future. In the script from listing 4, without changes to `greeting`, the result of the call is different. In addition to that, the side effect that the `greetReversed` call made has been completely overriden.

These examples are rather small and thus it is not that difficult to understand the context, but in an enterprise application where there are many more components, the solution does

```scala
import scala.util.chaining._

var name = "Foo"

def greet(): String = s"Greeetings $name!"
def greetReversed(): Unit = {
  name = name.reverse
}

greetReversed()
greet() tap println // Greeetings ooF!
```

Listing 3: An execution of greet which first reverses `name`

not scale that well. As the uses of the mutable state increase, the understanding of functions which use them start to be more complex.

2.2.2 The substitution model and equational reasoning

When referential transparency is given, a new reasoning model for functional programs can be achieved, called equational reasoning. When reasoning about a functional program, it is possible to start substituting function calls for the value they result in. To understand how a program behaves in term of its inputs, the only thing it is needed to do is to substitute function calls for results they produce.

This is possible because of the definition of a function. If a function f maps an element from the domain a to one of the codomain b then $f(a)$ can transparently be replaced by b and the same kind of substitution that were presented on the figure 3 can be applied to programs.

The substitution model can be useful in multiple scenarios. One example is the ability to debug errors in a reproducible manner.

The first step is to get the arguments of the misbehaving function. Once they have been acquired, the next step is to start substituting each function call for the value it returns. Whenever the substitution is made, it is checked that the result of it is the one expected. At the

```scala
import scala.util.chaining._

var name = "Foo"

def greet() = s"Greeetings $name!"
def greetReversed(): Unit = {
  name = name.reverse
}

def greetOther(whom: String): Unit = {
  name = whom
}

greetReversed()
greetOther("Bar")

greet() pipe println // Greeetings Bar!
```

Listing 4: An execution of greet which changes the `name` after reversing it

moment that a function results to a non expected result, the cause of the error is found.

As told before in the local reasoning section, this is not possible in a non functional program, as a function result is not self contained but it depends on the context in which it is called.

This debugging process can be done automatically by running a debugger which allows to evaluate expressions on the go. A more manual approach would be by the use of the Scala REPL (Read–eval–print loop). This tool allows to import certain functions of a program in a shell which can evaluate them with arbitrary parameters to help to identify the flaws of corresponding programs.

2.2.3 Function Composition and safe refactoring

The substitution model can also be used to safely refactor code when repetitions occur. There are two reasons why functional code can be safely refactored. The first is referential transparency. The second is because of function composition.

As defined at the beginning of this section, functional programs are defined by the composition of functions. If we have a function `f: A => B` and another function `g: B => C`, both can be composed by mathematical composition $g \circ f$ to create a new function `h: A => C`. At the beginning of the section, it was stated that this is indeed the essence of functional programming. Programs are the result of composed functions that are themselves, at the same time, the result of other functions or expressions composition.

Even programs which define variables which are then used by other functions are in the end functions defined by means of composition. This is a corollary of the substitution model. As variables in functional programming are no more than named results of expressions, every variable can be substituted by the expression which produced it. As this expressions can also be composed of other variables, this process has to be done recursively until there are no more variables.

At the very end of the substitutions the only thing that will be left is a pure expression.

Listing 5 showcases this behaviour. Here, `andThen` is forward function composition. A pure function like `greetCapitalized` is nothing more than the composition of previously defined functions.

```scala
val prependGreeting: String => String = name => s"Greetings
   $name!"
val capitalize: String => String = _.capitalize

val greetCapitalized: String => String = capitalize andThen
   prependGreeting
```

Listing 5: An example of how functional programming is about function composition

The same function can be defined by the use of intermediate variables instead of function composition. Listing 6 is the transformation of the previous example using intermediate variables. In the end, both functions are semantically equivalent, but the former does explicit function composition and the other is by defined through intermediate variables.

```
def greetCapitalized(name: String): String = {
  val capitalizedName = name.capitalize
  val prependGreeting = s"Greetings $capitalizedName"
  prependGreeting
}
```

Listing 6: An example of how functional programming is about function composition

Using this principle, it is possible to refactor a set of expressions used within a function to a new one and replace them transparently. This can be achieved without compromising the programs functionality as long as side effects are not present in them. If, during the program design process, repeated code is found, it can be factored into a function, and then the repeated code can be replaced by function calls.

2.3 Functional programming constructs and programming languages

The functional programming paradigm suits better in some programming languages than in others. This usually lies in the constructs a language provides to build programs.

2.3.1 Statements and expressions

In imperative languages, building blocks are usually conditional and loop statements. These constructs are inherently non functional as the way they behave is by having declared mutable variables that get updated in these statements. This is very clear in C style for and while loops.

2.3.2 Imperative statements

A while loop evaluates an expression. If this expression is true, the body block of the while loop is executed. In case it is false, the next statement to the block is executed.

The definition of the while loop spots the lack of referential transparency. It expects that an expression can produce two different values. Some times it will be false and if, we want the loop to end, at least once it will be true.

The for statement has a similar behaviour to the while statement. With the addition of a first statement that will be executed previous to the first loop iteration and a second one that will be executed at the end of each iteration.

Even if the expression used in the condition of the loops were referential transparent, the statements would add no value to the programs. A condition that evaluates to true would lead to an infinite loop which would never produce any value and, if it were false, the block would never get executed.

The if/else statement of imperative languages has the problem that it does not produce any value. It's code blocks have to perform a side effect, like modifying a variable, in order to do something useful.

There is one exception in which the if/else statement could produce a value while not breaking referential transparency. This is the case in which the return from a function is done inside the if block. As in Scala this imperative construct is not present, the Listing 7 presents this in a Java class `FunctionalIf` which showcases in the function `abs` how the if construct can be used in a referential transparent manner.

2.3.3 Functional expressions

Functional programming languages by contrast provide constructs that enable the definition of programs as the composition of referential transparent expressions. To provide conditional logic, Scala provides an if/else expression. In contrast to the if/else statement, the expression produces a value, which is the result of the expression of the block that get executed [1].

Another expression that Scala provides is the for expression. A for expression traverses a data structure in order to produce a new one by filtering and transforming their elements.

Along with if and for expressions, a way to iterate and construct functional programs is by using recursion. Recursion is an useful tool in order to loop through collections without breaking referential transparency. The problem with recursion is that each nested call makes the run time system allocate one extra frame in the execution stack to save the context in which the recursive call was made.

There is a way to overcome this, as functions in which the last expression calculated is the

[1]In case the if expression doesn't contain an else and the condition is not satisfied, the returned value is the only value in the `Unit` type [11]

```java
public class FunctionalIf {
    public static void main(String[] args) {
        int x = Integer.parseInt(args[0]);
        System.out.println(abs(x));
    }

    private static int abs(int x) {
        if (x < 0) {
            return -x;
        } else {
            return x;
        }
    }
}
```

Listing 7: An example of imperative if else which does not break referential transparency

recursive call, called tail-recursive funtions, can be efficiently implemented by an optimisation of the compiler which can substitute the recursive call for a `goto` statement without the expenses of new stack frame allocations [12] [2].

An example of a tail recursive function can be a one which calculates the sum of a list of integers. To make use of tail recursion, a auxiliar function is provided. It accepts, along with the list whose elememnts are yet to be added, an integer parameter (the accumulator), which accumulates the sum of all elements already added. As the addition of integers is a monoid with 0 as its identity, the initial call must be made with 0 as accumulator, as shown in Listing 8.

[2]In Scala, in order to tell the compiler to produce a compile time error if this optimization cannot be done, the `@tailrec` annotation can be used[13]

```scala
import scala.annotation.tailrec

def sum(xs: List[Int]): Int = {
  @tailrec
  def go(xs: List[Int], acc: Int): Int = xs match {
    case head :: tail => go(tail, acc + head)
    case Nil => acc
  }

  go(xs, 0)
}
```

Listing 8: A tail recursive function for summing a list of Integers

2.3.4 Higher order functions

Higher order functions refer to functions which accept functions as parameters. In Scala, functions have their own type, like `String`s or `Int`s do. This way they are treated as another regular type by the compiler, having the capacity of being passed around as any other value.

As an example of use, a string trimming function is shown in listing 9. This function will iterate, character by character, the string and will remove a character if a specific condition is not met. The predicate used to indentify remaining characters is decided by the caller. This predicate is a function `pred: Char => Boolean`.

2.4 Polymorphism in Functional Programming

2.4.1 Parametric polymorphism

One way in which functional programs can express polymorphism by the use of generic types in function declarations. This mechanism, available nowadays in many programming languages, allow functions to take parameters with an abstract type which will be resolved when the function is called. In Scala, to use a generic type in a function, the generic type must be defined between square brackets after the function name. This way, the function can be

```scala
import scala.annotation.tailrec

def trim(s: String, pred: Char => Boolean): String = {
  @tailrec
  def filter(chars: List[Char], acc: List[Char]): List[Char] =
   chars match {
    case head :: tail =>
      if (pred(head)) filter(tail, head :: acc)
      else filter(tail, acc)
    case Nil => acc
  }

  filter(
    s.toList,
    Nil
  ).reverse.mkString
}
```

Listing 9: Trimming a string based on a predicate

defined one time but be used with different types. The function filter defined previously for Listing 9 can be generalized for a generic list `List[A]` as shown in 10

2.4.2 Type classes

Although parametric polymorphism is the right tool when the details of the generic type are irrelevant, it lacks the possibility of defining a behaviour that is associated with the type itself.

Type classes [3] were defined in Haskell as a way to implement Ad-hoc polymorphism [14]. A type class is an abstract representation of a behaviour that types belonging to the class possess. If a type can implement the type class behaviour, it becomes an instance of the type class and it is said that the type belongs to the class.

[3]Not to be confused with classes in Object Oriented Programming

```scala
import scala.annotation.tailrec

def filter[A](as: List[A], pred: A => Boolean): List[A] = {
  @tailrec
  def go(as: List[A], acc: List[A]): List[A] = as match {
    case head :: tail =>
      if (pred(head)) go(tail, head :: acc)
      else go(tail, acc)

    case Nil => acc
  }

  go(as, Nil).reverse
}
```

Listing 10: Filtering the elements of a generic List

In Scala, implicit resolution was implemented as a way to have type classes resolved at compile time without the need of the function caller to provide the instance manually and have it resolved automatically by the compiler in an object oriented language [15].

To illustrate the example, a typeclass called `Show` is presented in Listing 11 with some instances. This typeclass is very illustrative because in Scala there is a `toString` method in the `Any` class[16] which every class must extend but it is very error prone as its default implementation returns the `hashCode` of the class, which may not be the wanted implementation. In case that the programmer forgets to override the method, the error can only be caught at run time. When using a type class instead, if a specific instance is not resolved at compile time, the program will fail to compile improving its overall resilience.

Typeclasses are a very useful form of polymorphism because they allow to decouple structure from behaviour. If a specific behaviour is desired for a given type, one does not need to modify the source code of the class; instead, a type class instance can be provided, even in a separate source file. This is specially useful when using third party libraries. If a new functionality is required for classes in compiled compiled libraries, corresponding source code cannot be modified. Nevertheless, a type class instance with the desired functionality can be provided to enrich the class capabilities.

There is one way to make the use of typeclasses less cumbersome and more natural with the use of Implicit Classes. This mechanism allows to wrap an existing class and add methods which use the instances of the given type class as if they had the originally defined type. Listing 12 shows how they can be used for our instances.

2.5 Functional data structures

At the beginning of the section, among the different kind of side effects we mentioned, was the case of *Modifying a data structure in place*. If modifying a data structure in place produces a side effect and breaks referential transparency, data structures should be defined in an immutable way by default and, as a corollary on this property, operations on them must not modify their internals but return a new copy of the structure with the changes required.

Languages that weren't designed to be functional ones usually define their data structures as mutable ones. In Java, the Collection class, root in the hierarchy of a great number of data structures, defines most of its operations as mutable ones [17]. As an example, the method `add` has the following definition `boolean add(E e)`. The returned type of the method is a boolean which, according to the documentation, "is true if this collection changed as a result of the call". This description implies that mutation has been done in place.

The main problem with this is that libraries that want to subclass the Java Collections API and be compatible with the standard library have to maintain these constraints. This has lead to alternative immutable collection implementations like Google's Guava Immutable Collection library [18] serving only as immutable representations of data structures but not providing a rich set of operations to use them.

Scala, for example, provides two type of collections. Immutable and mutable ones. Scala documentation states "Scala collections systematically distinguish between mutable and immutable collections. A mutable collection can be updated or extended in place. This means you can change, add, or remove elements of a collection as a side effect. Immutable collections, by contrast, never change. You have still operations that simulate additions, removals, or updates, but those operations will in each case return a new collection and leave the old collection unchanged” [19].

```scala
import scala.util.chaining._

trait Show[A] {
  def show(x: A): String
}

case class Person(name: String, age: Int)

implicit val showInt: Show[Int] = (x: Int) =>
  x.toString

implicit val showDouble: Show[Double] = (x: Double) =>
  x.toString

implicit val showPerson: Show[Person] = (x: Person) =>
  s"Person(${x.name}, ${x.age})"

// type classes can even be resolved recursively
implicit def showList[A](implicit ev: Show[A]): Show[List[A]] =
    (xs: List[A]) =>
  xs
    .map(ev.show)
    .mkString("List(", ", ", ")")

object Show {
  def show[A](a: A)(implicit evidence: Show[A]): String =
    evidence.show(a)
}

Show.show(List(1, 2, 3)) tap println // List(1, 2, 3)
```

Listing 11: Different instances of the Show typeclass

```scala
import scala.util.chaining._
import $file.Show, Show._

implicit class ShowOps[A](value: A) {
  def show(implicit evidence: Show[A]): String =
    evidence.show(value)
}

List(1,2,3).show tap println // List(1, 2, 3)
```

Listing 12: Adding syntax to the Show typeclass

3 Reactive Systems

3.1 The reactive manifesto

As mentioned in the introduction, software systems nowadays have to handle big amounts of data, coming from thousands of concurrent users, which are demanding low latency responses in the order of milliseconds and robust systems with a 100% of up time.

The reactive manifesto [20] calls Reactive Systems the ones able to cope with this expectances and gives this description: "Reactive Systems are more flexible, loosely-coupled and scalable. This makes them easier to develop and amenable to change. They are significantly more tolerant of failure and when failure does occur they meet it with elegance rather than disaster. Reactive Systems are highly responsive, giving users effective interactive feedback."

At the same time it describes whhich are the traits of reactive systems. These traits can be categorised hierarchically by organizing them towards the end goal they serve. Being responsive is the trait which directly provides the user value to systems, as systems that are responsive are more appealing for users. Resiliency and elasticity are the characteristics that have to be present in order to be responsive, and lastly, being message driven is the way through which systems can easily accomplish resiliency and elasticity. This relationship can be visualised in Figure 5.

3.2 The reactive principles

3.2.1 Responsive

Responsiveness is the cornerstone of the reactive systems. Systems which are responsive are more comfortable to use for users as they respond faster and adapt quicker to their needs.

Google found out in 2007 that additional 0.5 seconds of load time of a search could lead to a loss of interest of on the search of a 20%. [21].

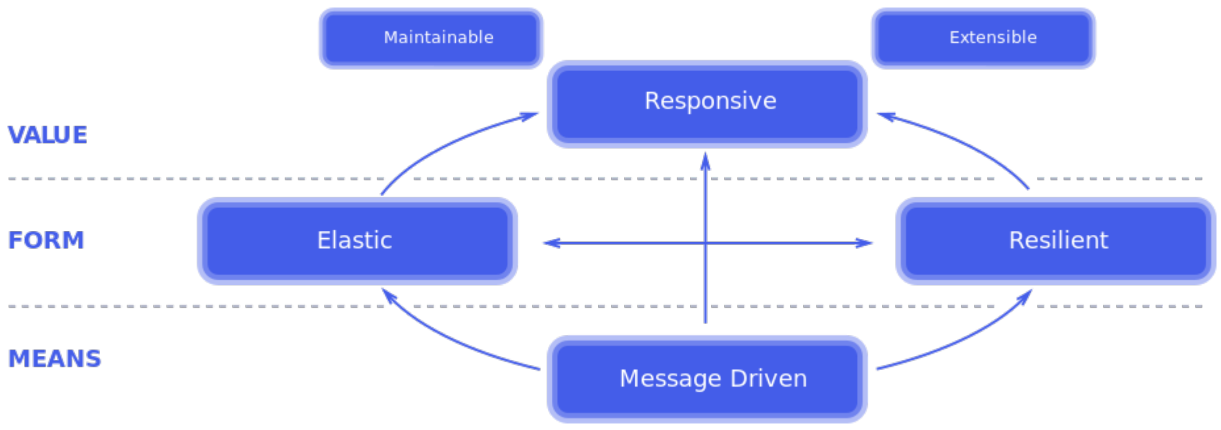


Figure 5: A hierarchical view of the reactive principles.

More recently, in an study developed by Akamai Technologies in 2017 [22], other insights about consequences of pages responsiveness were analyzed, being some of the most interesting the following ones:

- A 100-millisecond delay in website load time can hurt conversion rates by 7 percent[4].
- A two-second delay in web page load time increases bounce rates by 103 percent.
- 53 percent of mobile site visitors will leave a page that takes longer than three seconds to load.

When time is that relevant in consumer satisfaction and hundred of milliseconds makes the difference for a potential new user, it is important to design systems that respond to user requests with low latency in a consistent manner.

3.2.2 Resilient

Resilience provides a better responsiveness as systems react better to failure. A resilient system is able to operate even if parts of it are failing. As an example, if a user is using a social network, he should be able to see friends' posts even if the chat system is unavailable.

At the same time, resilience is the ability to recover from errors without manual intervention meaning that local errors shouldn't be propagated but rather handled and managed by different parts of the system.

[4]In the marketing context, conversion rates are the amount of users which when visiting a web page complete their desired goal, for example, buying an item when visiting an online commerce.

When designing reactive systems, errors should be expected to occur. Even if all the possible applications errors under control of the designers could be avoided, there are other elements of the application which are out of control, such as network errors or interactions with external systems. This principle is stated in the *Design for failure* approach [23] which embraces the acknowledge of possible errors, thus having to make systems which can have errors but can gracefully recover from them.

3.2.3 Elastic

Applications traffic isn't stable. Online commerces usually have much higher demands on holidays seasons like Christmas. A marketing campaign can create higher traffic than usual in applications. Even an unexpected reference in a newspaper or a website may increase the number of users to an unmanageable amount for the original design.

Load balancers or more advanced orchestration systems like Kubernetes [24] can handle elasticity on the infrastructure level, allowing replicas of the application to coexist. However, applications have to be designed to allow for the distribution of work among the instances of the application. This is referred as to Systems scalability.

Scalability The Scalability of a system is the capacity that it has to increase its throughput respectively to its hardware resources. It is defined by the Universal Law of Computational Scalability [25], the formula is a variation of the Amhdalh's law [26] and is present in figure 6

$$C(N) = \frac{N}{1+\alpha(N-1)+\beta N(N-1)}$$

Figure 6: The Universal Law of Computational Scalability formula.

In the formula, parameter N represents the amount of concurrent processes running the application, α is the time that is lost because of the need to wait for shared resources to become available and β is the time that distributed nodes take to have consistent data.

The mere act of increasing the parallelism of an application by adding more computation nodes doesn't mean that the application will scale linearly with it. Systems have to be designed in order to use efficiently available computing resources. In the Universal Law of Computational Scalability this is measured by α and β. If systems are not designed to be scalable both parameters will be very high and adding parallelism can even decrease performance.

3.2.4 Message driven

The way to achieve Elastic and Resilient systems is by the use of asynchronous message passing. Thanks to this mechanism application are decoupled both with respect to their space and and time constraints.

If modules of the application communicate by means of asynchronous messages, the end location of the resource which will handle the message is transparent for the caller. A message will be delivered to a mailbox. The specific receptor of the message is not known. A load balancer can choose which system will be the recipient of the message, allowing for **Elasticity**.

As messages are asynchronous, the sender of the message does not need to wait for a response from the receiver, and can handle other workloads or even terminate as is no longer responsible for the message.

This decoupling provides greater **Resilience** because the asynchronous boundary isolates errors from being propagated. Indeed, errors can be propagated as messages, having specific error handler modules which can act in consequence to avoid the collapse of the system and allow a gracefully recover from failure.

4 Elements of Functional and Reactive Systems

4.1 Abstracting over Computational Effects

In 1991 Eugenio Moggi [27] described a model in category theory for separating the set of values a object of type `A` may have from the notion of computing those values, `T A`, which denotes an effect on top of this type. This model abstracts from type `A` the possible results that the computation could have. Some examples of computations mentioned on that text of interest to this topics are:

- Side effects, that modify a set of possible states `S`: `State[A, S]`.
- Exceptions, where `E` is the set of possible exceptions: `Either[A, E]`.
- Interactive Input/Output, which results in a value of type `A`: `IO[A]`.

From a functional programming perspective, a very interesting finding of that work is that thess effects over type `A` can be abstracted over and we can define operations that transform the type A regardless of the effect in which it is contained, as long as the effect `F` is a monad.

4.1.1 Monads

A monad is category that provides two operations over a type constructor `F[_]`: `flatMap`, also usually called `bbind`, and `pure`, known as point or return. In Listing 13, a trait that defines the Monad typeclass in Scala is shown.

```scala
trait Monad[F[_]] {
  def unit[A](a: => A): F[A]

  def flatMap[A, B](ma: F[A])(f: A => F[B]): F[B]
}
```

Listing 13: The monad trait.

The semantics of the operations can be inferred from their types. The pure function takes a value `a` and lifts it into the monad context `F`. The flatMap operation takes a monad `F[a]` and a function, which transforms an `a` into a monad with the same context but a with possible different type inside, `F[B]`. This operation mantains the monad context.

4.2 Example of monadic effects

Some monad instances can be defined for representing different effects.

Option Monad The Option Monad is an effect which may represent a value of a type `A` or not. An `Option[A]` has two possible values. `Some(a)`, which means that the value is present, or `None`, meaning that there is no value.

The function flatMap in this case returns a transformed `Some` if the monad in which it was applied was also a `Some` or directly returns a `None` if the monad was empty and no transformation could be applied.

Either Monad When a function can either succeed or fail the Either Effect can be used to encode this possibility. Represented with type `Either[E, A]`, it has two possible disjoint values. Either are usually right biased, being the `A` or "right" value the one meaning the success and the `E` value the one meaning an error.

State Monad The State Monad is an effect which, given a state of type `S`, performs a state transition resulting in a value of type `A`, modeling an state machine. As state cannot be mutated in place, the state transition returns both the new state and the resulting value.

```scala
import $file.Monad, Monad._

implicit val optionMonadInstance: Monad[Option] = new Monad[
   Option] {
  def unit[A](a: => A): Option[A] = Some(a)

  override def flatMap[A, B](ma: Option[A])(f: A => Option[B]):
    Option[B] = ma match {
    case Some(value) => f(value)
    case _ => None
  }
}
```

Listing 14: The option monad instance.

When using the state monad, flatMap directly returns a monad with the new state, avoiding the need to explicitly pass the state all along the way, as shown in Listing 16.

IO Monad Side effects are inherently non functional. A side effect such as reading input from the keyboard implies that every time that this operation is called the outcome may be different, breaking referential transparency.

However, if we look to the description of operations that do side effects, they have nothing that implies this lose of referential transparency. For example, a function to read from keyboard whould have type `() => A` which is a perfectly valid type for a pure function.

If the execution of the function with the side effect is delayed, what it would end up being is just a description of it, without exposing its impure nature. This is what the IO monad does. When a function is passed to the IO the execution, is suspended and the description of such computation is captured. It is not until the programmer runs the descriptions wrapped in the IO Monad that the referential transparency is lost.

If this side effect execution is delayed until the very end of program execution, these operations can keep functional at its core and the benefits of purity are not lost until the interpretation of the side effects, which is usually done as the last step of the computation. Usually,

```scala
import $plugin.$ivy.`org.typelevel::kind-projector:0.10.3`
import $file.Monad, Monad._

implicit def eitherMonadInstance[Err]: Monad[Either[Err, *]] =
  new Monad[(Either[Err, *])] {
    override def unit[A](a: => A): Either[Err, A] = Right[Err,
   A](a)

    override def flatMap[A, B](ma: Either[Err, A])(f: A =>
   Either[Err, B]): Either[Err, B] = ma match {
      case Right(value) => f(value)
      case Left(err) => Left[Err, B](err)
    }
  }
```

Listing 15: The either monad instance.

in order to avoid running the execution of the side effects manually, the entry function of the application expects an IO to be returned which will be then be interpreted. This is exactly what the main method of Haskell does[28].

Alternatively, the IO datatype can be thought of as an state transformer over the state not belonging to the program itself but to the external world. When interpreted, this state is read through external ports, i.e. network card, peripherals, storage disk, ..., and, at the same time, the new produced state can be "written" to these external elements.

4.2.1 The resilience of Computational Effects

In statically typed programming languages, computational effects offer a greater resilience as the programmer has to explicitly deal with the effect and can not ignore the effectful nature of it, like having a possible error, an Either effect, or being a possibly future value, i. e. an IO effect.

This implicit treatment of effects, just like the use of a `null` pointer, has lead to great a number of mistakes, being called by Tony Hoare, one of the first designers of programming languages, his billion dolar mistake [29].

Nullability is not the only case of implicit effect in programming. The use of unchecked exceptions in object oriented programming languages can let errors get propagated without the programmer supervision.

Effects are then a pattern that improve the reactiveness of functional programs by improving their resilience thanks to the use of type system.

4.3 Leveraging Multiprocessing by means of Futures

Futures, also called Promises, provide an asynchronous computational model in which computations can be ran in a separate thead of execution, appart from the thread calling the future itself.

A Future can be created by using the `apply` method of the companion object of the trait `Future`. An invocation has this form `Future(expr)`. This would evaluate expr in the background. The call returns immediatly with a result of type `Future[A]`, being A the type of expression `expr`.

4.3.1 Making models reactive by the use of Futures

Computations that could block the thread of execution, e.g. Blocking I/O like fetching a resource through the network, should be ran inside Futures. This contributes to the overall application reactiveness by terms of responsivenes by improving the overall request latency and by reducing the load of the main execution thread.

Latency improvements of Futures By spawning blocking calls in a separate thread of execution, the response time of a request transitions from being the sum of the latencies of the independent blocking operations to the latency of the longest one. This is represented in figure 7.

4.3.2 Futures are highly composable

Futures are designed to be composable. Instead of transforming the result of the Future by waiting for the evaluation of the inner expression to be completed, some methods of a

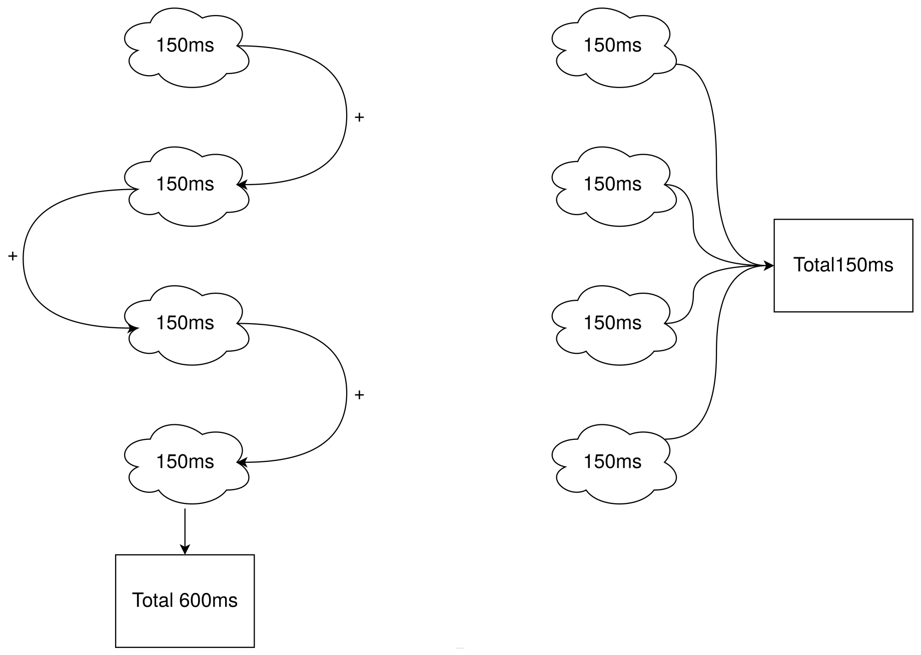


Figure 7: On the left of the figure, a sequential run of blocking operations. On the right, a parallel execution which combines operation's results.

`Future[A]` allow to transform it by passing a description of what should be done once the Future is completed. These methods don't block execution until the Future is completed but return inmediatly a `Future[B]`, being `B` the type of the value obtained by tranforming `A`.

Freeing the load of the main thread of execution In web servers there is usually a fixed thread pool for handling requests. This thread pool has a maximum number of threads available to prevent the system from collapsing. In Tomcat, for example, when this threshold is surpassed, no more requests can be met and requests start to be stacked [30]. As a consequence, latency starts to increase in a great manner.

Futures have an implicit parameter which defines the execution context in which the Future expression will be ran [31]. An execution context may have associated an specific thread pool. This can be used for making the blocking calls to be executed in a separate pool for blocking tasks, freeing the main thread thread pool which is only responsible of spawning the tasks for Futures, but not of running them.

4.4 The IO monad as a referential transparent Future

Futures have many properties that make them an appropiate construct for modeling asynchronous operations. However, their operations are not referentially transparent. By the definition of it, if Futures were referentially transparent the result of the operations on a Future should be the same regardless of the operations being composed directly or saved in variables.

This lack of referential transparency is shown in Listing 17. Two Future variables are created on the top level `futureByVariable` and `futureByComposition`. This Futures are created by composing two Futures created by the same operation, `random.nextInt`. However, in `futureByVariable` as `Future` caches the results of the function which was ran on it, the operation isn't performed the second time. This caching is indeed a side effect given that a variable with the result value has to be updated once the operation has been performed.

The IO Monad does not perform this caching and it is possible to save a value of type IO into a variable without losing referential transparency as Listing 18 shows.

The project Cats Effect [32] provides a standard `IO` type for Scala. This `IO` type provides both synchronous and asynchronous evaluation models. In addtion, the IO datatype provides an asynchronous processing model similar to Future but referentially transparent.

```scala
import $plugin.$ivy.`org.typelevel::kind-projector:0.10.3`
import $file.Monad, Monad._

case class State[S, +A](run: S => (A, S))

object State {
  def unit[S, A](a: A): State[S, A] =
    State(s => (a, s))
}

implicit def stateMonadInstance[S]: Monad[State[S, *]] =
  new Monad[(State[S, *])] {
    override def unit[A](a: => A): State[S, A] = State(s => (a,
    s))

    override def flatMap[A, B](ma: State[S, A])(f: A => State[S
   , B]): State[S, B] =
      State(s => {
        val (a, s1) = ma.run(s)
        f(a).run(s1)
      })
  }
```

Listing 16: The state monad instance.

```scala
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import scala.util.Random

val futureByVariable = {
  val random = new Random(0L)
  val x: Future[Int] = Future(random.nextInt)
  for {
    a: Int <- x
    b: Int <- x
  } yield (a, b)
}

val futureByComposition = {
  val random = new Random(0L)
  for {
    a: Int <- Future(random.nextInt)
    b: Int <- Future(random.nextInt)
  } yield (a, b)
}

futureByVariable.onComplete(println) // Success
   ((-1155484576,-1155484576))
futureByComposition.onComplete(println) // Success
   ((-1155484576,-723955400))
```
Listing 17: An example of how a Future is not referentially transparent.

```scala
import $ivy.`org.typelevel::cats-effect:2.1.3`, cats.effect.IO

import scala.util.Random

val ioByVariable = {
  val random = new Random(0L)
  val x: IO[Int] = IO(random.nextInt)
  for {
    a <- x
    b <- x
    _ <- IO(println((a, b)))
  } yield (a, b)
}.unsafeRunAsyncAndForget() // (-1155484576,-723955400)

val ioByComposition = {
  val random = new Random(0L)
  for {
    a <- IO(random.nextInt)
    b <- IO(random.nextInt)
    _ <- IO(println((a, b)))
  } yield (a, b)
}.unsafeRunAsyncAndForget() // (-1155484576,-723955400)
```

Listing 18: The IO Monad does not cache values and is referentially transparent.

5 Building a Functional and Reactive Application

5.1 Functional systems in Scala

Scala is not a pure functional programming language. For this reason one has to be careful when choosing third party tools for creating functional programs. Some libraries may rely on mutable state or perform side effects in order to perform their actions, which would break referential transparency.

One first approach to deal with this problem would be to decide having "mostly functional" programs. The problem is that this not completely functional programming model does not work[33]. Thinking that this mixture would give the benefits of functional programs is a fallacy. Referential transparency is given or not, there is no middle ground. If some of the elements which compose a program are not referentially transparent, this is propagated to the whole program.

For this reason, when choosing libraries, it is important to choose foundationally functional ones. In the case of the study presented in this section, the Cats library [4] will be the foundation for building functional programs.

Its documentation describes it as "a library which provides abstractions for functional programming in the Scala programming language. ... A broader goal of Cats is to provide a foundation for an ecosystem of pure, typeful libraries to support functional programming in Scala applications".

At its core Cats provides, among its abstractions, a set of type classes and datatypes. The majority of these type classes are categories from Category Theory such as Monad, Monoid or Functor, with instances for many relevant types. But there are also utility type classes, like the Show typeclass which was shown in Section 2.

However, Cats doesn't provide any utility for interacting with the real world in a functional way. There is a separate project, Cats-effect, [32] which "aims to provide a standard IO type for the Cats ecosystem, as well as a set of typeclasses (and associated laws) which characterize general effect types". This IO type was the one presented in the monads section and its data type provide a set of operations to work with I/O operations such as reading from a database or communicating through the network.

One point mentioned in the IO section was the fact that running an IO computation was the element which made referential transparency lost, and thus it should be the last operation done in the application. Cats Effect provides `IOApp` which is a safe wrapper of a Scala app which expects an `IO` and runs it, hence ensuring that programs can remain purely functional without exposing any kind of impurity.

5.2 The case of study

For putting into practice the concepts we are studying, a simple web application will be developed. It will serve as a basis for an application to organize tennis matches.

Two REST resources will be exposed: players and matches. A player will be represented with an `UUID` and a match will involve two players and an optional player (the winner).

The methods on these resources will be:

1. `GET /{resource}`, which will obtain all the resources saved.

2. `POST /{resource}`, which will create a new resource on the database.

3. `GET /{resource}/{id},` which will return the resource with the given id or a 204 No Content response.

4. `DELETE /{resource}/{id}`, which will delete a resource with the given id.

The resources will be `players` and `matches` and they will be persistent in a PostgreSQL relational database,

5.3 Representing lazy sequences with Streams

One of the main concerns of the `IO` datatype is to separate computation descriptions from their evaluations. They represent a computation to be made which, when evaluated, will produce a single value. However, there are some situations in which, rather than obtaining a single value, the programmer may be interested in receiving an streams of values to be processed. A typical example of this would be working with I/O sources, like obtaining the lines of a source file.

This stream of data needs not be consumed entirely. For instance, a function may attempt to obtain a line of a file matching an specific criteria. As soon as the line is found, the rest of the file doesn't need to be scanned. Although a function which does this specific computation can be expressed using `IO`, the lazy representation for obtaining the lines from the file can not be expressed by it. For this reason, the Stream datatype is created.

The datatype which will be used in our study case is the one provided by FS2, an acronym for **Functional Streams for Scala**, and provides "purely functional, effectful, and polymorphic stream processing" [34].

A FS2 `Stream` has the following type definition `Stream[+F[_], +O]`. In this definition `O` is the type of the elements which are contained in the stream and `F` is the kind which evaluates the effects that the descriptions of the values of the streams may have. In our case of study, this effects will be wrapped by `IO` if the computations are effectful or by `Pure` in case that the operation implies no effect.

For example, the previously mentioned function for getting the first line of a file matching a criteria can be implemented with FS2 in a functional manner like shown in Listing 19.

When implementing functional programs, `Stream` will be a good representation when working with potentially unbounded, non strict and sequential sources of data.

5.4 Serving a Web Server

On top of Cats-effect, Http4s provides a typeful, functional and streamed model for creating HTTP servers.

Http4s routes are modeled as `Kleisli[OptionT[F, *], Request, Response]`, or simplifying the Kleisli Category, `Request => F[Option[Response]]`, what this mean is that the application routes accept a request and return and effectful optional response[5]. The result must be effectful because handling the response may imply performing a side effect, such as reading from the database or accessing a resource from the network, and the response has to be optional because the given route may not have an appropiate handler.

`Request` and `Response` datatype encode the HTTP requests and responses. One interesting element of them is that the body of this request is represented with the type `EntityBody[+F[_]]`. And when looking at its type definition, the entity body is defined as `type EntityBody[+F[_]] = Stream[F, Byte]`. When defining Http4s, it was mentioned that it provided a streamed model for HTTP This streaming model is very important for reactive applications. If a request was not streamed, the response time of the web server would be the following.

$$(\text{latency of receiving the request and processing} + \text{web server operations latency})$$

As the whole request has to be processed on its entirety before being able to work on it this processing time will have to be added to the operations time. If the request is streamed, both the processing of the request and the operations on them can be interleaved, reducing the overall latency of the request. This can also be applied for the response, as the response body is also modeled as an `EntityBody`. By providing a streamed model of HTTP, the applications reactivity is increased by terms of responsiveness.

To create handlers for HTTP requests, Http4s provides the `HttpRoutes.of[F]` method, which accepts a partial function `Request => F[Response[F]]` and lifts it into `OptionT` type, being it `None` in case that the partial function is not defined or `Some` in case it is. For example, the routes for the `/players` resource of the application is presented in Listing 20.

HttpRoutes has a `SemigroupK` instance so it provides the `combineK` method for combining a set of routes into a greater router. Thus, with the `SemigroupK` instance in place, an application router can be composed from other routes as Listing 21 shows.

5.5 Accessing a database

For working with persistent data within an IO, Doobie [35], a functional JDBC layer for Scala, provides a monadic API for describing queries and commands on a database. Op-

[5]In our study case this effect `F` will always be `IO`

erations in Doobie return descriptions of computations within a context. For example the `ConnectionIO[A]` type describes queries and commands that can be expressed when a `java.sql.Connection` is available and that returns a value of type `A` when interpreted. Internally, `ConnectionIO` values are represented as a Free Monad. This means that `ConnectionIO` are combinable with high level operators such as `flatMap`, or even be foldable. Listing 22 shows an example of how to build and combine `ConnectionIO` values.

Although these values represent queries to a database, they are agnostic about specific details like how or where they are going to be ran. This is responsability of the `Transactor` data type.

`Transactor` has information about the connection details for the database but also has information about which execution contexts are being used for the different operations on the database. One of them is the `connectEC` in which threads will be waiting for acquiring a database connection and also the thread pool in which already connected threads will be blocked wating for results. This `Transactor` is parametrized over an effectful Monad `F` target.

Once a `Transactor` is available, it can be passed to a `ConnectionIO` and move the descriptions to the chosen effectful monad where the description of the operations to be performed are no longer dependent on the context of `java.sql.Connection` but operation descriptions deferred on `IO`

5.5.1 Reducing queries mantainance and improving type safety with Quill

Database schemas evolve over time. Users interests change in the timeline and in software developed the need to adapt over time is common. It is important to write code that is able to adapt to these changes and require minimum changes when evolutions occur.

If queries on the database are done with plain SQL, as changes to the underlying representation are made, this queries must be rewritten. For this reason Quill [36] provides a Scala case class to Database schema mapping. These queries are resolved to an internal abstract syntax tree for Quill and translated to a database query at compile time, having a low runtime overhead. This queries can even be validated at compile time against the database resulting in a compilation error in case that the mapping has been incorrectly done.

Quill query model is based in "a practical theory of language-integrated query based on quotation and normalisation of quoted terms" [37] and is able not only to map case classes to database queries but to provide database queries as for comprehensions and normalization of nested queries to provide efficient database access for applications.

Queries are type safe as they are performed over a case class but they even have capabilities such as returning optional values when a value may be missing.

For example, the query shown in Listing 23 extracted for the companion application gets a tennis match from the database with each player information and points for each set and even the potential winner of it as an optional value given that it is acquired as a left join.

5.5.2 Streaming database records

As presented on the HTTP section, when streaming information, the application response times are improved. For this reason Doobie provides a way to return database elements in a streamed manner. This combined with HTTP streaming allows to improve the overall latency in a great manner.

To return streamed content with Quill, it is only needed to apply the `io.getquill.context.stream` function to the quoted query. If combined with the query of Listing 23, to stream it, it would only be needed to do it like `stream(findAllMatches)`. This query would nevertheless need to be transacted as any other Doobie request.

5.6 Making systems reactive with actors

The main trait which made systems reactive was the use of message passing. One tool that can be used for modeling message driven architectures is the Actor model. Its origin dates back to 1973 [38], when they were shown as a way to model artificial intelligence. At a later moment in 1985 the Actor Model was presented as a model of concurrency for distributed systems [39].

The actor model is a great model for efficient distributed systems because it accomplish the three traits that reactive systems have to meet to be responsive. Actors, by definition, communicate by means of messages. Actors don't share state nor do they share code. They consist of an internal state and only communicate through messages.

Actors implementations such as Erlang processes [40] and Akka Actors [41] have resilient trees. This resilience is implemented through Supervision Trees which is a way of having fault tolerant systems within this concurrency model. In a Supervision Tree there are actors whose responsability is to monitor the behaviour of actors in the systems. This supervisors are able to restart actors and thus gracefully recover from errors in case they occur [42], [43].

Finally, actors are also elastic because an actor functionality is self contained. As they receive messages and have a self contained state, an actor can be multiplied. In case that the actor is not able to cope with the load it is required to do, more actors can be summoned and the messages can be load balanced according to the availability of them.

5.6.1 Making functional actors

Akka was mentioned before as an implementation of the Actor Model in Scala, however Akka Actors are slightly composable. Its message passing semantics relies highly on side effects and its internal state is modeled using unsafe mutable primitives which make them non referentially transparent.

For this reason, in the scope of functional systems, an actor primitive has been developed to allow the benefits of asynchronous message passing but in a referential transparent manner.

To design such actor every element will be analyzed to come up with an implementation fulfilling the requirements.

Firstly, a mechanism for receiving request and obtaining a response is needed. It will be called a Mailbox and this will be a type representing a function
`mailbox: Input => F[Output]`. As one of the concerns with actors was composability, this mailbox is highly composable as this function can be wrapped in a Kleisli Category and as long as `F` has a monad instance, as is the case of `IO`, the Kleisli Category will also have a monad instance.

As multiple request can come to the actor at a given time, a data structure for storing the multiple messages is needed to hold the incoming messages. For this purpose FS2 provides a FIFO Queue data structure which holds the messages that come to the actor.

In this queue, it will not only be saved the input to the actor but also a `Deferred` structure. Deferred models a value that will be present at a later point in time. Its `get` method will block until an element is set with its `complete` method, in our case the actor responsible for handling the request.

As the element may not be delivered at any point on time, a timeout can be provided to avoid deadlocks. For this reason the Concurrent typeclass provides the `race` method which accepts another effect of the same type `F` and returns an `F[Either[A,B]]` result, being A the type wrapped in the effect in which race was called and B the type of the effect which was applied to handle the race. If the former is the first to be completed, the either will have its value and in the other case it will have the `B` value.

To provide a timeout, the deferred can be raced against a timer that is asleep for the timeout parameter time. If no timeout is provided, the response will be raced against the receiver thread because if the actor thread is killed it is guaranteed that the request will never be processed.

With all these elements the Mailbox implementation is presented in Listing 24.

Once the mailbox is available, the next step is to create the actor responsible of managing the Mailbox. When creating actors the programmer can choose to have an stateful actor which can have an internal state and produce a response based on its input and its state while producing a state transition. For managing the state the `Ref` datatype [44] will be used, which allows for mutable state.

The actor will initialize its corresponding state, if applicable, and then a queue for the mailbox will be created. Then the fiber containing the actor receiver function is created and, lastly, a mailbox with the reference to the actor fiber and queue are created, being it returned so that the clients can send requests to the actor. Listings 25 and lst:statelessactor show the stateful and stateless actor implementation respectively.

The receiving function for each actor will then:

1. Dequeue an element of the queue with its corresponding response `Ref` holder. The queue implementation will block until an element is available.

2. Apply the receive function that the programmer has provided for the input and the corresponding state.

3. Mark the response `Ref` as completed.

This will be repeated forever in a loop until the fiber associated with the actor is terminated. Listings 27 and 28 show the receive function again for stateful and stateless actors.

```scala
import java.nio.file.Paths

import $ivy.`org.typelevel::cats-effect:2.1.3`, cats.effect.{
   Blocker, ExitCode, IO, ContextShift}
import $ivy.`co.fs2::fs2-io:2.4.0`, fs2.io.file, fs2.text
import scala.concurrent.ExecutionContext

implicit val contextShift: ContextShift[IO] = IO.contextShift(
   ExecutionContext.global)

def findLine(filename: String, predicate: String => Boolean):
   IO[Option[String]] = {
  Blocker[IO].use { blocker =>
    file.readAll[IO](Paths.get(filename), blocker, 4096)
      .through(text.utf8Decode)
      .through(text.lines)
      .find(predicate)
      .compile
      .fold(Option.empty[String])((_, s) =>
        Option(s))
  }
}
```

Listing 19: Finding the first line matching a criteria with streams.

```
private val routes: HttpRoutes[F] = HttpRoutes.of[F] {
  case GET -> Root =>
    ???
  case GET -> Root / FUUIDVar(id) =>
    ???
  case POST -> Root =>
    ???
}
```

Listing 20: The router for the `players` resource. Routes implementation is omitted for brevity.

```
private def httpApp[F[_]](): Kleisli[F, Request[F], Response[
 F]] = (
  PlayerController.qualifiedRoutes
  <+>
  MatchController.qualifiedRoutes
).orNotFound
```

Listing 21: The router for the whole application.

```
val dbValues: ConnectionIO[(Int, Double)] =
for {
  a <- sql"select 42".query[Int].unique
  b <- sql"select random()".query[Double].unique
} yield (a, b)
```

Listing 22: A database query which returns a tuple.

```
private def findAllMatches =
  quote {
    val tennisMatchQuery = querySchema[TennisMatchTable]("
 tennis_match")
    for {
      tMatch <- tennisMatchQuery.sortBy(_.id)
      winner <- query[Player].leftJoin(w => tMatch.winner.
 contains(w.id))
      player1 <- query[Player].join(_.id == tMatch.player1)
      player2 <- query[Player].join(_.id == tMatch.player2)
      sets <- querySchema[TennisSetTable]("tennis_set")
      .join(_.matchId == tMatch.id)
    } yield (tMatch, winner: Option[Player], player1, player2
 , sets)
  }
```

Listing 23: A Quill query returning all the elements of a match.

```scala
object Mailbox {

  type Mailbox[F[_], Input, Output] = Input => F[Output]

  def apply[Output, Input, F[_] : Concurrent : Monad](
    queue: Queue[F, (Input, Deferred[F, Output])],
    receiver: Fiber[F, Unit],
    timeout: FiniteDuration = 0.seconds
  )(
    implicit timer: Timer[F], F: MonadError[F, Throwable]
  ): Mailbox[F, Input, Output] = (input: Input) => {
    def getTimeout: F[Unit] =
      if (timeout <= 0.seconds) receiver.join
      else timer.sleep(timeout)

    def getTimeoutError: Throwable =
      if (timeout <= 0.seconds) FiberTerminatedException
      else TimeoutException

    for {
      deferredResponse <- Deferred[F, Output]
      _ <- queue.offer1((input, deferredResponse))
      output <- (getTimeout race deferredResponse.get)
      .map {
        case Right(a) => Some(a)
        case _ => F.raiseError(getTimeoutError)
      }
    } yield (output)
  }
}
```

Listing 24: A Purely functional mailbox implementation.

```scala
def from[
    F[_] : Concurrent : Monad : Timer,
    State,
    Input,
    Output
  ](
    initialState: State,
    receive: (Input, Ref[F, State]) => F[Output]
  ): F[Mailbox[F, Input, Output]] =
    for {
      state <- Ref.of[F, State](initialState)
      queue <- Queue.unbounded[F, (Input, Deferred[F, Output])]
      receiver <- receiver(receive, state, queue)
      mailbox = Mailbox(queue, receiver)
    } yield (mailbox)
```

Listing 25: A stateful actor implementation.

```scala
  def from[
    F[_] : Concurrent : Monad : Timer,
    Input,
    Output
  ](
    receive: (Input) => F[Output]
  ): F[Mailbox[F, Input, Output]] =
    for {
      queue <- Queue.unbounded[F, (Input, Deferred[F, Output])]
      receiver <- statelessReceiver(receive, queue)
      mailbox = Mailbox(queue, receiver)
    } yield (mailbox)
```

Listing 26: A stateless actor implementation.

```scala
private def receiver[Output, Input, State, F[_] : Concurrent
 : Monad](
  receive: (Input, Ref[F, State]) => F[Output],
  state: Ref[F, State],
  queue: Queue[F, (Input, Deferred[F, Output])]
): F[Fiber[F, Unit]] =
  (for {
    (input, response) <- queue.dequeue1
    output <- receive(input, state)
    _ <- response.complete(output)
  } yield ()).foreverM.void.start
```

Listing 27: The receiving function for a stateful actor.

```scala
private def statelessReceiver[Output, Input, F[_] :
 Concurrent : Monad](
  receive: (Input) => F[Output],
  queue: Queue[F, (Input, Deferred[F, Output])]
): F[Fiber[F, Unit]] =
  (for {
    (input, response) <- queue.dequeue1
    output <- receive(input)
    _ <- response.complete(output)
  } yield ()).foreverM.void.start
```

Listing 28: The receiving function for a stateless actor.

6 Conclusions and Futures Lines of Research

6.1 Conclusions

After having studied the Functional Programming paradigm our main conclusion is that it provides a mindset shift when producing programs in terms of their building blocks.

Imperative programming building blocks are procedures that modify the program state. The focus of such procedures in the whole program is not *what* the procedure accomplishes but *how* it does accomplish it. A procedure in imperative programming does not have to return a value to do something valuable to the application. Procedures are so common that the `void` return type is usually used to represent this kind of "functions" that, without returning any result, provide a meaningful purpose to the application.

By contrast, in functional programs, the fundamental building blocks are functions. These functions are seen as opaque representations of the value they return. For this reason, programs are rather declarative than imperative. The focus of these functions is *what* they return and not *how* they do it. Because of this, functional programs can be considered as values rather than as a set of instructions that modify the program state.

In the end, functional programs interacting with the real world are a description of what operations have to be done by terms of an `IO` Monad. This has also been shown with the building blocks used for the companion application. For example, Doobie functions returned a `ConnectionIO` value that was no more than a description of the value that would be returned once a `java.sql.Connection` was provided but these functions didn't perform the operations themselves. The Http4s routes built a description of what to do with an incoming request but nothing more than that. So, it the end, a functional program can be seen as a static description of a series of operations that should be performed. It is not until the program is running that these operations finally occur.

This change of paradigm fulfills the premise that was proposed in the introduction. As functions are mere descriptions of the value they produce, they can be safely reused without affecting the already existing application semantics. This enables faster changes in functional programs, as the programmer has to be bothered only about producing new values and not about breaking existing program functionality.

Apart from that, Functional Programming enables a new reasoning model for programs. Given that functions are mere descriptions of the value they produce, the programmer does not have to be judicious on the context in which they are called. For the programmer, it is only relevant to the value they produce and thus, functions are easier to understand. This concept was presented in Chapter 2 as “Local Reasoning”.

At the same time, the study of reactive systems has provided a set of characteristics to look for in systems in order for them to be responsive. Functional Programming has proven to be a very suitable paradigm for Reactive Applications. Functional programming constraints make programs more resilient to effects such as errors, which can be enforced to be handled explicitly, reducing in this way the possibility of having unhandled errors.

The functional programming model also enables a very safe concurrent model. With the IO datatype, it is easy to describe computations that will be processed in parallel and, with functional programs not relying on mutable shared state, some problems such as race conditions and deadlocks are not present.

Finally, the immutable message passing semantics of Reactive Systems are very aligned with the functional style of programming, as functions results are no more than a value produced by terms of immutable inputs that can be seen as messages.

All that makes Functional and Reactive Systems appropriate for today’s applications demands, because they need not only to be responsive but they need to adapt quickly to changes demanded by customers.

6.2 Future lines of Research

Some of the concepts studied in this work could be explored in more depth. Next, we enumerate some topics that could be further studied as an interesting line of future.

Category Theory and Functional Programming Some categories from Category Theory have been used to combine values in the application, such as the Monad and Kleisli Categories. An study of the fundamentals of such categories and their use in functional programming could help in writting more expressive, lawful and concise programs.

Development of a Functional Actor System An actor implementation based on functional elements have been developed for the companion application but it didn't provide some features available in more advanced Actor Systems such as supervision trees or load balancing. These are really useful but have been omitted in our system due to lack of time. Having an implementation which would provide theses elements would make working with actors more convenient.

Imperative and Reactive Systems In this document the main element of study has been Functional and Reactive systems. Although some benefits have been found, it would be interesting to study Reactive systems in an imperative paradigm, to compare them with their functional counterparts and to be able to choose the most suitable option for a given scenario.

7 Conclusiones y Líneas Futuras

7.1 Conclusiones

Tras estudiar los conceptos que fundamentan la Programación Funcional, la principal conclusión es que ésta implica un cambio de mentalidad a la hora de implementar programas respecto a los elementos con los que éstos se construyen.

Los elementos con los que se construyen los programas imperativos son procedimientos que modifican el estado global del programa. Estos procedimientos no se centran exclusivamente en el valor que producen sino, además, en el *cómo* se producen. Un procedimiento en programación imperativa no tiene porque devolver un resultado para realizar algo constructivo en la aplicación. Esto es tan habitual que en los lenguajes imperativos existe un tipo, `void`, que se utiliza para declarar el tipo devuelto por las "funciones" cuyo propósito no es devolver un valor sino modificar el estado global del sistema.

Sin embargo, en la programación funcional los bloques elementales son funciones. Estas funciones pueden ser vistas como representaciones opacas del valor que devuelven. Por eso mismo los programas funcionales se consideran declarativos más que imperativos ya que dichas funciones se centran en *qué* valores deben devolver, y no en `cómo` devolverlos. Así, los programas funcionales se pueden interpretar como valores más que como una serie de instrucciones que modifican un estado global. en el valor que producen si no en el *cómo* lo producen. Un procedimiento en programación imperativa no tiene por que devolver un resultado para realizar algo constructivo en la aplicación. Tan es así que en los lenguajes imperativos existe un tipo `void` que puede declararse como devuelto por las funciones que no devuelven un valor en sí, sino que modifican el estado global como propósito.

Sin embargo en la programación funcional los bloques elementales son funciones. Estas funciones se pueden ver como representaciones opacas del valor que devuelven. Por eso mismo los elementos de programas funcionales se consideran declarativos más que imperativos. Porque se centran en *qué* valores devuelven, y no en `cómo` los devuelven. Por eso mismo los programas funcionales se pueden considerar valores más que una serie de instrucciones que modifican un estado global.

Al final, los programas funcionales que interactúan con el mundo real son descripciones de las operaciones a realizar dentro del contexto de una Mónada `IO`. Esto se ha visto a la hora de crear los bloques que formarían la aplicación que acompaña a este documento. Por ejemplo, las funciones de Doobie devolvían un valores de tipo `ConnectionIO` que no eran más que descripciones de los valores que se devolverían una vez que se proporcione una `java.sql.Connection`. Las rutas de Http4s eran una descripción de qué hacer con una petición HTTP entrante por lo que, al final, los programas funcionales son descripciones estáticas de operaciones a realizar pero no es hasta que el programa se interpreta que dichas operaciones finalmente acaban realizándose.

Este cambio de paradigma está de acuerdo con las premisas que se proponían en la introducción de este trabajo. Como las funciones son meras descripciones del valor que devuelven en función de sus entradas, éstas pueden ser reusadas sin afectar a otras funciones que ya forman parte dela aplicación. Esto permite que los cambios de un sistema se puedan realizar de manera más rápida, ya que el programador tiene que preocuparse únicamente de producir nuevos valores, y no de si éstos afectan a la funcionalidad ya existente.

Además, la Programación Funcional permite un nuevo modelo de razonamiento sobre los programas. Dado que las funciones son descripciones de los valores que producen, el programador no debe pensar en cómo va a afectar el contexto en el que éstas sean utilizadas. Lo único que debe preocuparle es el valor que éstas producen y es por ello que los correspondientes programas son más fáciles de producir y de entender. Este principio es el que se presentó en el Capítulo 2 como "Razonamiento Local".

Al mismo tiempo, el estudio de la programación reactiva ha revelado una serie de características que los sistemas deben tener para ser capaces de responder eficientemente a las interacciones con los usuarios. La Programación Funcional ha mostrado ser muy apropiada para construir Sistemas Reactivos ya que ciertos elementos, como el manejo explícito de los errores, aumenta la resiliencia de los programas, reduciendo la cantidad de errores en un sistema que pueden acanar siendo no controlados.

La programación funcional también introduce un sistema para la concurrencia muy seguro. Con el tipo de datos `IO` es muy fácil representar operaciones que pueden ser computadas en paralelo y, dado que las distintas partes de un programa funcionale no dependen de un estado mutable compartido, algunos problemas típicos de los sistemas concurrentes tales como una Condición de Secuencia o un Bloqueo Mutuo no puedan ocurrir.

Finalmente, el modelo de intercambio de mensajes de los Sistemas Reactivos está bastante alineado con el estilo de Programación Funcional, dado que los resultados de las funciones son simplemente valores producidos por entradas inmutables que pueden considerarse mensajes.

Todo ello hace los Sistemas Funcionales y Reactivos apropiados para afrontar las demandas a las que se enfrentan las aplicaciones de hoy en día, que no solo tienen que responder rápidamente a las interacciones de los usuarios sino adaptarse constantemente a los cambios que éstos demandan.

7.2 Líneas Futuras

Algunos de los conceptos que se han estudiado podrían explorarse con más profundidad. A continuación se presentan algunos temas que podrían ser estudiados como líneas futuras.

Programación Funcional y Teoría de Categorías Algunas categorías de la Teoría de Categorías se han usado para combinar valores en las aplicaciones, tales como las Mónadas o las Categorías de Kleisli. Un estudio de los fundamentos de dichas categorías y su relación con la programación funcional podría ayudar a escribir programas más expresivos, claros y concisos.

Desarrollo de un Sistema Funcional de Actores En el documento se ha desarrollado una implementación básica de un actor. Sin embargo, alguna de las características más avanzadas de algunos Sistemas de Actores, como el balanceo de carga o los árboles de supervisión, no se han podido implementar debido al tiempo que requeriría. Una implementación que proporcionase dichas características haría el uso de actores más conveniente en el contexto de la Programación Funcional.

Sistemas Imperativos y Reactivos Este documento se ha centrado en el estudio de Sistemas Funcionales y Reactivos. Aunque se hayan encontrado beneficios en la combinación de ambos, sería interesante estudiar los Sistemas Reactivos dentro del paradigma imperativo, para poder compararlos con sus contrapartes funcionales y así poder elegir la opción más apropiada para cada escenario.

References

[1] *The Scala Programming Language.* [Online]. Available: https://www.scala-lang.org/ (cit. on p. 8).

[2] *Ammonite.* [Online]. Available: https://ammonite.io/ (cit. on p. 9).

[3] *Typelevel.scala*, 2020. [Online]. Available: https://typelevel.org/ (cit. on p. 9).

[4] *Cats: Home.* [Online]. Available: https://typelevel.org/cats/ (cit. on pp. 9, 43).

[5] J. Hughes, "Why Functional Programming Matters," en, *The Computer Journal*, vol. 32, no. 2, pp. 98–107, 1989, ISSN: 0010-4620, 1460-2067. DOI: 10.1093/comjnl/32.2.98. [Online]. Available: https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/32.2.98 (cit. on p. 11).

[6] J. Nicholson and C. Clapham, "The Concise Oxford Dictionary of Mathematics," en, p. 876, (cit. on p. 11).

[7] P. Chiusano and R. Bjarnason, *Functional Programming in Scala.* 2013, p. 304, ISBN: 9781617290657 (cit. on p. 12).

[8] *Using the GNU Compiler Collection (GCC): Common Function Attributes.* [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-8.1.0/gcc/Common-Function-Attributes.html#Common-Function-Attributes (cit. on p. 12).

[9] D. A. Spuler and A. S. M. Sajeev, "Compiler Detection of Function Call Side Effects," en, p. 13, (cit. on p. 12).

[10] C. Strachey, "Fundamental concepts in programming languages," *Higher-Order and Symbolic Computation*, vol. 13, no. 1, pp. 11–49, 2000, ISSN: 13883690. DOI: 10.1023/A:1010000313106 (cit. on p. 13).

[11] *Scala Standard Library 2.13.2 - scala.Unit.* [Online]. Available: https://www.scala-lang.org/api/current/scala/Unit.html (cit. on p. 19).

[12] G. L. Steele, “Debunking the ”Expensive procedure call” Hyth or, procedure call implementations considered harmful or, LAMBDA: The ultimate goto,” in *Proceedings of the 1977 Annual Conference, ACM 1977*, New York, New York, USA: Association for Computing Machinery, Inc, Jan. 1977, pp. 153–162, ISBN: 9781450339216. DOI: 10.1145/800179.810196. [Online]. Available: http://dl.acm.org/citation.cfm?doid=800179.810196 (cit. on p. 20).

[13] *Scala Standard Library 2.13.2 - scala.annotation.tailrec.* [Online]. Available: https://www.scala-lang.org/api/2.13.2/scala/annotation/tailrec.html (cit. on p. 20).

[14] C. Hall, K. Hammond, S. P. Jones, and P. Wadler, “Type classes in Haskell,” in *European Symposium On Programming*, 1994, pp. 241–256 (cit. on p. 22).

[15] B. C. Oliveira, A. Moors, and M. Odersky, “Type classes as objects and implicits,” in *ACM SIGPLAN Notices*, vol. 45, Oct. 2010, pp. 341–360. DOI: 10.1145/1932682.1869489. [Online]. Available: https://dl.acm.org/doi/10.1145/1932682.1869489 (cit. on p. 23).

[16] *Scala Standard Library 2.13.3 - scala.Any.* [Online]. Available: https://www.scala-lang.org/api/current/scala/Any.html#toString():String (cit. on p. 23).

[17] *Collection (Java Platform SE 8).* [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html (cit. on p. 24).

[18] *ImmutableCollection (Guava: Google Core Libraries for Java HEAD-jre-SNAPSHOT API).* [Online]. Available: https://guava.dev/releases/snapshot/api/docs/com/google/common/collect/ImmutableCollection.html (cit. on p. 24).

[19] *Mutable and Immutable Collections | Collections | Scala Documentation.* [Online]. Available: https://docs.scala-lang.org/overviews/collections-2.13/overview.html (cit. on p. 25).

[20] *The Reactive Manifesto*, 2014. [Online]. Available: https://www.reactivemanifesto.org/ (cit. on p. 29).

[21] M. Mayer, *Google Marissa Mayer speed research - YouTube.* [Online]. Available: https://www.youtube.com/watch?v=BQwAKsFmK_8 (cit. on p. 29).

[22] Akamai, “The State of Online Retail Performance,” Akamai Digital Publishing, Tech. Rep., 2017, p. 18 (cit. on p. 30).

[23] J. Sussna, “Designing Delivery,” in *Designing Delivery*, O’Reilly Media, 2015, p. 232. [Online]. Available: https://learning.oreilly.com/library/view/designing-delivery/9781491903742/ch04.html (cit. on p. 31).

[24] *Production-Grade Container Orchestration - Kubernetes*. [Online]. Available: https://kubernetes.io/ (cit. on p. 31).

[25] N. J. Gunther, “A General Theory of Computational Scalability Based on Rational Functions,” Aug. 2008. [Online]. Available: http://arxiv.org/abs/0808.1431 (cit. on p. 31).

[26] D. P. Rodgers, “Improvements in multiprocessor system design,” *ACM SIGARCH Computer Architecture News*, vol. 13, no. 3, pp. 225–231, Jun. 1985, ISSN: 01635964. DOI: 10.1145/327070.327215. [Online]. Available: http://portal.acm.org/citation.cfm?doid=327070.327215 (cit. on p. 31).

[27] E. Moggi, “Notions of computation and monads,” *Inf. Comput.*, vol. 93, pp. 55–92, 1991 (cit. on p. 33).

[28] P. Hudak, J. Peterson, and J. Fasel, “A Gentle Introduction to Haskell: IO,” in *A Gentle Introduction to Haskell*, 2000. [Online]. Available: https://www.haskell.org/tutorial/io.html (cit. on p. 36).

[29] *Null References: The Billion Dollar Mistake*. [Online]. Available: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/ (cit. on p. 37).

[30] The Apache Software Foundation, *Apache Tomcat 8 Configuration Reference - The HTTP Connector*, 2020. [Online]. Available: https://tomcat.apache.org/tomcat-8.5-doc/config/http.html (cit. on p. 38).

[31] *Scala Standard Library 2.13.2 - scala.concurrent.ExecutionContext*. [Online]. Available: https://www.scala-lang.org/api/current/scala/concurrent/ExecutionContext.html (cit. on p. 38).

[32] *Cats Effect: Home.* [Online]. Available: https://typelevel.org/cats-effect/ (cit. on pp. 39, 44).

[33] E. Meijer, "The Curse of the Excluded Middle "Mostly functional" programming does not work," *Communications of the ACM*, vol. 57, no. 6, pp. 50–55, 2014 (cit. on p. 43).

[34] *fs2: Home.* [Online]. Available: https://fs2.io/ (cit. on p. 45).

[35] R. Norris, *Doobie*, 2020. [Online]. Available: https://tpolecat.github.io/doobie/ (cit. on p. 46).

[36] *Quill.* [Online]. Available: https://getquill.io/ (cit. on p. 47).

[37] J. Cheney, S. Lindley, and P. Wadler, "A Practical Theory of Language-Integrated Query," *ACM SIGPLAN Notices*, vol. 48, no. 9, pp. 403–416, 2013. DOI: 10.1145/2500365.2500586. [Online]. Available: http://dx.doi.org/10.1145/2500365.2500586 (cit. on p. 47).

[38] C. Hewitt, P. Bishop, and R. Steiger, "A Universal Modular Actor formalism," in *Advance Papers of the Conference*, Stanford Research Institute, 1973, p. 235 (cit. on p. 48).

[39] G. A. Agha, "ACTORS: A Model of Concurrent Computation in Distributed Systems," Tech. Rep., Jun. 1985. [Online]. Available: https://dspace.mit.edu/handle/1721.1/6952 (cit. on p. 48).

[40] Ericsson AB, *Erlang Processes*, 2018. [Online]. Available: http://erlang.org/documentation/doc-10.0/doc/reference_manual/processes.html (cit. on p. 48).

[41] Lightbend Inc., *Akka Actors*, 2020. [Online]. Available: https://doc.akka.io/docs/akka/current/typed/actors.html (cit. on p. 48).

[42] *Erlang – Supervisor Behaviour.* [Online]. Available: https://erlang.org/doc/design_principles/sup_princ.html (cit. on p. 49).

[43] *Supervision and Monitoring • Akka Documentation.* [Online]. Available: https://doc.akka.io/docs/akka/2.5.31/general/supervision.html (cit. on p. 49).

[44] *cats-effect - cats.effect.concurrent.Ref.* [Online]. Available: https://typelevel.org/cats-effect/api/cats/effect/concurrent/Ref.html (cit. on p. 50).

Appendix A
Installation Manual

Requirements: Unix environment with a Bash shell and PostgreSQL database instance with name `fairplay` running in port 5432.

Download the associated source code and navigate to the root folder `FairPlay`. There, run the following commands in a Bash shell:

```
chmod +x sbt
./sbt run
```

The web server will be exposed in port 8080.