



UNIVERSIDAD
DE MÁLAGA



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO 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 2017

Fecha defensa: 7 de julio de 2017

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Keywords:

Contents

1	Introduction	5
1.1	Motivation	5
2	Concepts of Functional Programming	7
2.1	Introduction	7
2.2	Referential transparency	9
2.2.1	Local reasoning	10
2.2.2	The substitution model and equational reasoning	11
2.2.3	Function Composition and safe refactoring	12
2.3	Functional programming constructs and programming languages	14
2.3.1	Statements and expressions	14
2.3.2	Imperative statements	14
2.3.3	Functional expressions	16
2.3.4	Higher order functions	16
2.4	Polymorphism in Functional Programming	17
2.4.1	Parametric polymorphism	17
2.4.2	Type classes	18
2.5	Functional data structures	20
3	Reactive Systems	23
3.1	The reactive manifesto	23
3.2	The reactive principles	24
3.2.1	Responsive	24
3.2.2	Resilient	24
3.2.3	Elastic	25
3.2.4	Message driven	26
4	Elements of Functional and Reactive Systems	27
4.1	Improving Resilience with Computational effects	27

4.1.1	Monads	27
4.2	Example of monadic effects	28
4.2.1	The resilience of computational effects	31
4.3	Leveraging Multiprocessing by the use of Futures	31
4.3.1	Making models reactive by the use of Futures	31
4.3.2	Futures are highly composable	32
4.4	The IO monad as a referential transparent Future	33
5	Building a Functional and Reactive Application	37
5.1	Functional systems in Scala	37
5.2	Representing lazy sequences with Streams	38
5.3	Serving a Web Server	40
5.4	Accessing a database	41
5.4.1	Reducing queries mantainance and improving type safety with Quill .	42
5.4.2	Streaming database records	43
5.5	Making systems reactive with actors	44
5.5.1	Making functional actors	44
Appendix A Installation		
	Manual	53

Introduction

1.1 Motivation

As technology has been evolving, the non functional requirements of software systems have been growing to be more demanding. Nowadays, a typical cloud service is being used.

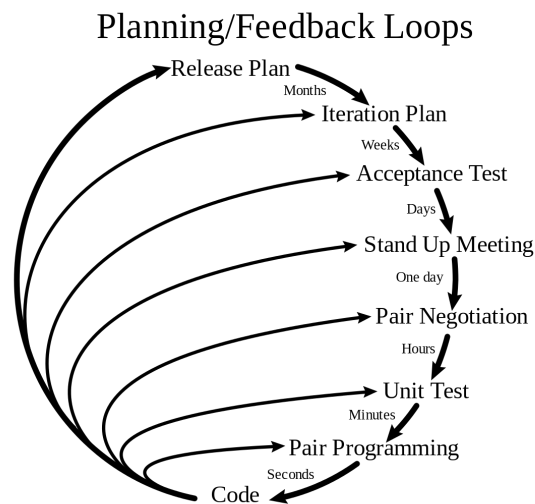


Figure 1: A diagram showing the iterations of extreme programming.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna

dictum turpis accumsan semper.

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 [1].

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)” [2].

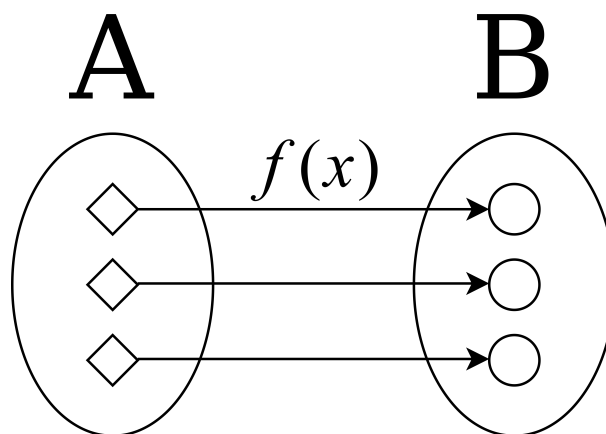


Figure 2: A visual representation of the function f

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

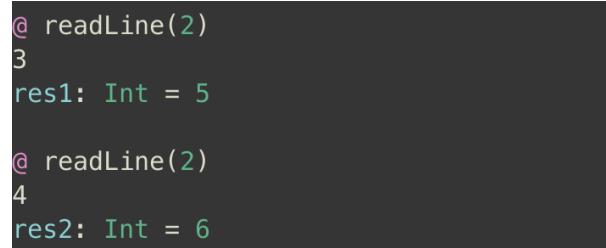
program other than to compute a result given its inputs” [3].

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 3, 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`, 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 3: 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[4] while functions that have side effects are called procedures.

Some side effects that can make a function non-pure are [5]:

- 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 4, this process is showcased.

$$\begin{aligned}2x - x &= \sqrt{16} + 2 \cdot 2 \\x &= 4 + 4 \\x &= 8\end{aligned}$$

Figure 4: Solving a equation by simplifying thanks to referential transparency

This property is called referential transparency [6] 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.

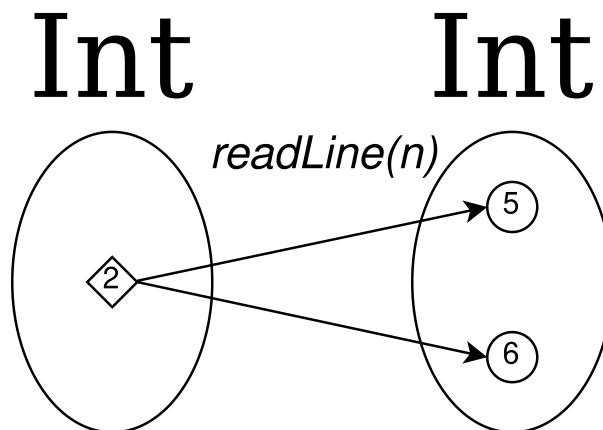


Figure 5: A representation of the relationship introduced by the readLine procedure

Figure 5 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 `Ints`. 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 necessary 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"Greetings $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 `Greetings 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 overridden.

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

```

import scala.util.chaining._

var name = "Foo"

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

greetReversed()
greet() tap println // Greetings ooF!

```

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

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 4 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 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

```

import scala.util.chaining._

var name = "Foo"

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

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

greetReversed()
greetOther("Bar")

greet() pipe println // Greetings Bar!

```

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

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 $f: A \Rightarrow B$ and another function $g: B \Rightarrow C$, both can be composed by mathematical composition $g \circ f$ to create a new function $h: A \Rightarrow C$. 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.

```
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.

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

```
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

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.

```
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

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 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 [8] 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.

2.3.4 Higher order functions

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

¹In 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 [7]

²In 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[9]


```

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

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 identify 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 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

```

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

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 classes³ were defined in Haskell as a way to implement Ad-hoc polymorphism [10]. 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.

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 [11].

³Not to be confused with classes in Object Oriented Programming

```

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

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[[Add reference to API]] 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 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 [12]. 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 [13] 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” [14].

```

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

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

implicit class ShowOps[A](value: A) {
  def show(implicit shower: Show[A]): String =
    shower.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 today's software systems have to handle big loads of data, coming from thousands of concurrent users, which are demanding low latency responses by terms of milliseconds and robust systems with a 100% of up time.

The reactive manifesto [15] calls Reactive Systems the ones able to cope with this expectancies 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 what the traits of reactive systems are. This traits can be categorised hierarchically by the end goal they serve. Being responsiveness the trait which directly provides value to systems and the other three characteristics that reactive systems need to meet in order to be responsive. This relationship can be visualised in the Figure 6.

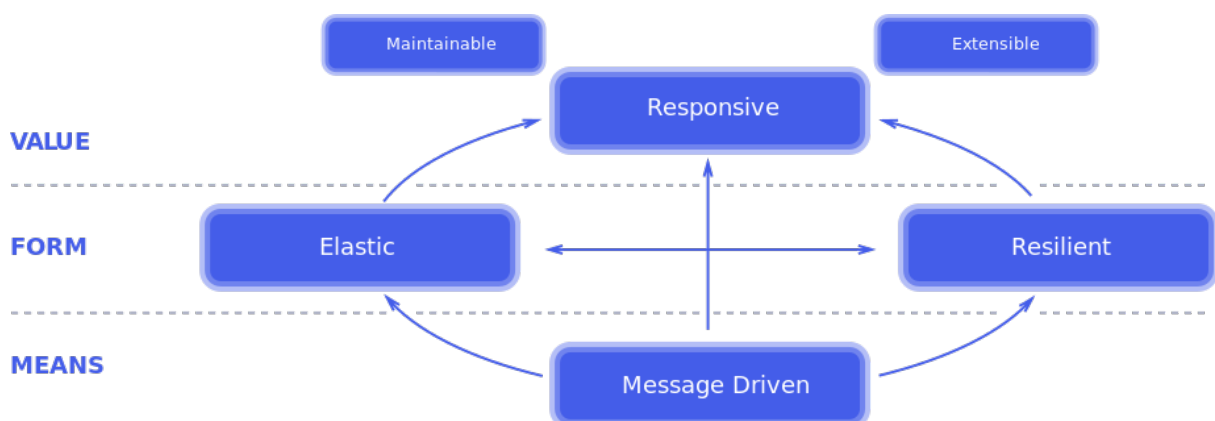


Figure 6: A hierarchical view at the reactive principles.

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%. [**MayerGoogleYouTube**].

More recently in an study developed by Akamai Technologies in 2017 [16] other insights about consequences of pages responsiveness, being some of the most interesting.

- A 100-millisecond delay in website load time can hurt conversion rates by 7 percent
- A two-second delay in web page load time increase 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 request 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 if the chat system is unavailable.

At the same time resiliency 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.

When designing reactive systems errors should be expected to occur. Even if all the possible applications errors in the designers control could be avoided there are other elements of the applications which are out of control, such as network errors or interactions with external systems. This principle is stated in the *Design for failure* approach [**Sussna2015DesigningDelivery**]

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 season 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 with the original design.

Load balancers or more advanced orchestration systems like Kubernetes [17] can handle elasticity on the infrastructure level, allowing replicas of the application to coexist. However applications have to be designed to allow the distribution of work among the instances of the application, this is referred as the 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 [Gunther2008AFunctions], the formula is a variation of the Amhdalh's law [Rodgers1985ImprovementsDesign] and is present in figure 7

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

Figure 7: The Universal Law of Computational Scalability formula

In the formula the parameter N represents the amount of concurrent process running the application. α is the time that is lost because of the need to wait for shared resources to become available. And β 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 to it. Systems have to be designed in order to use efficiently the computer 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, using this mechanism application are decoupled both by space and time.

If modules of the application are communicated by 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 systems be the recipient of the message, allowing **Elasticity**.

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

This decoupling provides greater **resilience** because the asynchronous boundary isolate 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 of failure.

4

Elements of Functional and Reactive Systems

4.1 Improving Resilience with Computational effects

In 1991 Eugenio Moggi [**MoggiNotionsMonads**] described a model in category theory for separating the set of values a object A has from the notion of a computation of it, $T A$, which had an effect on top of this type. This model abstracts from the type A the possible results that the computation could have. Some examples of computations mentioned on the text of interest to this topics are.

- Side effects that modify an set of possible states S : $\text{State}[A, S]$
- Exceptions where E is the set of possible exceptions: $\text{Either}[A, E]$
- Interactive Input/Output which results in an A value: $\text{IO}[A]$

From a functional programming perspective a very interesting finding of that article is that this effect over A can be abstracted and define operations that transform the type A regardless of the effect in which it is contained as long as the effect F has a monad instance.

4.1.1 Monads

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

```

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 by following the types of them. The pure function takes a value `a` and lifts it to the monad context `F`. The `flatMap` operation takes a monad `F[a]` and a function which transform an `a` into a monad with the same context but a possible different type inside `F[B]`. This operation maintains the monad context.

4.2 Example of monadic effects

Then some monads instances can be defined for different effects.

Option Monad The Option Monad is an effect which may have a value of a type `A` or not. An `Option[A]` has two possible values. `Some` 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.

```

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

Either Monad When a function can either succeed or fail the Either Effect encodes this possibility, represented with the 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.

```

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 15: The either monad instance

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

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

```

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 16: The state monad instance

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 the function read from keyboard whould have the type $() \Rightarrow A$ which is a perfectly valid type a pure function could have.

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 it 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 programs, they 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, 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 [AIO]

4.2.1 The resilience of computational effects

In static typed programming languages computational effects offers 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. IO.

This implicit treatment of effects like the use of a `Null` pointer has lead to great number of mistakes, being called by one of the first designer of it, Tony Hoare, its billion dolar mistake.

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

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

4.3 Leveraging Multiprocessing by the use of Futures

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

A Future can be created using the `apply` method of the companion object of the trait `Future`. An invocation have 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 result type of the 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 responsiveness 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 separate threads 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

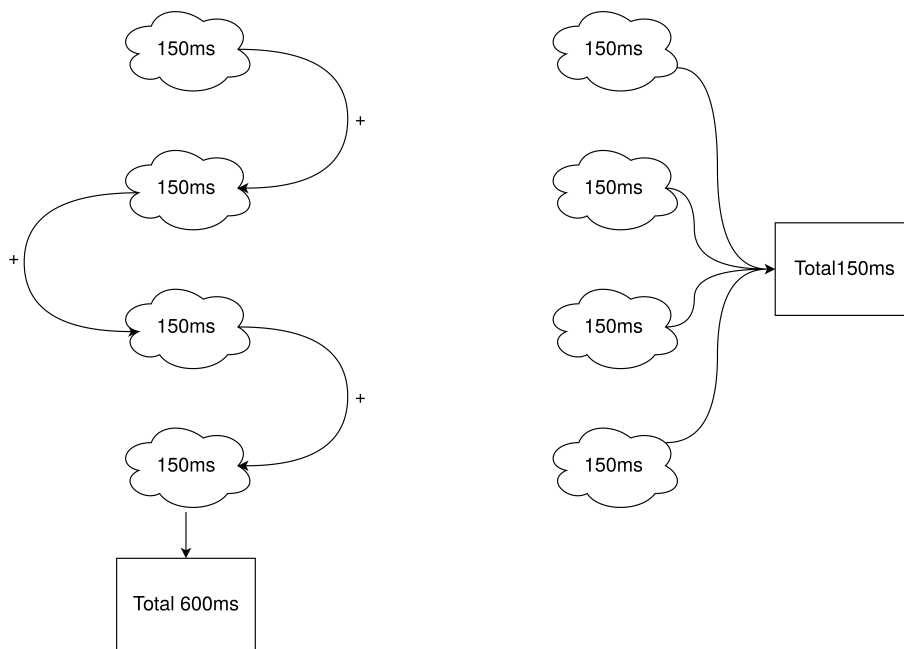


Figure 8: At the left of the figure a sequential run of the blocking operations. At the right a parallel execution which combines the results

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 completion of the inner expression to be completed method of a `Future[A]` allow to transform it by passing description of what should be done once the Future is completed this methods don't block until the future is complete but return a `Future[B]` being B the type of the transformed value from 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 systems from collapsing. In tomcat for example when this threshold is surpassed no more request can be met and the request start to be stacked. Moment in which 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 [`ScalaScala.concurrent.ExecutionContext`]. 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 of Futures, but not of running them.

4.4 The IO monad as a referential transparent Future

Future has many properties that make it an appropriate construct for modeling asynchronous operations. However its operations are not referential transparent. By the definition of it, if Future was referentially transparent the result of the operations on a Future should be the same regardless if they operation functions were composed directly or if saved in variables.

The project Cats Effect [**CatsHome**] provides a standard IO for Scala. This IO provides both synchronous and asynchronous evaluation models. IO provides a referent

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

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

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

```

futureByVariable.onComplete(println) // Success
((-1155484576,-1155484576))
futureByComposition.onComplete(println) // Success
((-1155484576,-723955400))

```

Listing 17: An example of how Future is not referentially transparent

This lack of referential transparency is shown in Listing 17. Two future variables are created on the top level `futureByVariable` and `futureByComposition`. These 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 it into a variable without losing referential transparency as Listing 18

```

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(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 [**MeijerTheWork**], 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 study presented in this section the Cats library [**Cats:Home**] 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 abstraction a set of type classes and datatypes, the majority of this 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 the 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, 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 element mentioned in the IO section was the fact that running the IO 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. Ensuring that programs can remain purely functional without exposing any kind of impurity

5.2 Representing lazy sequences with Streams

One of the main concerns of the `IO` datatype is to separate computation descriptions from the evaluation of them. 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 value 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, 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 isn't needed to be scan. Although a function which does this specific operation can be expressed using `IO` the lazy representation of obtaining the lines of the file can not be expressed by it. For this reason the `Stream` datatype is created.

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

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 streams and `F` is the kind which evaluates

the effects that the descriptions of the values of the streams may have. In the case of study, this effects will be wrapped by IO if the description 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.

```
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

When implementing functional programs Stream will be a good representation when work-

ing with potentially unbounded, non strict and sequential sources of data.

5.3 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 an effectful optional response⁴. The return 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 appropriate 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 to 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

(latency of receiving the request and processing + web server operations latency)

as the whole request has to be processed on its entirety before being able to work on it. 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` 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

⁴In our study case this effect `F` will always be `IO`


```
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

router can be composed from other routes as Listing 21 shows.

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

Listing 21: The router for the whole application

5.4 Accessing a database

For working with persistent data within an IO Doobie provides a monadic API for describing queries and commands on a database. Operations 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.

```

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

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 responsibility 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 waiting 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`

5.4.1 Reducing queries maintenance and improving type safety with Quill

Database schemas evolve over time. Users' interests change in the timeline and in software developed is common to need to adapt over time. 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 these queries must be rewritten, for this reason `Quill` 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, these 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” [CheneyAQuery] and is able not only able to map case classes to database queries but 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 even have capabilities such as returning optional values when a value may be missing.

For example the query shown in Listing 24 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.

```
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

5.4.2 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 24 to stream it it would only be needed to do it like this. `stream(findAllMatches)`, this query would nevertheless need to be transacted as any other Doobie request.

5.5 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 [**HewittAFormalism**], 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 [**Agha1985ACTORS:Systems**].

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 messages. Actors don't share state nor share code. They consist of an internal state and communicate through messages

Actors implementations such as Erlang processes and Akka Actors are also resilient. Supervision trees is the way of having fault tolerant systems within this concurrency model. In a Supervision tree there are actors whose responsibility is 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 [**ErlangBehaviour**], [**SupervisionDocumentation**].

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 more actors can be summoned and the messages can be load balanced according to the availability of them.

5.5.1 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 rely highly on side effects and its internal state is modeled using unsafe mutable primitives which make them non referen-

tially 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 the actor every element of an element will be analyzed to come up with an implementation fulfilling the requirements.

In first place it is needed a mechanism for receiving request and obtaining a response. It will be called Mailbox and is 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 and as long as `F` has a monad instance, as is the case of `IO`, the Kleisli 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 equal effect `F` and returns an `F[Either[A,B]]` being `A` the type wrapped in the effect in which `race` was called and `B` the type that the effect which was applied to `race` had. 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 `get` can be raced against the 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 this elements the Mailbox implementation is presented in Listing ??.

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 produc-

ing a state transition. For managing the state the `Ref` datatype [`Cats-effectCats.effect.concurrent.Ref`] will be used, which allows to have a mutable state.

The actor will start its corresponding state, if applicable, and then a queue for the mailbox, 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 complete.

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

```

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

```

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

```

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


```

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

```

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

References

- [1] 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](https://doi.org/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. 7).
- [2] J. Nicholson and C. Clapham, “The Concise Oxford Dictionary of Mathematics,” en, p. 876, (cit. on p. 7).
- [3] P. Chiusano and R. Bjarnason, *Functional Programming in Scala*. 2013, p. 304, ISBN: 9781617290657 (cit. on p. 8).
- [4] *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. 8).
- [5] D. A. Spuler and A. S. M. Sajeed, “Compiler Detection of Function Call Side Effects,” en, p. 13, (cit. on p. 8).
- [6] 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](https://doi.org/10.1023/A:1010000313106) (cit. on p. 9).
- [7] *Scala Standard Library 2.13.2 - scala.Unit*. [Online]. Available: <https://www.scala-lang.org/api/current/scala/Unit.html> (cit. on p. 16).
- [8] G. L. Steele, “Debunking the ”Expensive procedure call” Myth 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](https://doi.org/10.1145/800179.810196). [Online]. Available: <http://dl.acm.org/citation.cfm?doid=800179.810196> (cit. on p. 16).
- [9] *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. 16).

- [10] 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. 18).
- [11] 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](https://doi.org/10.1145/1932682.1869489). [Online]. Available: <https://dl.acm.org/doi/10.1145/1932682.1869489> (cit. on p. 18).
- [12] *Collection (Java Platform SE 8)*. [Online]. Available: <https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html> (cit. on p. 20).
- [13] *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. 20).
- [14] *Mutable and Immutable Collections | Collections | Scala Documentation*. [Online]. Available: <https://docs.scala-lang.org/overviews/collections-2.13/overview.html> (cit. on p. 20).
- [15] *The Reactive Manifesto*, 2014. [Online]. Available: <https://www.reactivemanifesto.org/> (cit. on p. 23).
- [16] Akamai, “The State of Online Retail Performance | Spring 2017 | Akamai,” Tech. Rep., 2017 (cit. on p. 24).
- [17] *Production-Grade Container Orchestration - Kubernetes*. [Online]. Available: <https://kubernetes.io/> (cit. on p. 25).

Appendix A

Installation

Manual

Requirements:

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.