



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	9
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	13
2.3.1	Statements and expressions	14
2.3.2	Imperative statements	14
2.3.3	Functional expressions	14
2.3.4	Higher order functions	15
2.4	Parametric polymorphism and Type classes	15
2.4.1	Parametric polymorphism	15
2.4.2	Type classes	16
2.5	Functional data structures	16
3	Reactive Systems	19
3.1	The reactive manifesto	19
3.2	The reactive principles	20
3.2.1	Responsive	20
3.2.2	Resilient	20
3.2.3	Elastic	21
3.2.4	Message driven	21
4	Functional and Reactive Systems elements	23
4.1	Improving Resilience with Computational effects	23

4.1.1	Monads	23
4.2	Example of monadic effects	24
4.2.1	The resilience of computational effects	25
4.3	Leveraging Multiprocessing by the use of Futures	25
Appendix A Installation		
	Manual	29

Introduction

1.1 Motivation

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

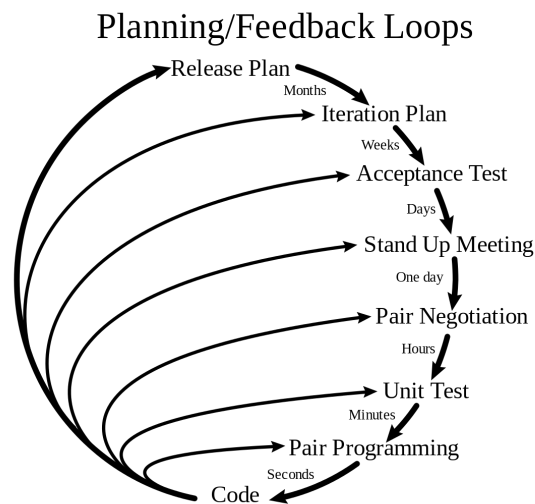


Figure 1: A diagram showing the iterations of extreme programming.

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, by 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].

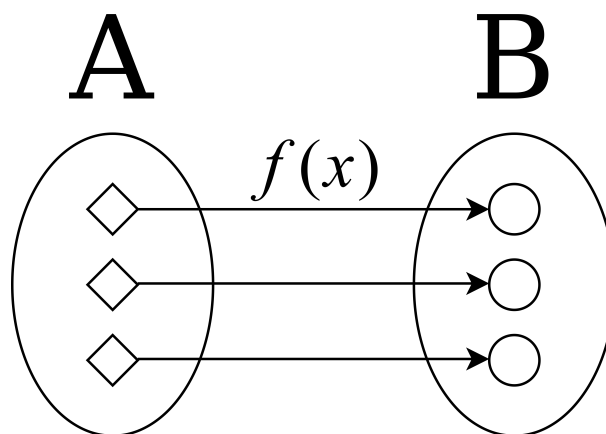


Figure 2: A visual representation of the function f

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

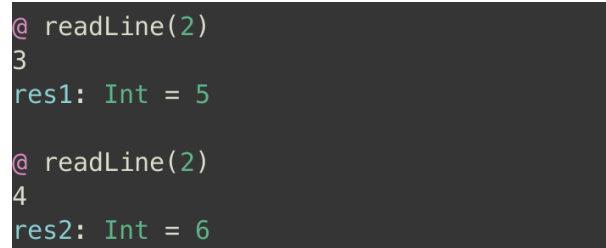
program other than to compute a result given its inputs” [3].

Function constructs in many programming languages don’t have the traits of mathematical functions. One example of it 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) = {  
  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 `res1: 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 language functions not perform as mathematical functions are called side effects. To make differentiation functions that don’t have side effects are referred to as pure functions[4] while functions that have side effects are called procedures.

Side effects that can make a function not pure are [5].

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

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 with referential transparency

This property is called referential transparency [6] and is a very important property of functional programming.

The reason of 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. [[Include another example]]

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 itself and not the context in which its 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 = "Santi"

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 it is possible to analyse that the output Will be `Greetings itnaS !` as the name is assigned to `Santi` first and then is reversed.

```
var name = "Santi"

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

greetReversed()
println(
  greet()
)
```

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

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 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.

```
var name = "Santi"

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

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

greetReversed()
greetOther("Pepe")

println(
  greet()
)
```

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

This 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.

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 its needed to do is 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 of 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 use of the Scala REPL (Read-eval-print loop). This tool allows the import certain functions of a program in a shell which can run them with arbitrary parameters to help to identify the flaws of 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 of why functional code can be safely refactored. The first is, again, referential transparency. is because of function composition. As defined at the beginning of the 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 composed by functions that are at the same times composed of other functions or expressions.

Even programs which define variables which are then used by other functions are at the end functions defined by 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 no more variables are there.

At the very end of the substitutions the only thing that will be left is a pure expression. This is very useful because a function composition that has been used in many places can be factored to an auxiliary function without compromising the behaviour of the functions and improving mantainability of the code.

Listing 5 showcases this behaviour. A pure function like `greetCapitalized` is no more than the composition of previously defined functions.

```

val prependGreeting: String => String = name => s"Greetings
    $name !"
val capitalize: String => String = 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

```

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

With that being said this process can be

This way it is possible to refactor a set of expressions used within a function to a new one. This can be achieved safely as long as side effects are not present in them. [[Follow an example using this]].

This is also a corollary of the substitution principle as how the function composition mechanism works

2.3 Functional programming constructs and programming languages

The functional programming paradigm suits better in some programming languages than others. This usually lies in the constructs a language gives to create 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 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 block of the `while` code 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 at least once it will be true if the loop ends.

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

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

The `if/else` statement of imperative languages has the problem that it does not produce any value. The code block of it has to do 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. Which is in the case that the return of the function is done inside the `if` block `[[Another example]]`.

2.3.3 Functional expressions

Functional programming languages by contrast provide constructs that enable composing programs by referentially 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 the element of it. At a later moment in the document [\[\[Complete with chapter when ready\]\]](#) for expressions will be revisited and explained in greater detail.

Along with the `if` and the `for` expressions a way to iterate and construct functional programs is by using recursion. Recursion is a 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 allocate one extra frame in the 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, 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 [\[\[As always follow an example\]\]](#)

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 other kind of values. As an example a string trimming function can be created. This function will iterate character by character the string and will remove the element if a specific condition is met. The predicate to remove a character is decided by the caller. [\[\[Introduce example\]\]](#)

2.4 Parametric polymorphism and Type classes

2.4.1 Parametric polymorphism

Functional programming can express polymorphism by the use of generic types in function declarations. It allow functions to take parameters with an abstract type. In Scala to use a generic type in a function, the generic must be defined between square brackets after the

¹In case that the `if` expression doesn't contain an `else` and the condition is not satisfied the returned value is the `Unit` [\[7\]](#)

²In Scala in order to tell the compiler to fail if the substitution cannot be done the `@tailrec` annotation can be used[\[9\]](#)

function name. This way the function can be defined one time but used by different types. As an example a list length function can be defined.

[[Example parriba y pabajo]]

This function will take a `List[A]` of elements of a generic type `A` and compute the length of it regardless of the actual type of the elements of the list. When the function is called the actual type `[A]` will be resolved.

2.4.2 Type classes

Although parametric polymorphism is good when the details of the generic type are irrelevant it lacks the possibility to define a behaviour that is associated with the type itself.

Type classes³ was defined in Haskell as a way to implement Ad-hoc polymorphism [10]. A type class is an abstract representation of a behaviour that types which belong to the class posses. If a type can implement the type class behaviour it becomes an instance of it and the type itself belongs to the class.

[[Present example]]

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].

Typeclasses are a very useful because they allow to decouple structure from behaviour. If a specific behaviour is desired for a given type it is not needed to modify the source of the type itself, a type class instance can be provided in a separate source file. This is specially useful when using third party compiled dependencies. If a new functionality it is required for them as they are compiled its source code cannot be modified. However a type class instance can be provided to enrich its capabilities.

[[Rellenar con type classes]]

2.5 Functional data structures

At the beginning of the section among the different kind of side effects one of the mentioned was *Modifying a data structure in place*. If modifying a data structure in place produces a

³Not to be confused with Object Oriented classes

side effect and breaks referential transparency data structures must be defined immutable by default, and as a corollary on this property, operations on them must not modify their internals but return a new one with the changes desired.

Languages that weren't designed to be functional usually define their data structures as mutable ones. In Java, the `Collection` class, root of the hierarchy of a great number of data structures defines most of its operations as mutable [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". The definition 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 this constraints. This has lead to 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 modify 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].

3

Reactive Systems

3.1 The reactive manifesto

As mentioned in the introduction each today's 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 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 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 5.

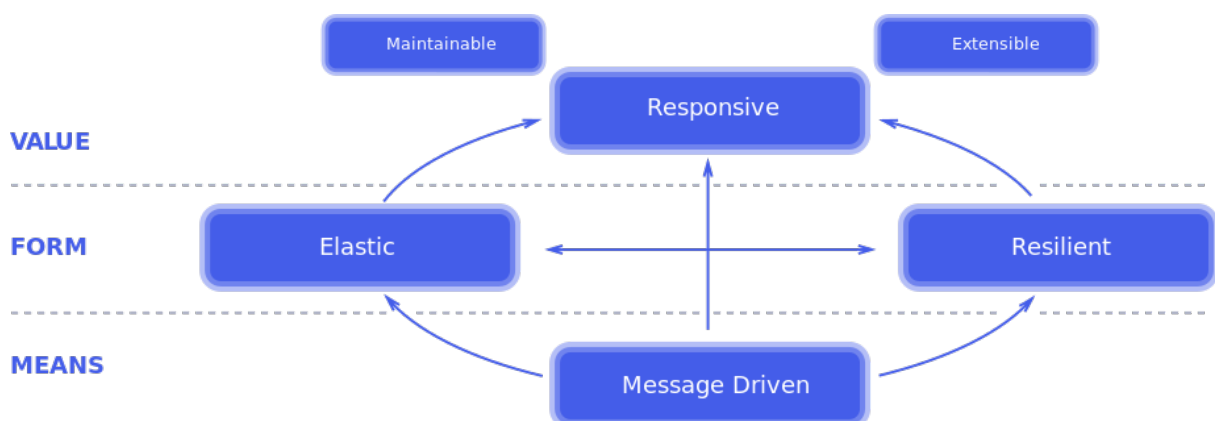


Figure 5: A hierarchical view at the reactive principles.

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%.

More recently in an study developed by Akamai Technologies in 2017 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

With that in mind is important to design systems that respond to user request with low latency in a consistent manner.

At the same responsiveness is about creating systems in which errors can be detected quickly and dealt effectively.

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 and 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 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 [[Insert reference]] which embraces the acknowledge

of possible errors, thus having to make systems which can have errors but needs to be able to 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 [16] 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 refered 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 [[Insert citation]], the formula is a variation of the Amhdalh's law [[Insert citation]] 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 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.

3.2.4 Message driven

The way to achieve Elastic and Resilient systems is by the use of asynchronous message passing by this mechanishm 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 will be the recipient of the message, allowing **Elasticity**.

As messages are asynchronous the sender of the message doesn't 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

Functional and Reactive Systems elements

4.1 Improving Resilience with Computational effects

In 1991 Eugenio Moggi [[Insert citation]] described a model in category theory for separating the set of values a object A could have from the notion of a computation of it, $T\ A$, which had an effect on top of this type. This computation 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 : $State[A, S]$
- Exceptions where E is the set of possible exceptions: $Either[A, E]$
- Interactive Input/Output which results in an A value: $IO[A]$

From a programming point of view 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 was 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`⁴ and `pure`⁵. In the following figure the a trait that defines the Monad typeclass in Scala is defined.

⁴On literature is also usually called `bind`

⁵Also known as `point` or `return`

[[Example]]

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 `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

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.

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.

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 a breaking of referential transparency. For example the function `read from keyboard` would have the type $() \Rightarrow A$ which a pure function could have.

If the side effect is abstracted from the function definition the purity is maintained. This is what the IO monad does. When a function is passed to the IO the execution is suspended. It is not executed until the programmer runs the computation, moment at which purity is lost.

If side effects execution is delayed until the very end of the execution programs can keep functional at its core and the benefits of purity are not lost until the interpretation of the side effects.

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 thread 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`.

Futures favour the use of non blocking computation. 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 largest one. This is represented in figure

7

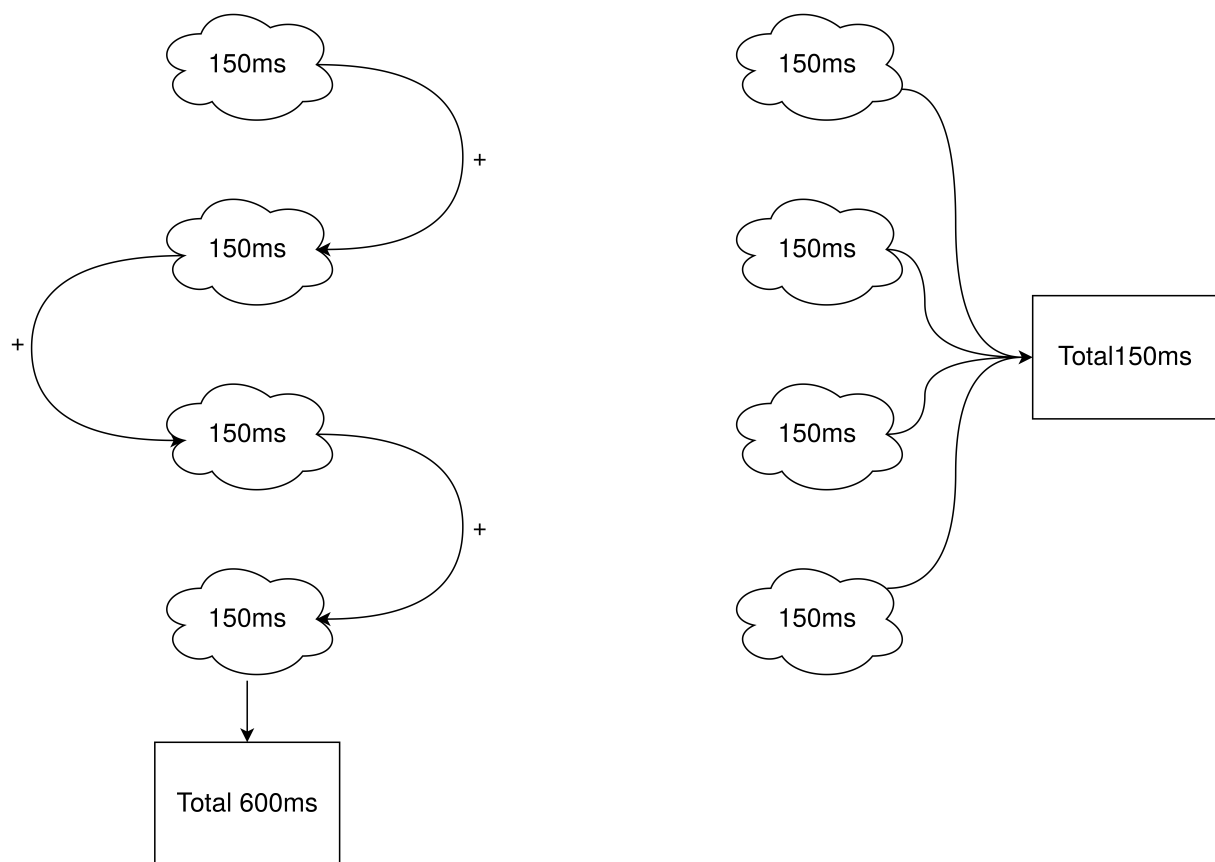


Figure 7: At the left of the figure a sequential run of the blocking operations. At the right a parallel execution which combines the results

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. Sajeew, “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. 15).
- [8] 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](https://doi.org/10.1145/800179.810196). [Online]. Available: <http://dl.acm.org/citation.cfm?doid=800179.810196> (cit. on p. 15).
- [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. 15).

- [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. 16).
- [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. 16).
- [12] *Collection (Java Platform SE 8)*. [Online]. Available: <https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html> (cit. on p. 17).
- [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. 17).
- [14] *Mutable and Immutable Collections | Collections | Scala Documentation*. [Online]. Available: <https://docs.scala-lang.org/overviews/collections-2.13/overview.html> (cit. on p. 17).
- [15] *The Reactive Manifesto*, 2014. [Online]. Available: <https://www.reactivemanifesto.org/> (cit. on p. 19).
- [16] *Production-Grade Container Orchestration - Kubernetes*. [Online]. Available: <https://kubernetes.io/> (cit. on p. 21).

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.