ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

FUNCTIONAL PROGRAMMING IN MODERN SOFTWARE SYSTEMS

Elaborata nel corso di: Ingegneria Dei Sistemi Software

Tesi di Laurea di:
ENRICO BENINI

Relatore:
Prof. ANTONIO NATALI

ANNO ACCADEMICO 2015–2016
SESSIONE III

PAROLE CHIAVE

Functional Programming
Software Engineering
Domain Specific Language
Metamodeling
Functional Languages

To everyone who supported me in these years.
Thank you from my heart.

Contents

Introduction

Latest evolution in main stream Object Oriented Programming (from now on OOP) languages led to a shift towards Functional Programming (FP) abstractions. For instance, we can now find lambda expressions and operations like *map*, *filter* and *fold* in both Java and C# through LINQ or the *stream* interface. In the meanwhile, multi-paradigm languages like F# and Scala, that successfully integrate abstarctions from multiple programming styles, obtained much more consensus. As evidence of that, many big companies like Facebook[30], Twitter[39], Lightbend[35], Intel[11], Linkedin[12], Netflix[38] and many others started to shift towards FP languages or a mixture of different paradigms. As a result this increases the job offers where a functional background is required.[43, 25, 76]

This mix of paradigms and models may lead to a change in the actual approach and method of systems design, in order to take advantage by the benefits of every one of them. Then, a new problem rise up: *how to model and integrate those abstractions in a single project? How to model an hybrid project?* The answer to these questions is not trivial as a superficial approach may lead to poor and inconsistent architecture.

To successfully provide a valid answer to the mentioned questions an investigation of the core concepts of FP and its main design strategies is needed. Then, the system designer needs a tool that embraces these ideas. This tools must helps him to build a model of the system or one of its components respecting the paradigm constraints, providing a fixed and consistent architecture. Throughout this master thesis, all the aforementioned topics will be discussed.

In chapter 1, some of the most successful FP and FP-influenced languages will be presented. Each section will have references to commercial case studies. An the end of the chapter there is also a simple analysis on the market share of those technologies and an overview on the job trends

related to functional programming.

In the chapter 2, the main concepts of functional programming will be debated altogether with the benefits and drawbacks of the style. Moreover, the main design patterns of functional programming will be discussed, showing their definition as well as how to imply them and what is the purpose of each one. Every section will provide examples for a better understanding of the main concepts.

The, the chapter 3 present a basic DSL as a way to define a model that capture the main functional abstractions, generating a code with specific properties and structure. Using this approach, there are also a couple of simpler examples showing the DSL in practice. Then, at the end of the chapter, the future works of the language are discussed with some implementation suggestions.

Finally, an analysis of related works will be presented. The focus of this analysis will be on:

- The developer responsibility for correctness
- The abstraction gap between the current mainstream technologies and functional abstractions
- How the the usage of a DSL can be a suitable solution for previous topics and for the integration between different styles and models

Chapter 1

Functional Programming In Industry

In this chapter, an exploration of the application of functional programming commercially will be done with the purpose of showing how this style is not used only for academic contexts, but it is production ready. The following sections are organized by technology, the most successfully functional languages will be discussed as well as multi-paradigm languages illustrating some of the employment of them into the industry.

Some of these technologies are very old, but their application by companies is far more recent. This means that they are becoming more relevant in the industry due to the new challenges like: concurrency, parallelism and distributed systems for instance. Moreover, the functional paradigm was weakly adopted before because it often demand more resources compared to the imperative or object oriented programs. However, nowadays the CPU performances and memory capacities are not a problem anymore, despite some special cases. For details about the actual benefits and problems of functional programming check out the chapter Functional Programming.

Finally, the influences of functional programming extends also in mainstream languages like C# and Java as shown in the end of the chapter.

1.1 Immutable Database

The idea of a database that do not allow the editing and the deletion of records seems ridiculous at first glance, but this thought of immutable facts

and event is supported by an approach to designing systems called Event Sourcing[21]. Basically, the concept is to store events (or facts) that change the system state, instead of snapshots of the state. The history of events can be replayed later on to produce a certain purpose-specific projection of what the state at any point in time looked like, or it can be queried to discover the actual state of the system in a given time. A common example of an application that uses Event Sourcing is a version control system. In addiction, storage is becoming inexpensive, which makes feasible immutable data storage at scale.

Another few reasons that can lead to immutable databases can be the natural fit with the stream processing, used extensively in the IOT for example, and the possibility to avoid locking the database during concurrent read and write operations. Several database vendors claim an ability to perform non-blocking writes, but that is within the context of eventual rather than immediate consistency.

Moving from theory to practice, there are already some projects that embrace this concept, in particular the Datomic database[8], Microsoft Tango object database[4] and other projects like LinkedIn's Apache Samza and Apache Kafka share this ideas. Check out this article for more details [50].

This trend leads naturally to functional programming languages because they already embrace the principle of immutability to simplify how state is handled, for instance. This proof that the core concept of functional programming has a perfect match into the database field.

1.2 Pure Functional Programming in Industry

This section will contain some of the most important cases of functional programming in industry. On the internet tons of examples can be found, one of the most noticeable is the 'Commercial Users of Functional Programming'[1] that since 2004, every year, report real world functional programming applications. Other resources about real application of functional programming are the sites:

FunctionalWorks[76] where all the job offers about

functional programming are collected

Haskell in Industry [25] it is a specific page of the Haskell's

Wiki about real usage of the language commercially.

1.2.1 Erlang

Erlang is one of the most successful functional programming language. It was designed by Joe Armstrong, Robert Virding, and Mike Williams in 1986 and mainly developed in Ericsson. The first use case of Erlang is to build massively scalable soft real-time systems with requirements on high availability. It is often used in (i) telecom application (ii) instant messaging and (iii) computer telephony. Erlang is very famous for his concurrency model that influenced many new technologies, its high performance when dealing with networking and communication in general and the ability to be updated "on the fly" without stopping the running service.

Those features makes it one of the first choice when the real time is a core requirement and the application have to work under heavy load.

Facebook

Thinking about heavy load and real-time web applications in the world, the Facebook chat is for sure one of them and it is based on the Erlang language [59, 60]. To point out the degree of loading, at the moment of writing there are over 1.79 billion monthly active Facebook users.

Amazon

The Amazon's web service 'SimpleDB' [3] allow to build a database NoSql that is highly available, flexible, and scalable. SimpleDB is part of the Amazon Web Services and has the goal to:

- Provide high availability through multiple replicas geographically distributed
- Free the developers from database management not adding a direct business value: like handle multiple replicas or take care of hardware and software maintenance for instance
- Easy APIs
- Integration with the others Amazon services

- Low price using the strategy ‘pay on consumption‘

Yahoo [22]

Yahoo used Erlang to build up the Delicious bookmark service instead of Perl. In the previous implementation the team faced several problems and challenges like: hard debugging, multi-threading management and bad scaling in general. After the Erlang refactoring instead, they experienced a lots of benefits like (i) increasing performance (ii) scaling (iii) easy live migrations and (iv)fault tolerance. All the data and code was rewrited in both front-end and back-end. In addiction, the Yahoo team developed also a new module from scratch, entirely using Erlang, for rolling migrations.

WhatsApp [64]

Another of the most real-time and heavy load service is the mobile application chat Whatsapp. The service announced more than one billion users in February 2015 and was acquired by Facebook in February 2014 for approximately US$19.3 billion. In the same way the Facebook chat works, also the Whatsapp chat is ruled by Erlang. It helps in terms of parallelization, measurement and decoupling in order to avoid bottlenecks. For instance, When a particular partition suffers some high latency, it wont affect the others.

Open-Source Projects

Erlang is also used in several successful open-source projects. A couple of the most important are:

RabbitMQ [61] Is a message broker middleware with a large community and commercial support. RabbitMQ makes the (i)reliability, (ii)tracking, (iii)persistence, (iv)delivery acknowledgment, (v)publisher confirms, and (vi)high availability it is main features. In addiction, this support multiple languages and operating systems.

CouchDB [17] Is a documented-oriented NoSql open source database software developed by the Apache foundation. Stores the documents in JSON and used Javascript as query language. Some of the most interesting features of CouchDB are:

- Handle a high volume of concurrent readers and writers without conflict. Thanks to ACID semantics.
- Can go offline and sync automatically with the server when the device goes back online.
- Eventual Consistency
- HTTP API

CouldHaskell [74] Is a platform that enables the network communication in the Erlang-style to the Haskell programming language. It provides multiple libraries and patterns for the communication between nodes through TCP and in-memory messaging or others implementations like Windows Azure for instance.

1.2.2 Elixir

Elixir is a dynamic functional programming language that runs on the Erlang virtual machine, then it share most of the properties of Erlang like being distributed, fault tolerant and be particularly appealing to concurrent programming. Compiling to Erlang byte-code makes Elixir fully compatible with Erlang itself, this means that Erlang functions can be called within Elixir and vice-versa. It can be used also for web development and embedded systems.

Unfortunately Elixir has not a big market share compared to other functional programming languages, however, it is starting to get some very good traction in the startups/enterprises world, as well as in the community. The young age of the language is also a factor in the equation, Elixir is was released for the first time in 2011.

There are some big companies like Pinterest and Moz that are using it. A list is maintained here: `https://github.com/doomspork/elixir-companies`

A number of meetups have popped up around the world, the largest ones drawing hundreds of developers.

1.2.3 Haskell

Haskell is a pure functional language created in 1990 by a committee composed by different researchers with the following goals for the language:

1. It should be suitable for teaching, research, and applications, including building large systems.

2. It should be completely described via the publication of a formal syntax and semantics.

3. It should be freely available. Anyone should be permitted to implement the language and distribute it to whomever they please.

4. It should be usable as a basis for further language research.

5. It should be based on ideas that enjoy a wide consensus.

6. It should reduce unnecessary diversity in functional programming languages.

The main reputation of Haskell was, for most years, a language that belongs to academic purposes, not ready for production or with others stereotypes [49]. However, one of the goal of Haskell was, from the start to be suitable for building large systems.

Nowadays Haskell is a reference point for who want to learn functional programming and it has become used also in industry. In this section there are a couple of the most interesting use cases of Haskell in production.

Facebook

Speaking about Facebook again, they also use Haskell for the open source library Haxl. [47, 49, 45, 46] Haxl is a library for concurrent access multiple remote sources like databases and services. Altogether with other services, Haxl contribute to setup a defense against spam on Facebook. In particular, the services, where Haxl is used, manage the access to a fact base from a collection of users that require it. The role of these services is to identify if the users has the authorization to access those data and find if the request comes from a malware for instance. The business logic around the service defines some rules and the request must validate them. Facts come from diverse sources, so in order to run efficiently, rules must be able to fetch facts concurrently. At the same time, correctness and fast iteration demand that rules be kept free of performance details.

Commercial Haskell [69]

Commercial Haskell is a group of companies that have special interest in Haskell for commercial purposes and them contribute to build up open source tools and libraries to facilitate the usage of Haskell in industry. One of the most important tools developed by this group is the 'Stack Building Tool'[70]. It is used to manage the dependencies of an Haskell project organizing the libraries under sandboxes. The stack tool is used also to set up the Haskell project in support to this report, in the appendix 4 there is a little guide explaining some basic command of Stack.

Quickcheck [68]

Quickcheck is an open-source library for random testing of project properties. It was the first library that implements the concept of testing properties through random values generated instead of testing the programs using specified values. In particular, in the test code the programmer writes down a sequence of properties that the program must have. During the testing phase the library produce, at every run, different random values, based on the input of the property and how the test is build. If the code pass the test every time all went good, otherwise, when a failure is detected, the library try different values based on the failure in order to shrink the complexity of the parameters. The default number of run for every test is one hundred, but it can be changed as well.

Starting from this project the same idea was exported to others languages such as: Java[27], Scala[52], F#[67] and many others. It heavily influenced the industrial world making the property-based testing possible.

Intel

In the lab research of Intel the GHC compiler was analyzed for its performances and optimization. [44] The Intel Labs Haskell Research Compiler uses GHC as a front-end, but provides a new whole-program optimizing back-end by compiling the GHC intermediate representation to a relatively generic functional language compilation platform. During the realization of this compiler version different benchmarks were preformed over different kind of problems and the results compared to the native GHC compiler.

Google

Google use Haskell in company internal projects and in particular with the project Ganeti.[62] The project Ganeti is a virtual machine management tool building on top of existing technologies and open source software. The main responsibility of the system are:

- Disk creation management.
- Operating system installation for instances.
- Startup, shoutdown and failover between physical systems.

Check out the website of the project for more details: `http://www.ganeti.org`

1.2.4 ELM

Elm[14] is a pure functional programming language specifically designed for the web applications and graphical interfaces web-browser based. Introduced and designed in 2012 by Evan Czaplicki in his Thesis this language in highly influenced by Haskell and exhibit some of his features like: strong typing, modularity and immutability with the addiction of interoperability with Javascript, HTML and CSS.

Despite his young age, Elm earn a lot of consensus, both in the open-source world, with a growing and rich community, as well as in industrial one. Some examples of the application of Elm in production can be found here [24, 16].

1.2.5 Scheme and Clojure

Scheme and Clojure are functional programming languages based on Lisp and compiling to JVM byte-code. Clojure is the one with major success in the industry, whereas Scheme is mainly used for education purposes. Some of the main companies that use Clojure are: Facebook, Amazon, Oracle, Atlassian, Wallmart, eBay, Spotify and Red Hat. [26, 9]

1.3 Highly Influenced languages

Currently, in the industry there are a couple of successful multi-paradigm language that were designed from the start with the idea to integrate the functional concepts. Moreover, this languages grab the attentions of all the curious programmers or managers that want to try the functional style in production, because them allow to choose the programming style depending on the situation.

1.3.1 Scala

Scala[42] is one of the more successful language of the last years, it first appeared in 2004, developed by Programming Methods Laboratory of École Polytechnique Fédérale de Lausanne. Scala is based on the JVM, compiling in Java byte-code, and most of its syntax remind the Java one. Anyway it has all the features of a typical functional language and a lot of syntactic sugar that made Scala much more concise than his parent.

Speaking of Scala applications commercially we have tons of examples:

Twitter the famous social network use Scala for its back-end and develop a lot of libraries based on the language. [56]

Lightbend [35] this company, previously known as Typesafe, put his focus on building several open source infrastructures and frameworks based on Scala for the developing of reactive applications an provides training, consulting and commercial support on the platforms. Their tools are designed for different fields, from web frameworks(play [36]), to concurrency libraries (Akka [33]), big data platform (Spark[37]), micro-services (Langom [34]) and others.

Linkedin also the leader in professional networking uses Scala to build up its system. /cite{Scalalinkedin} Recently it was acquired by Microsoft.

Others to find other examples of Scala in production you can check out them here. [2, 66]

1.3.2 F#

F# is a multi-paradigm language from Microsoft that was introduced in 2005, five years after the well known C# language. Its syntax is based on the ML language and is highly compatible with the OCaml language. As well as C#, it works over the .Net platform, so it is completely compatible with the others platform's languages and can use the libraries of the .Net framework. One interesting characteristic of F# is that it is a functional first language, this means that it is designed to favor this programming style over the others.

F# is used by a lot of companies around the world, probably not as famous as the software houses mentioned on the Scala subsection, but the trends[41] indicate that usage and popularity of F# is generally growing. A couple of interesting examples of F# used in production are the following:

Jet.com [31] is an e-commerce based on F# and they are very enthusiast about this technology. Here there is an interview from the F# Conference 2016 [32].

Waagner Biro In this case the F# language was used to architect the Cladding of the Louvre Abu Dhabi Dome in symbiosis with Rhino Script. [65].

Others as well you can check out others enterprise realities here [18].

1.3.3 Javascript

Javascript is the standard language for the develop of web pages on the front-end and recently it is also often used on back-end side. It is supported by all the major browsers and it is used to perform dynamic content on sites, with it, you can: manage the DOM of the page, react to DOM's events and perform almost any kind of computations. As a standard, Javascript is used by almost everyone, from the leaders of the market, to small realities.

Javascript was born as a multi-paradigm language, so it contains also some functional programming concepts Some of the most important functional features of the language are:

- Functions as First Class Citizens

- Closures
- Higher Order Functions
- Partial Function Application and Currying
- Recursion
- Anonymous Functions

In the section 2.2 those concepts will be explained and discussed. Altogether with functional features Javascript enables a lot of other programming styles like (i) event-driven (ii) imperative, (iii) prototype-based and (iv)asynchronous. In addiction, with its dynamic typing it is one of the most flexible programming language and leave in the programmers hands several ways to accomplish a task. Anyway, this can also be a big problem sometimes because can easily and quickly transform a Javascript code base in a mess. Teams must put a lot of effort to maintain the code as clean as possible and consistent to some code styling rules. Moreover, the lack of a type system force the system designer to move all the constraints deriving from a type system into a testing suite in order to avoid run-time exceptions and ensure correctness.

To overcome these problems, the few years have borne witness to an explosion in the number of Javascript frameworks, libraries and tools to provide a fixed architecture to projects, add missing features and more. Nowadays, one of the most important choice of a team facing a new problem is to choose the Javascript tools. This choice is often mainly driven by the team knowledge, but issues comes back when, during the building of a project the team spot a requirement that do not has a clear mapping to the chosen tool or if the next version of it contains big break changes. Take as an example the famous Angular framework, passing from the version one to the next it changes a lot of API. In this case the problem was solved allowing the coexistence of both Angular versions in the same project in order to develop new features in the newest version and incrementally refactoring the oldest ones.

In this direction, the Javascript standard adds, at every release, new abstractions directly into the language to solve several problems that at the moment are addressed to frameworks and libraries. A noticeable example of that is the introduction of promises as a solution to the well known callback

hell, flattening the structure. As a result, developers has to keep up to date constantly, often from both sides: language and frameworks specifications.

A possible solution to this problems can be the usage of Javascript as *assembly language of the web*:

> Javascript is an assembly language. The Javascript + HTML generate is like a .NET assembly. The browser can execute it, but no human should really care what's there. - Erik Meijer

The idea here is to put an actual compiler and a language over the Javascript, adding mainly type checking and syntactic sugar for other abstractions. In addiction, almost every of them allow transparent interoperability with plain Javascript in case something is missing in the DSL. Among them, some are highly influenced by functional programming and in particular there are porting from functional languages to Javascript, for example:

Fable Is a porting of F# to Javascript based on the Babel compiler.

GHCJs Haskell to Javascript compiler.

ScalaJs Scala to Javascript compiler.

ClojureScript Closure to Javascript compiler.

Another noticeable project with the same goal is the Elm programming language introduced previously.

Using this approach a team can leave all the responsibilities about the management of the underneath infrastructure, like update to the latest version of Javascript for instance.

1.4 Functional Programming in Main Stream Languages

The Functional paradigm also influenced some of the most important and active main stream programming language. In order to illustrate a couple of ideas that cross-over to the most popular languages this section will exhibit the features introduced in the C# and Java languages that have foundation in the functional programming paradigm.

1.4.1 C#

Compared to Java, C# language is the fist to add some functional features. This is probably caused by the coexistence of the F# language on the same platform. At the state of art, a typical C# developer can start to write a lot of code using functional abstractions. [63, 71]

The Business application of C# are countless as well as Java, it is the forth most used language in the world, at the moment of writing, looking at the Tiobe Index.[6] In the following are listed some of the main features included in the language:

Functions as First-class Values This is for sure the most important foundation of the functional paradigm and the possibility to treat some of the C# methods as first class citizens is a huge contamination of C#.

To introduce this idea in C# the Microsoft team added a special type called 'Delegates'. When a delegate is declared is like if a method signature is declared as well and it means that now on, in that class or where the delegate is in scope, its name can be used exactly as a type and every variable/argument of that type would be a method reference instead of a simple variable. This means that a developer can, for example, store function as variables or create collections of functions, pass them to other methods and apply them to fetch the result.

In later versions of the language a more concise way to express the same concept was introduced: the function types, predicates and actions. They simply remove some of the boiler plate code of delegates and are built-in the .net platform.

In the end, to makes this ideas more flexible, functions can be created dynamically through the lambdas or anonymous delegates. A noticeable additional features linked to the lambdas is the type inference of parameters type. In fact in the delegates we are forced to strictly express the type of parameters while in lambda form they are inferred by their usage.

Closure The idea of closure is very simple and very important in a functional context. Them, mixed with the idea of functions as first class citizens, enhance the flexibility of the language. Closures will be explained in the section 2.2.1.

Recursion Recursion is another simple idea, but very powerful. With recursion we can call a function inside itself. It is a feature often used in functional programming to simulate the classic imperative loop. One of the most simple example of recursion is the implementation of the factorial function, when the result is a recursion call of the function itself with lower values until the reaching termination condition, when the parameter is equal to one.

Partial Functions (Currying) With the features of higher order functions, lambdas and functions as first class objects the language also gain the partial application as well as currying. Check out the definition of currying and partial application in the section 2.2.1.

LINQ LINQ is an extension of the C# language and it stands for '.Net Language Integrated Query'. It is mainly used for querying the databases in synergy with the Entity Framework and to manage the XML documents, but LINQ can also provide useful extension method to collections. Right away, this seems to have nothing to do with functional programming at all, but comparing the LINQ's API with some typical operands and higher order functions found in functional programming languages there are several similarities. For example, LINQ implements operations like map, fold and reduce. In fact most of the methods provided by LINQ are higher order functions.

Furthermore, when a typical programmer have to deal with LINQ he must consider his laziness. This means that when a LINQ's query is stored into a variable it is not immediately executed, but it will be executed only when some computation is performed with that data. For example, when a *foreach* statement is performed. This behaviour can confuse and lead to errors and bugs the programmers that do not know this aspect, but it is actually very useful in term of performance, because the useless data will never be evaluated, and the database will be queried only once.

Reactive Extension The reactive extension library is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators. The .Net version of this library was the first one that implemented the concepts of the reactive programming paradigm and it highly influenced other libraries

that export those concept using different technologies.

The reactive paradigm focuses mainly on the data flows and propagation of change, but it is also highly coupled to the functional paradigm as well. In fact also in reactive libraries there are several higher order operators. Them works as transformation of data and them can be chained to build a data flow. As a proof of that, often reactive and functional programming are related together under the name of 'Reactive Functional Paradigm'. Moreover, the idea of observable, that is core in most of the reactive libraries, is the duality of the most known IEnumerable.[15]

This ideas are in Haskell[51] and Elm[13] as well.

Back to the role of the reactive extension and reactive programming in the enterprise, a couple of noticeable examples are:

Netflix [7] Is the most famous company on the internet providing HD streaming videos, from films to TV series. Netflix contributes to many open source reactive libraries and uses them in production. Netflix alone handles the 37% of the US internet traffic.[29]

Lightbend [35] This company makes of reactive and Scala a key point of his platform.

1.4.2 Java

The Java language also added different functional abstractions recently with the Java 8 version. This helps to reduce the verbosity of the language itself and made it more concise and flexible dealing with collections for instance.

Java is, and was in the past years, by far the most used programming language in the world according to the Tiobe Index[6]. A list of the main functional features in the language are:

Lambda Expressions and Functional Interface When a beginner Java developer starts to build it is first GUI app he has to deal with the concept of event handler to control the behaviours of GUI elements. Most of the time this means create a in-place anonymous object with one single method that contains the logic attached to the GUI event. Before the Java 8 this was the typical approach to this situation.

However, with this new version of the language, the lambda expressions were introduced and now the same developer can avoid a lot of boiler-plate code and directly insert the logic more clearly.

This is only one of the examples where the lambda expressions helps today developers. Another benefit that comes with the lambda expressions is the possibility to avoid parameter types because them can be automatically inferred by the compiler thanks to the idea of 'functional interface'. In order to explain the relationship between this two concepts and what is a functional interface we must consider that the lambda expressions can be compared to a single and only method of an object. In this way we can map the concept of lambda expression to the motto of object oriented: "Everything is an Object". At this point it is necessary to understand where a lambda expression can be used and the answer still very simple: *where it is required an object that implement an interface with a single method.* That interface is called a 'functional interface'. Previously these interfaces are also called Single Abstract Method interfaces (SAM Interfaces), and there are a lot of examples of them like: Java.lang.Runnable, Java.awt.event.ActionListener, Java.util.Comparator, Java.util.concurrent.Callable and others.

Stream Interface Starting from the previous idea of functional interfaces and the LINQ extension for the .net platform, also in Java 8 was introduced a way to manage collections in a functional style with the stream interface. Referring to the Java documentation, the stream interface is *a sequence of elements supporting sequential and parallel aggregate operations.* [53]

The Stream interface is like the Java counterpart of the LINQ library discussed in the section 1.4.1. In fact the stream interface has a lot of higher order function as well to manipulate collection in general.

Closure Refer to the section 2.2.1 to learn more.

Recursive Functions In Java we can call a function from its body triggering a recursion.

Partial Application and Currying Check out the section 2.2.1 where this concepts are explained.

1.5 Functional Programming Popularity and Jobs

After the analysis of several languages, it is worth watching on the number of jobs and usage of the functional technologies commercially and also in open source. Taking as a reference different kinds of online sources for job searching, source code repositories and programming language indexes, like: stackoverflow.com, github.com, indeed.com redmonk.com, tiobe.com and Pypl.github.io it is possible to have an overview through time of the usage of functional languages.

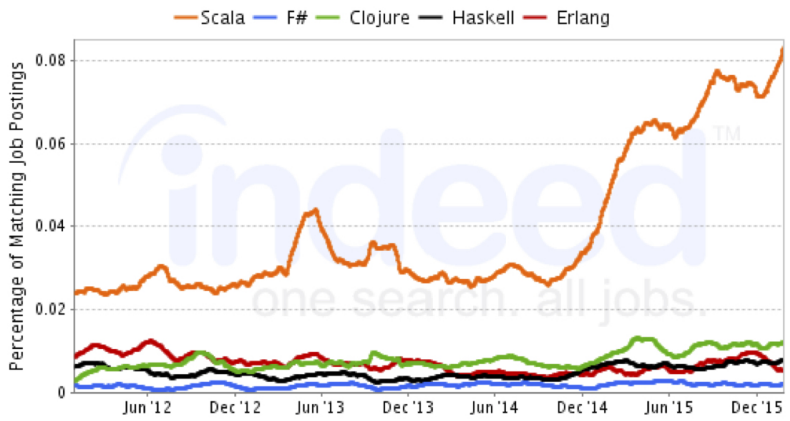


Figure 1.1: Jobs Trends from Indeed.com - Short Term

Starting from the jobs and comparing functional languages together we can see from figure 1.1 that the most used one of the previous mentioned in this chapter is the Scala language. The causes of this success can be several, from the possibility to still program in and object oriented style carrying some of the most typical and well known abstractions, like the *class* concepts to a lot of syntactic sugar. Another interesting observation is that functional languages was at their level in 2009-2010 in figure 1.2.

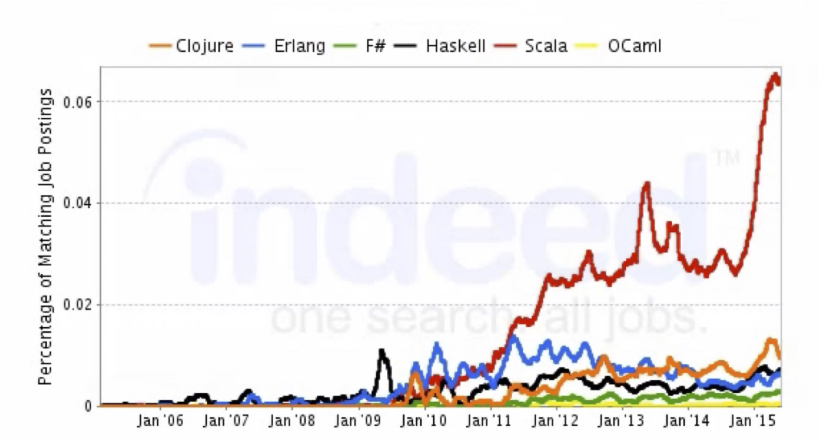


Figure 1.2: Jobs Trends from Indeed.com - Long Term

After the job frequency of the previous figures, also comparing functional languages by contributors exhibit an uptrend (figure 1.3).

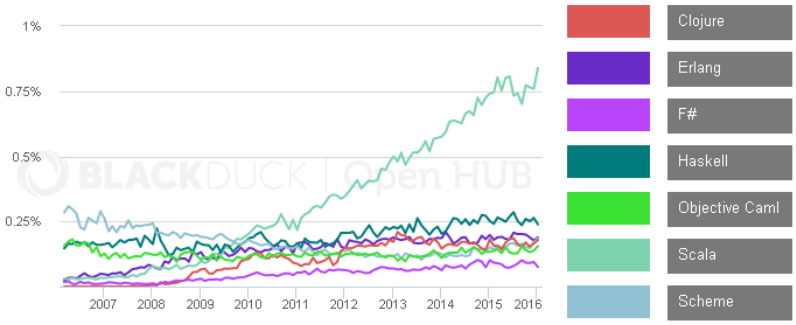


Figure 1.3: Repository Contributors Trends OpenHub

Finally, comparing the functional programming share on Github to the total market share them are quite higher. In fact, there are a few programming languages that gets the majority of the total share like: Java, C#, Javascript, PHP and Python. In the site `http://langpop.corger.nl` all this statistics are reported in a nice popularity chart. In figure 1.4 there is the current snapshot at the moment of writing.

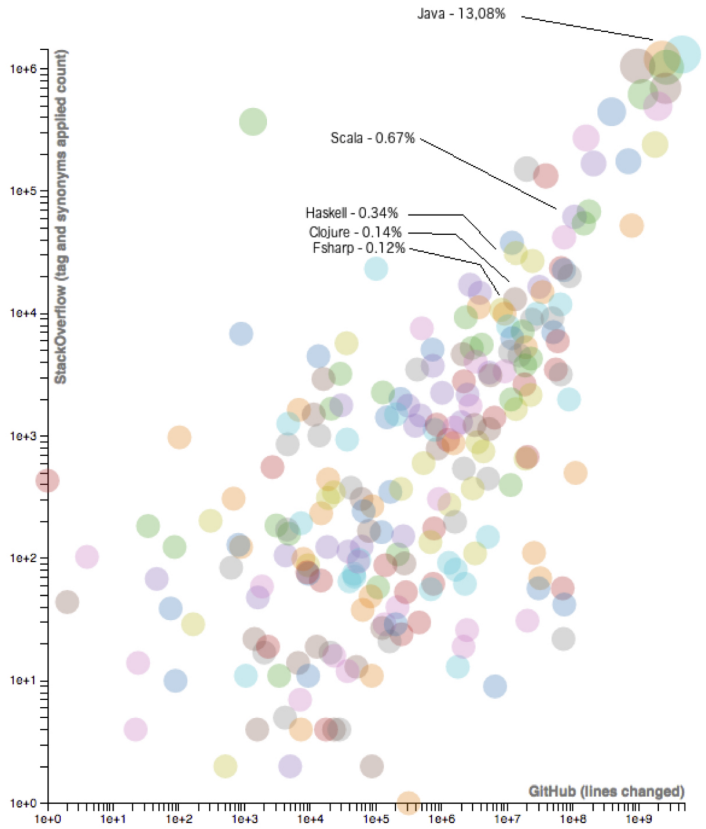


Figure 1.4: Github Language Popularity Chart Based on Line Changed and Stack Overflow Tagged

Other online interesting consultable resources on this topic are: `http://stackoverflow.com/research/developer-survey-2016` and `http://githut.info`

Chapter 2

Functional Programming

The chapter Functional Programming In Industry witness the importance of the functional style in enterprise and how it influenced several languages. Instead, during this chapter we dive a little into the theory and explore its concepts and ideas, but first is important to understand why the functional style becomes a trend analyzing briefly its benefits and drawbacks.

Benefits [28, 75]

Concurrency and parallelism Data structures are immutable by default, sharing state and avoiding locks is much easier. In addiction, functional programming makes a strict distinction between pure functions, without side-effects, and impure functions. This also makes handling parallelism easier. With immutable elements no synchronization mechanisms are required and in those cases where a shared resource is needed between different threads the mutable state is confined into specific areas of the program. An example of this is the Mvar discussed in the section 2.4.2

Conciseness and Declarative It generally takes less lines of code to solve the same problem and describe 'what' is required to do to solve the problem instead of 'how'. A more declarative code can be better understand by non-technical team.

Correctness The type system of a functional programming language is

typically more expressive compared to an object oriented one and it prevents a large number of errors at compile time, like null reference exceptions. Furthermore, designing a system based on pure functions and types, helps tremendously the function-composition and the testing phase. Unfortunately, dynamic languages lack this kind of support for correctness, so in those cases there is higher probability to incur into a run-time exception. There Is an idiomatic phrase, often used by functional developers, that enclose the correctness benefit of functional programs:

> Once your code compiles it usually works.

Code Reasoning Thanks to immutability and referential transparency the understanding of what the code does is way more simpler. Most of the code result in a deterministic computation, so mistakes related to changes of values over time, state management and others are confined. The developer knows exactly the value of almost everything at compile time, so the code reasoning becomes simpler and this reduce the number of times when debugger is needed.

Optimization Using the Lazy evaluation and constructing function through the tail recursion allow an additional optimization. For instance, the compiler and garbage collector can adopt several optimizations. In addiction, in functional programs we can represent some structures like infinite collections.

Composition For the functional programming paradigm the function composition is a key point. Therefore, compose smaller and simpler functions creating bigger ones, using a bottom up design is much more natural and easier. Moreover, thanks to higher-order functions, smaller and more general modules can be reused more widely, easing subsequent programming.

Testing Most of the time testing is used to validate mutation, so with immutability, altogether with pure functions, helps to separate where things change. If the places where changes occur are isolated by severely restricting mutation, then there is a much smaller space for errors to occur and have fewer places to test.

Drawbacks

Learning Curve What is beyond doubt is that, learning to develop in a functional programming style in the right way is far more difficult that in object oriented or imperative, to do it well requires a substantial investment of time and effort. From a different point of view this aspect can also be a benefit because the higher learning curve act as a filter and a magnet: filtering out lazy developers and attracting the most passionate ones. Obviously the learning curve is subjective from person to person and depends heavily on their background.

Performance This seems a contradiction to the previous 'optimization' point, but here the subject are the special cases where performance is a priority. Under this condition, the imperative and object oriented code works better than the functional one due to the lower level of abstraction: the imperative and object oriented code reflect better the hardware architecture with the idea of compute one instruction or statement at a time, instead the functional programming that is based on the lambda calculus. Typically with imperative style the developer has more control over memory consumption and computation optimization. Anyway, is very difficult to compare programs because them depends on several factors like: compiler, different implementations, parallelization and the skill of the developer as well. [58]

Support, Resources and Tools As you can see from the Tiobe index[6], at the moment of writing none of the top ten most used languages are purely functional. This makes harder to find support, resources and tools than a more popular language and programming style.

Memory Usage The lazy evaluation and immutability made more difficult to predict the time and space costs of evaluating and in average a functional program require more memory space. However, at the same time the compiler can take advantage from these program properties and apply different kind of optimizations. As well the garbage collector can be more effective.

Hiring As already said in the previous point, due to the learning curve and effort required by the learning process of a functional language, it

is more likely that someone starting functional programming will give up before realizing the promised productivity gains. From the point of view of a manager, this can be viewed at the same time as a pitfall of a benefit because it makes also more likely that a functional programmer is a more skilled programmer. [23] Anyway, section 1.5 report the increase of functional jobs, so learning functional programming becomes more appealing.

2.1 Brief History

The functional programming history starts in the 1936, when the lambda calculus was formalized by Church and Rosser. This is the foundation of all the functional languages because it describe the computation model using only functions.

Another milestone in functional programming was the LISP in 1958. The LISP language is computationally complete and is based on the S-expression and the M-Language. The former allow to define structures like lists and trees due to a pairing operation, the dot. The latter is used to define functions and their application and conditional expression in which the S-structures can be used. An important aspect of LISP is the possibility to define recursive functions, but it has not the concepts of higher order functions. However, McCarthy, the creator of LISP, shows that M-language expressions and functions can be easily encoded as S-expressions and then defines in the M-language functions, eval and apply, that correctly interpret these S-expressions.

The Algor 60 is not a functional language, but is in this list due to some similarities with functional programming in its evaluation. For example, the default passing mode is *call by name*, the binding of variables recall the beta reduction and procedures can be passed as arguments.

With ISWIM, a peer of Algor 60, in the early 60', higher order functions are defined and used without difficulty. In addiction, the ISWIM paper also has the first appearance of algebraic type definitions used to define structures. This is done in words, but the sum-of-products idea is clearly there.

As a successor of the ISWIM language comes PAL in the late 60'. One of the main interesting ideas in PAL was the organization of code by layers

that divide the layer with mutable variables and assignments for instance.

In the 1972 comes the SASL language that had *let ... in ...* and *rec ... in ...* for non-recursive and recursive definitions. The only method of iteration was recursion and it implements the tail recursion. An expression in SASL can be evaluated at compile time, so it can be replaced by its value. In the evolution of the language some keywords were dropped in favor of others. The pattern matching was introduced and the language becomes lazy. Anyway, the language still type-less as LISP and Erlang.

After the SASL, the Miranda language added to the SASL features the abstract data types and polymorphic type discipline of Milner. Miranda was a product of Research Software Ltd, with an initial release in 1985, and subsequent releases in 1987 and 1989. Miranda was the predecessor of Haskell (1.2.3). All the details can be found here [72].

Another family of functional languages rise in the middle seventies from the ML language and it is the base of languages like OCaml and F#. Some important features of this language are:

- Call by value evaluation
- Parametric Polymorphism
- Garbage Collector
- Static Typing
- Pattern Matching
- Abstract Data Type

The ML language do not have the lazy evaluation, but it can be simulated to produce infinite lists for example.

Both the standard ML and Haskell were standardized in 1997-1998.

2.2 Main Concepts

This is the core section of this chapter because here it contain the explanation of the core concepts from the functional programming using simplest examples. The idea is to skip some of the theoretical and technical part in favor of a more fluent lecture, but keeping the focus on where and when

those abstraction can be useful. Sometimes, be too technical can drop the concentration from what really matters: how employ and integrate all the abstractions in the right way in everyday developing.

Here, we scratch the surface of these ideas and give an overview of them, for a more deep understandings of those concepts several resources can be found on the web.

2.2.1 Function First Class Citizen

Function Definition and Side Effects

The most compact definition of functional programming can be: *programming with functions.* Therefore, the function has a core role in this style. The idea of function comes first of all from math. A mathematics function can be summarized in: *a relation between a set of inputs and a set of permissible outputs with the property that each input is related to exactly one output.* This is exactly the same definition of a function in functional programming.

Analyzing what means to work with a mathematics functions, often called pure functions, we can find that:

- If the result of a function is not used then call that function is useless in the first place.

- If two functions are independent between each others then them can be called in any order and in parallel. Their execution is thread safe.

- If a function is called with the same argument then the previous result of the same call can be used instead of re-execute it. This idea is called referential transparency and is not a new concept at all, in fact it can be found in the command query responsibility segregation pattern where the queries return results and do not change the state of an object, so them are referential transparent.

- If a function's result is never used, than the function will not be executed at all. This is the basics of laziness and can be found in different kind of tools and libraries. In section 2.2.4 these concept will be discussed a little further.

So if a function is a pure function it must have those properties. Sometimes, the idea of a pure function can be related to a lookup table where every input is linked to one output, and often this is actually done also in object oriented for performance purposes (eg. Erlang tables).

At this point a developer can argue that is not always possible to properly encode every possible computer computation inside a mathematics function, and this is true. For example, a random value generator is an operation that does not require an input or we can think of it as a function with always the same empty input, and every time it returns a different output. Another example is when a function crash with an exception, and we can go on listing infinite other examples. Those situations had to be managed by every language, and functional ones are no exception. Further, in this chapter will be showed how functional languages can handle those situations with only functions, simulating different context computation. In particular, how they encode, using pure function, functions that are not pure, called side-effect functions, using specific design patterns.

Functional Programming Citizens

Back to the title of this section, is quite obvious that, in a functional program, the functions are the citizen exactly as the objects are for the object oriented one. This means that them can be treated as an object in a Java program for instance, so a function can appear anywhere in the program:

1. Passed as argument
2. Assigning function to variables
3. Returned from another function

A function that handle other function as in the first two points is called an *higher order function*. Higher order functions enables *partial application* and *currying* adding an additional degree of flexibility to the program.

The listing 2.1 provide some simpler Haskell examples of functions as first class citizen and higher order functions.

```
1 module HigherOrderFunctionCurryingClosure where
2
3 algebraApplicator :: (Int -> Int -> Int ) -> Int -> Int -> Int
4 algebraApplicator f x y = f x y
```

```haskell

applySum :: Int -> Int -> Int
applySum x y = let f = algebraApplicator (+)
               in  f x y

applySumLambda :: Int -> (Int -> Int)
applySumLambda x = \y -> x + y

ifThenElse :: (a -> b) -> (a -> b) -> Bool -> (a -> b)
ifThenElse f g c = if c then f else g
```

Listing 2.1: Higher Order Functions Example

The function ‘algebraApplicator‘ is a function that takes as parameters: a function, that takes two Integer and return an Integer, and two integer. Then, it returns an Integer. The body is very straightforward, simply apply the two parameters to the input function and return the result. This is one of the most simpler example of an higher order function because it takes a function in input. Moreover, in the function ‘applySum‘ we can see how the ‘algebraApplicator‘ is used and in particular it is partially applied and fixed to the ‘f‘ symbol, that is immutable. Partial application will be discussed below in this section. This two functions provides an example for the first two point above, the third one is the function ‘ifThenElse‘ where the if-then-else construct is applied, but the values in input and output are also functions. Notice also how, in the signature of the ‘ifThenElse‘ function, some generics types named ‘a‘ and ‘b‘ are used to define the actual signature of input functions.

```
> algebraApplicator (+) 2 2
4
> applySum 2 2
4
> (ifThenElse (applySum 2) (algebraApplicator (*) 3) True) 2
4
> (ifThenElse (applySum 2) (algebraApplicator (*) 3) False) 2
6
```

Figure 2.1: Higher Order Funciton Application

In figure 2.1 we can see the execution of the function in the listing 2.1. The first two executions are straightforward, but the last two might need some explanation. As parameters of the 'ifThenElse' function there are the applications of the 'applySum' and 'algebraApplicator' functions. It is easy to discover that those functions lack of a parameters themselves. This transform them directly in a function that needs that parameters, matching the 'ifThenElse' signature. Finally, the application of the 'ifThenElse' returns a function and it is applied to the last passed value. Notice how the changing of the Boolean value also changes the result due to the different returned function from the 'ifThenElse' one.

The previous behaviour of the figure 2.1 was possible because of Partial Application and Currying, them often are used as synonyms, but are not the same concept:

Currying Allow to transform the evaluation of a function with multiple parameters in an evaluation chain of functions with a single parameter. Starting from a function with two parameters, the first function of the function chain is a function that require the first parameter of the initial function and return a function that require the second parameter and return the result of the initial function applied to both parameters. This can be applied to every multiple parameter function. A signature example of a three parameter function curried: $x \to y \to z \to f(x, y, z)$

Partial Application It is a slightly different concept from currying because it allow to reduce the arity of a multi-parameter function applying the function to a subset of its parameters and get back a function that require the remaining parameters. From a function of n parameters we can apply $m < n$ parameters to it and return a function that require $m - n$ parameters.

What is very powerful in a functional approach is the possibility to handle function exactly as data, so, instead of moving data around and transform it through several computational step we can compose and manipulate functions together and move them to the data. This seems to be quite tricky and an overcomplicated, but it becomes very useful when all the needed arguments are not present at the same time, but them are available through different stages. Then, instead of passing all the set of arguments

to every stages of the computation we can directly apply them and pass the new functions. This topic is also reminds to the Reader Monad, go to section 5 for more.

In summary, with currying, we can automatically transform a function with N parameters in an higher order function with N-1 or less parameter that returns the first function wrapped around a simple lambda.

```
> (applySumLambda 2) 2
4
> (algebraApplicator (+) 2) 2
4
```

Figure 2.2: Currying and Partial Application Execution

Watching the function ‘applySumLambda‘ of listing 2.1 and its execution in figure 2.2, it is an example of the currying. In fact it deconstruct the previous version ‘applySum‘: from a function that takes two parameters into a function that require only one parameter and return a function of one parameter. If ‘applySum‘ had instead been of more than two parameters, then this process will create a chain of one functions returning one after each others. In figure 2.2 there is also an example of partial application using the function ‘algebraApplicator‘. In this case, the function is applied to two arguments and return a new function requiring the last one, that is immediately provided. Haskell automatically provides partial application and currying based on the number of parameters passed to a function.

Finally, the concept of *Closures* is also very important in functional programming. Closure is the ability of functions to access variables from containing scopes, even when those scopes no longer exist. With the closures the developer can grab the current state of the program, or whatever value in the scope, at the time of the function creation and embed it inside the new function. It can be compared to currying, but the values used inside the function are not coming from the input but from the scoping where and when the new function is created. An example of closure can be the function ‘applySumLambda‘ in listing 2.1 where the ‘x‘, first argument of the outer lambda, is used in the returning lambda, so when the function is applied with an argument it will be bounded inside the returning function.

Combined, with higher-order functions allow the programmer to write functions that return other functions which “remember” the arguments passed

to the function that generated them, and are able to use those arguments elsewhere in their body.

2.2.2 Function Composition

Composition is a concept that is crucial within software engineering in general and not only in functional programming. During the analysis of an hard problem the common practice is to reduce it in a sequence of smaller problems and then compose them to solve the first one. In addiction, the main goal of every IT professional and company is to enforce reusability of code, so the trend in programming is to build different libraries and frameworks specialized in solving different kind of problems and be as much generic as possible.

In functional programming the composition becomes easy because compose function is very natural. The idea is again stolen from math and consist in the process of create a new function that, when applied, is equal to apply two function in sequence. For instance, the functions $f : X \to Y$ and $g : Y \to Z$ can be composed to yield a function which maps x in X to $g(f(x))$ in Z. Intuitively, if z is a function of y, and y is a function of x, then z is a function of x. The resulting composite function is denoted $g \circ f : X \to Z$, defined by $(g \circ f)(x) = g(f(x))$ for all x in X. The $\circ$ operand is the math notation for function composition. In Haskell is called *after* and its operand is the dot. Notice how the role of types is core here, they has to match, so having a language with a type system can really helps because it ensure the correctness of the composition.

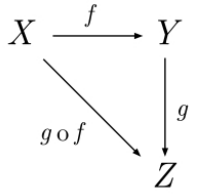


Figure 2.3: Pure Function Composition Commutative Diagram

Working with pure functions makes the function composition straight forward, but considering side-effect functions there is no way to compose them properly. Sometimes, those functions do not have parameters or a result, can be time dependent, blow up or have other unpredictable behaviour.

To solve this problem in functional languages there are proper abstractions, discussed later in this chapter, that ensure the composition also between this kind of functions.

In an object oriented world we can also compose computations, but, due to the abstraction of the object, often this implies some boiler code to manage:

- Creation and destruction of object's instances
- Several Internal or global states that effect behaviours, as a result the output is driven not only by the input
- Object's dependency
- No clear distinctions between pure functions and side effect functions

For those cases many specific design patterns was formulated and a software designer as well as developers must know them in order to maintain project properties like: maintainability, flexibility and reusability.

As an example of pure functions composition, in the listing 2.2 there are two pure functions composed together using both the point operator and nesting functions calls. Using the point operator result in a more compact form of composition, but under complex situations can also be difficult to read. This style of programming is called 'pointfree style'.

```
1  module PureFunctionComposition where
2
3  pureFunctionComponent1 :: Int -> Int
4  pureFunctionComponent1 x = x * x
5
6  pureFunctionComponent2 :: Int -> Int
7  pureFunctionComponent2 x = x `mod` 5
8
9  pureFunctionCompositionPointStyle :: Int -> Int
10 pureFunctionCompositionPointStyle = pureFunctionComponent2 .
       pureFunctionComponent1
11
12 pureFunctionComposition :: Int -> Int
13 pureFunctionComposition x = pureFunctionComponent2(
       pureFunctionComponent1 x)
```

Listing 2.2: Pure Function Composition Example

The following code executed in the Haskell interpreter behave like in figure 2.4. Remember that those function are not accessible from outside right now because them are pure functions so cannot perform input and output operations. The REPL, read evalutate print loop, allow us to try different computations, but it is like if the REPL environment is inside the program. To access those function from outside and perform meaningful computation effect-full functions must be inserted.

```
> pureFunctionComponent1 2
4
> pureFunctionComponent2 5
0
> pureFunctionCompositionPointStyle 5
0
> pureFunctionComposition 5
0
> pureFunctionCompositionPointStyle 6
1
> pureFunctionComposition 6
1
```

Figure 2.4: Pure Function Composition Execution

2.2.3 Immutability

The immutability is one of the concepts that most scares developers that moves from another style to the functional one. Anyway, there are different context in which the immutability property is required also in the common Object Oriented paradigm. For example, the value objects in the domain-driven design approach. The idea is very simple, (almost) everything is immutable, there are some way to have mutable state having Haskell as a reference. When a symbol is linked to some function or data it cannot change during the rest of the program lifetime. With immutable data the developer is forced to a different approach to solve problems and design the solution and the presence of this constraint is related to the high learning curve required to become a good functional developer. Anyway, later in the chapter is showed some of the way which a state can be simulated inside a functional programming using pure functions, but the default is immutability.

This feature of pure functional languages it is actually a restriction of developer's freedom and at first glance it seems a nonsense, but there are

several reasons and benefits deriving from it. Them are listed in the beginning of this chapter.

Recursion

Immutability prevent assignments and as a consequence the typical imperative loop is denied. Then, in functional programming, the recursion is used instead. Recursive functions are function that calls itself in the body with different parameters. In this way the loop in simulated and goes on and on until a base case is reached that cause the termination of the recursion. Every time a function is called its activation record is placed in the stack until its termination. During an heavy recursion this can easily cause a stack overflow. To solve this problem most compilers, especially the compilers of functional languages, implements the *tail recursion* optimization. Taking as an example also a multi paradigm language such as Scala, there is a way to instruct the compiler that a particular function is implemented using tail recursion, so the compiler can imply the required optimization.

The tail recursion consist in the reuse of the same function activator record at every recursive call instead of placing a new one. To perform this optimization the function must be designed in order to have the final action of the function to be the call to itself (tail call).

The most famous example of recursion is the Fibonacci's sequence. In listing 2.3 there are both versions: tail recursive and non-tail recursive.

```
1  module Fibonacci where
2
3  fib :: Int -> Int
4  fib 1 = 1
5  fib 2 = 1
6  fib x = fib (x-1) + fib (x-2)
7
8  fibTailRecursive :: Int -> Int
9  fibTailRecursive x = fibHelp 0 1 x
10                       where
11                         fibHelp a b n = if n > 0 then fibHelp b (
       a+b) (n-1) else a
```

Listing 2.3: Recursion and Tail Recursion Example

2.2.4 Lazy Evaluation

The lazy evaluation is an evaluation strategy that delay the evaluation of an expression until it is needed, it is also called *call by need*. This means that if an expression is never used inside a function it will not be evaluated and if an expression was already evaluated that the compiler can reuse the same result instead of evaluate that expression again. This is obviously true until no side effects are involved. As mentioned in the beginning of the chapter this strategy can be a benefit as well as a drawback due to the better performance as well as unpredictable memory usage. In addiction, with the lazy evaluation a developer can declare infinite data structures because them will not be stored at once but only the needed element will be evaluated.

This strategy can be simulated also in imperative languages like in the .NET framework with the LINQ library. In those mixed environments some awkward behaviours can happen due to the mixing of lazy evaluation and greedy evaluation. In fact, a developer has to know well what and when an evaluation strategy is applied in order to avoid those unwanted behaviours leading to bugs.

In listing 2.4 there is the definition of an infinite list and in figure 2.5 an example of its evaluation using the ‘take‘ primitive that get the first ‘n‘ elements of a list. It could not be possible without laziness.

```
1 module LazyEvaluation where
2
3 infiniteList :: [Int]
4 infiniteList = [1..]
```

Listing 2.4: Lazy Evaluation Example

```
> take 100 infiniteList
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,5
0,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,
97,98,99,100]
```

Figure 2.5: Infinite List Execution

2.2.5 Type System

Altogether with all the features, the type system in a functional program gain a very important role. This happens because it can check for invalid

program at compile time more effectively and ensure the program correctness. The risk of run-time exceptions decrease significantly, in fact, speaking about pure functional programming, you can often hear the phrase: "if a Haskell program compiles, it probably works". The types tend to be much more precisely described than the type systems of typical object oriented or procedural languages, for instance with the pattern matching the compiler can cause errors if some pattern is missing and also the algebraic data types helps in designing the problem domain.

Algebraic Data Type (ADT)

An algebraic data type is a composite type formed by combining other types. The most famous ADTs are:

Product Type Is also called tuple or record and consist in a set of types, called fields, linked together in a single type. The allowed values of the new type is a cartesian product of the field's values.

Sum Type A sum type is a way to create a disjoint union of types. To do that, during the declaration of a sum type a set of constructors are specified, a constructor is a function that works in the realm of types, then it can have zero or more types as parameters and returns the sum type where it is used. If all the constructors of a sum types have zero parameters, then the sum type is called an *Enumerable Type*. Then, to create a sum type you have to use one of the constructors specified in the declaration, passing some values of the types of the constructor parameters. In this way you have actually choose at creation time one of the options, so this particular sum type has a fixed value from one of its constructors, but the fact that you can construct a type using different ways and types made possible the disjoint union.

The word *algebraic* comes from the possibility to define a true algebra based on this types. For instance, we can represent a product type with A and B as an actual product $A * B$, in fact, if we fix the value of A the values that the product type can assume are: the fixed A value and all the values of B. The same happens with the sum type, where the possible values it can assume are the values of A and B. Furthermore, also the functions can be encoded in this algebra through exponentials: if you have a function from A to B then it is a B^A. Those ideas comes from the category theory that is the

foundation for several functional programming abstractions, a lot of others details about the subject were skipped in spite of simplicity. Anyway, know those theory can help a lot for deeply understand the concepts and how thinks works, but them are not necessary for using those ideas and have a briefly understanding.

What is more interesting is the possibility to define generic and recursive algebraic data types. With generic data types a developer can define a type parameter during the declaration of the ADT and use it as a normal type. Then, every time a new instance of this type is created the compiler bound that variable to the type used in that instance. This allow to construct polymorphic types. The recursive ADTs instead are types that in definition recall itself: for instance, a binary tree can reuse the binary tree type to define the type of its branches or a list can reuse the list type to define its tail. Both generic and recursive type features can be mixed together.

Synonym Type

The ability to define synonyms for existing types does not seems to be a great feature of a type system, but this can help a lot during the design of a system instead. Also in a functional program, as in an object oriented happens for interfaces, there is a phase of the analysis where the signature of functions are defined. During that phase, with the help of type synonyms we can make those signatures more fluent and human readable switching between *String* to *Name* even if the *Name* type is a *String* for instance. However, the most important reason that justify type synonyms is always the same: with them the compiler can check and toggle errors during function composition.

Type Classes

A typeclass is a sort of interface that defines some behavior, or another way to see it can be: grouping types together by some common API. If a type belong to a specific type class then that type has some specific operations available on it. Those operation can be specified for that particular type or can refer to a parametric implementation of them during the creation of the type classes. Type classes can be very useful to:

- Keep the functions as generic as possible, because we can specify a

function with parametric types but of specific type classes, so inside this function we know that the generic input types has specific operations

- Define some common generic abstractions explained in the Design Patterns section of this chapter

In some way it reminds to Interfaces in object oriented. Exactly like interfaces, type classes enforce some contract in types and we can also define an hierarchy between type classes in order to add abstraction on top of others.

Finally, the listing 2.5 reports a collection of examples of all the previous types.

```
1  module Types where
2
3  -- SumTypes
4
5  data Direction = Left | Right
       -- Enumerate Type
6
7  data Kind      = Club | Heart | Spade | Diamond deriving Eq
       -- Enumerate Type
8
9  data Tree a    = EmptyTree | Node a (Tree a) (Tree a)
       -- Generic Recursive Binary Tree
10
11 data List a    = Nil | Cons a (List a)
       -- Generic RecursiveList
12
13 -- Product Types
14
15 data Pair      = P Int Double                          -- Pair of
       an Int and a Double
16
17 data Card      = C Int Kind                            --
      Structure of a Card
18
19 -- Type Synonyms
20
21 type PhoneNumber = String
22 type Name = String
23 type PhoneBook = [(Name,PhoneNumber)]
24
```

```haskell

-- Record Type

data Person = Person { firstName :: Name
                     , lastName :: Name
                     , age :: Int
                     }

-- Type Classes

class Equal a where
    same      :: a -> a -> Bool
    different :: a -> a -> Bool

instance Equal Card where
    same      (C x1 k1) (C x2 k2)      = x1 == x2 && k1 == k2
    different (C x1 k1) (C x2 k2)      = x1 /= x2 || k1 /= k2
```

Listing 2.5: Types Recap

2.3 Design Patterns

In the section of Main Concepts an important problem was introduced: *how to deal with side effects, or state-full computation for instance, in a functional context?*. The problem with effect-full functions, and in general with context computations, is that them often do not match the simpler ideas like referential transparency and them do not compose well. To solve this problem some patterns were added at type level using the concept of type classes. With those we can distinguish side effect and pure functions, compose effect-full and context-full functions. This is a core point in functional programming because if this topic remain unsolved then no truly useful functional computation can really happen. A program is useless if it is unable to perform side effects.

Them can be called *Design Patterns* because them are widely used, not only in functional programming, but are the foundation also for other libraries and frameworks.

This design patterns comes from the category theory and they have a precise hierarchy[73], they are build on top of each other in order, using the type classes concept, to gain more abstraction. For example, when a type

is declared as an Applicative, we know that this type is also a Functor. See figure 2.6 for some of those relationships stolen from Haskell base library.

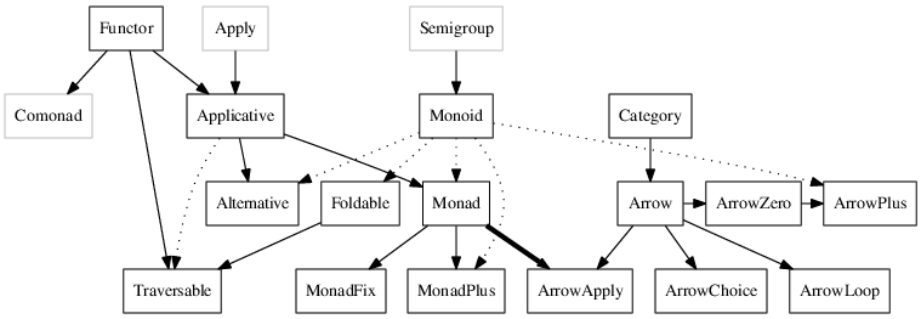


Figure 2.6: Design Patterns Hierarchy

In the following sections only the most relevant patterns for a typical programmer will be explained, and in particular how them are constructed internally.

Context-Full Computation Strategy

Before staring to dive into construction details, the strategy used to include context-full computation in a functional programming must be discussed in order to understand the actual goal of these patterns. The idea is to find a way to translate every context-full computation into something that is purely functional and in particular that is composable, at least with computation with the same type. Then, find if there is a specific pattern in order to abstract it and divide the common general part of it from the specific one, needed in that context.

First of all we have to figure out some of the main context-full computations that must to be in the language in order to make it useful:

Input/Output Without input and output a program cannot communicate with the outside world, so, even if that program can solve ideally every kind of problem, it becomes useless.

State Computation with states cannot be purely functional and them are core in most problems.

Error Handling A lot of computation can simply fail, so them sometimes will not return some value and something has to be done to manage the failure and mark the operation as critical.

Dependencies and Configurations Many times programs are based on the environment in which they are executed. Configuration and settings can completely change the behaviour of a system and most of the time this is done dynamically, on run-time.

Asynchronous and Concurrent Timing has a core role in countless situations. In addiction, nowadays almost all the CPUs are multi-core and systems are distributed by default. So asynchronous and concurrent computation is a must have property of a modern application to take advantage of current resources.

Indeterminism Every programmer can recall some situations where the current execution of a program is simply unpredictable. Those situations cannot be handled by pure functional programming due to its deterministic nature. Then, a strategy to make those computations explicit must be used.

At this point, what we want from the new abstraction is a way to hide and encapsulate the specific complexity of every problem listed above and provide to developers a simple way to manage those computations.

The most simple strategy known to all programmers is the sequential computation, where there is a simple control flow going from the top to the bottom. This approach will not suite well the above problems of course and this is actually the reason because the object oriented and imperative style struggle with them too. The object oriented paradigm introduced different

kind of principles and design pattern that are not directly related to the programming style, but them act as guidelines to avoid strange behaviours and manage the previous problems. For example, the state pattern helps dealing with state-full computation, but it is not built-in the language itself, but crafted by the developers and used by designer to modeling the problem. Then, the team must know them very well, and this is actually why them are often required by job interviewers during an hiring process.

The functional paradigm is not exception, so it also has its own design patterns, but them are a little more core in the programming style because without some of them the actual program becomes useless.

Thanks to function composition closures and higher order functions, the previous problems can be managed using pure functions shaping the actual patterns. Then, type classes comes into play and helps to separate the general part of the rising patterns to the specific ones and, in addiction, also several API related to those patterns can be constructed and generalized giving birth to something that can be called a framework. In fact, all those ideas are present, in the form of frameworks and libraries, to several programming languages, often implementing some of the most common specific implementation of the required pattern. Therefore, at the end we will have different specialized instances of this general abstractions, each one to manage a specific context computation.

Finally, the role of the type system here is core because it ensure the correctness of the composition and avoid bugs and run-time exceptions.

2.3.1 Functor

The first pattern showed is the Functor. It is often described to newcomers as a ‘container‘ of something with the ability to apply some transformation to the content of the container in order to get another container with a different content, but the same structure. Beyond the container analogy, we can think of a Functor as a way to give some kind of ‘computational context‘ to a specific data. In this terms it seems already complicated, but there are several examples of a Functor that are familiar to every programmer. The operation of every Functor is the *fmap*:

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

Listing 2.6: Functor Definition

Analyzing the fmap function and its signature in the listing 2.6, it is an higher order function where the first argument is a function, with the signature $(a \to b)$, that take a and return b and the second argument is the Functor f that contains a type a. The return of the fmap is a new Functor f containing b. As we can see from the type signature there is only one way to achieve the new Functor, and is the application of the input function to the content of the Functor and then wrap the result in the new Functor.

Examples

A list for example is the most common Functor. The list contain something of a particular type and we can apply a function to all the element of the list. This give back a new list, remember that all is immutable, with the results of every computation. When was said that the structure is maintained were meant that we cannot drop or add elements in this case using the operation provide by the Functor. This seems another limitation of functional programming, but it actually ensure consistency of the operation through out the transformation.

If we think of a list as a computational context we can say that the list type is used to express that there are zero or multiple elements of the inner type, and this is the actual context because with fmap we can apply pure function on single elements to the zero or more element context. The ‘fmap‘ for list is simply called ‘map‘, they are synonyms. In listing 2.7 there is the actual specific implementation of a Functor for the list case.

```haskell
instance Functor [] where
  fmap _ []     = []
  fmap f (x:xs) = f x : fmap f xs
```

Listing 2.7: List Functor Instance

```
> import Data.Char
> map intToDigit [1,2,3,4,5]
"12345"
```

Figure 2.7: List Fmap Example

In the example of figure 2.7 we can see a simple list of integer converted to a list of Chars (a.k.a String). The structure of the previous list is preserved,

the new list still has the same number of elements. The fmap operations can be chained in order to perform multiple transformations.

Providing another example, the Maybe type is a Functor. In fact the maybe wraps around a type and it can have two state: *Nothing*, saying that the maybe is empty, and *Just* x, saying that the maybe has a value of x. Executing the fmap operation of maybe functor, the given function is applied to the maybe's content, or, if the maybe is empty, then it remains empty. Then, also in this case the structure is maintained and the computational context of a maybe is a type with zero or one element, often used to model failures to avoid null exceptions. The same reasoning can be applied to all other Functors.

```
1  instance  Functor Maybe  where
2    fmap _ Nothing       = Nothing
3    fmap f (Just a)      = Just (f a)
```

Listing 2.8: Maybe Functor Instance

```
> import Data.Char
> fmap intToDigit Nothing
Nothing
> fmap intToDigit (Just 5)
Just '5'
```

Figure 2.8: Maybe Fmap Example

There are other influential example of functions like:

Either a b Represent a sum type of a and b, usually used to manage failures, but currying some extra information. An fmap execute the function f only on the right value of the either, otherwise the either will be returned unchanged.

```
1 data Either a b = Left a | Right b
2
3 instance Functor (Either e) where
4    fmap _ (Left a)  = Left a
5    fmap f (Right a) = Right (f a)
```

Listing 2.9: Either Definition and Functor Instance

```
> import Data.Either
> import Data.Char
> fmap intToDigit (Left "error")
Left "error"
> fmap intToDigit (Right 7)
Right '7'
```

Figure 2.9: Either Fmap Example

IO a Stand for a computation producing some value of type a alongside a side effect. When the fmap is applied then the computation is executed producing a result x and then return a new IO type wrapping around the result of the application of the fmap's input function to x, as expected. This type is very important because it is the actual way which the functional programs deals with side effects. In particular, if a function manage an IO type, that function is an effect-full function. This is a huge difference compared to other programs where the side effects can be performed anywhere in the program because it explicitly expose at type level those feature and then allow the type system to maintain the decoupling from the pure functions. Without this type a pure functional program is useless because it cannot perform side effect so is unreachable, we cannot push or pull any input or output from it.

In particular, in Haskell the IO type is the signature of the entry point of the program and it is the only type that has not a specific operation to get its internal value outside of an IO context: we can fetch the internal value of an IO only inside another effect-full function. This can be done thanks to another abstraction explained later in this chapter.

Here is showed that this type is a Functor, so the pure function can be applied to it, and it is very important in terms of composition, but is not enough because we also need to compose among the same types, for examples between IO types.

```haskell
instance Functor IO where
  fmap f x = x >>= (pure . f)
```

Listing 2.10: IO Functor Instance

```
> import Data.Char
> import GHC.Conc.Sync
> fmap intToDigit (getNumProcessors)
'2'
```

Figure 2.10: IO Fmap Example

In the figure 2.10 the effect-full function ‘getNumProcessors‘ is executed during the ‘fmap‘ and the result is extracted then passed to the input function of the ‘fmap‘. The result is another IO type with the number of processors as a String type, for simplicity the REPL directly prints the IO content. Moreover, watching the Functor’s instance for the IO type you can spot some alien operator and function like ‘pure‘ and ‘»=‘, them will be explained during the Applicative and Monad sections.

Functor Laws

At this point a designer can found very useful to implement some custom type as a Functor for some reason and to do that the only requirement seems to be: provide an implementation for the fmap function. This is obviously not enough because we do not know if the provided implementation of fmap also maintain the property of structure consistency. Then, the fmap has to obey to some specific laws. These laws are:

$$fmap\ id = id$$

The *id* function do not change the value of a type, then apply it to an fmap should not change the Functor in anyway and it is actually equal to apply directly the *id* function to the Functor, obtaining the Functor itself.

$$fmap\ (f \circ g) = fmap\ f\ \circ\ fmap\ g$$

If the function we pass to the fmap is a composition of two functions in a specific order then apply the function directly to the Functor or apply the composition function one after the other, in the same order, must result in the same value.

The origin of those laws come as well from category theory.

Lifting

The last way we can think of an fmap is the *lifting*. If we apply the currying to an fmap what we get is this signature $(a \to b) \to (f\ a \to f\ b)$. Then, the fmap can also be considered as a way to transform a pure function to a function within a Functor context, for example a function from Integer to Integer become a function from list of Integer to list of Integer.

2.3.2 Applicative

This type class adds another tool to the composability of functional programs. The Applicative[48] is based on the Functor type class, every Applicative is also a Functor, so the Applicative types has the fmap operation too. Besides the Functor definition, the Applicative(f) adds two new operations:

pure Signature: $a \to f\ a$. It is intended to upgrade a normal value to the Applicative context, build a default container for the given value.

apply Signature: $f(a \to b) \to f\ a \to f\ b$. As you can see the signature is exactly as the fmap one, but this time also the input function is wrapped inside the Applicative context. The intuition behind the Applicative operation is the application of a function inside a computational context.

The apply operation is very similar to the fmap, but with the apply the composition of function in the computational context is a little more easily, in particular because with the pure operation we can also build a chain with values in between. Usually for a given implementation of the apply there is only one possible implementation of pure.

```
1 class Functor f => Applicative f where
2     -- | Lift a value.
3     pure :: a -> f a
4
5     -- | Sequential application.
6     (<*>) :: f (a -> b) -> f a -> f b
```

Listing 2.11: Applicative Type Class

Applicative Laws

As well as in Functor, there are several laws for Applicative. Them ensure that the pure and Applicative implementations act as expected.

Identity Law $pure\ id\ <*>\ v = v$. If we lift the identity function and apply it to the value v then it must be equal to v.

Homomorphism $pure\ f\ <*>\ pure\ x = pure\ (f\ x)$. Apply a function and then lift its result to the Applicative context is the same that lift first the function and its argument to the context and then use the apply operation.

Interchange $u\ <*>\ pure\ y = pure\ (\$\ y) < * > u$. The order in which we evaluate the function and its argument does not matter. The symbol \$ means function application in a pure sense.

Associativity $u\ <*>\ (v\ <*>\ w) = pure\ (\ .\ )\ <*>\ u\ <*>\ v\ <*>\ w$. This law express the associativity: in the first case the second apply is executed first, in the second the opposite happen.

In addiction, there is another law that rule the relationship between the fmap and the apply of the Applicative. The law is:

$$fmap\ g\ x = pure\ g\ <*>\ x$$

What this law express is that the application of a pure function to an Applicative context value produce the same value as lift those function to the Applicative context and then apply it to the same value.

Examples

Most of the examples we saw in the Functor section are also valid for Applicative, for example the apply function for the *Maybe* type applies the fmap function if the input function is actually present otherwise it return Nothing. In this case also the input function can be *Nothing* or *Just f*, it is actually in the maybe context.

```
1  instance Applicative Maybe where
2      pure = Just
3
```

```haskell
    Just f <*> m = fmap f m
    Nothing <*> _ = Nothing

-- | Lift a function to actions. This function may be used as a value for
--   `fmap` in a `Functor` instance.
liftA :: Applicative f => (a -> b) -> f a -> f b
liftA f a = pure f <*> a
--   Caution: since this may be used
--   for `fmap`, we can't use the obvious definition of liftA = fmap.

-- | Lift a binary function to actions.
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = fmap f a <*> b
```

Listing 2.12: Maybe Applicative Instance and Lifting Primitives

```
> import Data.Char
> import Control.Applicative
> pure intToDigit <*> Nothing
Nothing
> pure intToDigit <*> Just 5
Just '5'
> pure intToDigit <*> ( pure (+) <*> Just 5 <*> Just 5 )
Just 'a'
> liftA intToDigit (liftA2 (+) (Just 5) (Just 5) )
Just 'a'
```

Figure 2.11: Maybe Applicative Example

In figure 2.11 there are the same exact computation of the figure 2.8, but using the apply. In addiction, there is also an example of composition of functions in the Applicative context. In the last line of the example, the concept of *lifting* introduced in section 2.3.1 for the Functor is reported using some primitives added in the listing 2.12. Here we can see a raw and first example of composition of context-full computation.

The same is valid for the list where the first input is a list of functions. The result would be another list where the elements of it are the application of every function of the first list with every input of the second list. Checkout the figure 2.12 for current example. Notice in listing 2.13 the usage of the list comprehension syntactic sugar, expressing the application of every function with every input.

```haskell
instance Applicative [] where
  pure x = [x]

  fs <*> xs = [f x | f <- fs, x <- xs]
```

Listing 2.13: List Applicative Instance

```
> import Data.Char
> import Control.Applicative
> pure intToDigit <*> [1,2,3,4,5]
"12345"
> pure intToDigit <*> (pure (mod 15) <*> (pure (^2) <*> [1,2,3,4,5] ))
"036ff"
```

Figure 2.12: List Applicative Example

Speaking of side effects and the IO context we can find out that it is an Applicative too. If we have an expression like $m1$ <*> $m2$ then the both arguments $m1$ and $m2$ are executed returning respectively a function f and a value x. Finally, the returning value of the whole expression would be a new IO type containing the result of f x. In figure 2.13 there is a simple example of and IO computation chain, in particular the 'getLine' is the actual effect-full computation, the others are only pure functions. Then, with Applicative we can chain pure functions in a context full computation, but we cannot chain two effect-full operation together.

```
> import Control.Applicative
> pure show <*> getLine
Hello World!!!
"\"Hello World!!!\""
> pure show <*> ( pure (\x -> "Greetings " ++ x)  <*> getLine )
Benkio
"\"Greetings Benkio\""
```

Figure 2.13: IO Applicative Example

With Applicative we showed how to apply a function in a specific context with a value in the same context and, thanks to the constraint where an Applicative is also a Functor, we can keep using all the benefit from the Functor typeclass we saw in the previous section.

What we have left is how to compose several context-full function together in the same context in order to get a bigger function.

2.3.3 Monad

The Monad is one of the most debated functional abstraction and can be explained from in fancy ways, like the famous burrito metaphor, or rigorously through the category theory with its definition:

> A Monad is just a monoid in the category of endoFunctors.

Here the Monad is introduced as a way to compose context-full computations trying to be as much clear as possible. With the Monad abstraction we finally reach the goal introduced in the beginning of this section.

The Monad is composed by two functions:

return $a \rightarrow m\ a$. This operation is the same operation we had encountered in the previous section(2.3.2), named pure. The name *return* can lead to ambiguity for developers coming from other imperative language because is a well known keyword. As previously, the return has the purpose to lift a value to the Monadic context.

bind $\gg\!=\ \ m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$. Looking at the definition does not seems that this operation allow us to compose two function in the Monad context together because it do not take as argument two functions as expected, but a value and a function as we saw for the apply operation. It is easy to prove that the input value is exactly the result of a previous context computation already applied. If we want a function composition that has two function as parameters and compose them we can refer to the >=> operator that has exactly that signature, but is defined in terms of bind. The >=> operator implementation in term of the bind one is: $f >=> g = \backslash x \text{ -> } f\ x \gg\!= g$

In addiction, With a signature like the bind we can argue that the same result can be computed also with the previous apply, but there is a little difference that made Monad more used and interesting then Applicative. In the Applicative, the input function must be a pure function wrapped inside the context, in order to do that, often a pure

function is lifted with the ‘pure‘ operation and then applyed, meanwhile here only the result of the input function is in the Monad context. This means that the input function is not a pure function, but a function that actual perform some context-full computation. Moreover, in the Applicative, we can also have an expression that result in a function as a first parameter, so it will first be evaluated and then the given argument will be applied. This is not the sequencing we want because, considering the value as coming from the past and the function as future computation, we want to execute previously the value and then the function. The definition of bind allow that.

In summary, bind allow us to: execute the computation in the expected order, decide if the computation must proceed based on the previous ones and chain them together simulating the imperative programming. Especially, with the usage of closures, we can reuse a previous result in future steps. Comparing this approach to an object oriented or imperative style, it is exactly like if the previous result is stored inside a constant in the computation scope and then reused later. What is provide in this case that is not possible in the imperative style is that the chaining of computation is not transparent like executing a statement after the other, but it is driven by previous results allowing more control.

```
1  class Applicative m => Monad m where
2    -- | Sequentially compose two actions, passing any value
     produced
3    -- by the first as an argument to the second.
4    (>>=) :: m a -> (a -> m b) -> m b
5
6    -- | Inject a value into the Monadic type.
7    return      :: a -> m a
8    return      = pure
```

Listing 2.14: Monad Definition

Now that we know what the bind operation is, then is easy to understand why the Monad abstraction is so important. With the Monad we can chain multiple computation together in a pure functional way.

At this point we can see how a Monadic computation is equivalent to a well known imperative one. In an imperative program we execute a statement after the other performing side effect and whatever computation we

like, but that computation can be refactoring as a chain of pure and effect-full functions that reuse previous values, through closures. Often, this idea is called 'the ambient Monad'. This is actually what is done in Haskell, as a syntactic sugar, to hide the function chain and provide to the developers an easy way to develop Monadic computation. There Is a big difference between an imperative program and a functional one with Monads: in an imperative program usually the types of the functions and statement are not so precise as in a functional program, like confine side effects, as a result is more easy to throw exception and run-time errors instead of compile time errors. Of course, the Monads are not a silver bullet to developing problems, be so explicitly in typing and using all this ideas requires often more reasoning and work also for simple stuff like a *printf* for example. Correctness comes with a cost.

In conclusion, remark that the Monad is not only used as a way to perform side effects, although it is the primary usage, but can be also used to design pure function computation and simulate other typical imperative programming behaviours. As a proof, a system designer can create its own custom Monad inside your project to manage special situations and instruct the type system.

Monad Laws

Left Identity $return\ x \gg\!= f\ =\ f\ x$ If we bind a function with a value x, properly lifted by the return, than the result is the same of the application of f acting on x.

Right Identity $m \gg\!= return\ =\ m$ The return bound to some value is equal to that value. Together with the previous law this law ensure the behaviour of the return function.

Associativity $(g >=> h) >=> k\ =\ g >=> (h >=> k)$ This law express the classical associativity law we saw previously too. This can be formulated also with the classic bind operation, but with the $>=>$ operator discussed in the beginning of the Monad section the idea of associativity appear more intuitive.

As for previous abstraction those laws are at the base of the Monad abstraction and derive from category theory. Them must be considered in

case of the building of a custom Monad in order to maintain the composition properties.

Examples

Obviously the most famous and important Monad is the IO Monad, but here we will speak also about some others equally important Monads that intend to simulate core imperative concepts. Every Monad type has different API and helpful functions to manage them at best.

1. The IO Monad

   It is the entry point of a pure functional program, taking Haskell as a reference, and it is mainly used to mark any indeterministic computation such as: randoms, concurrency, graphics, HTTP request and much more. When a function return type is an IO type this means that this function is not pure and its result can change every time it is called, also with same parameters. In this case the bind operation will execute the function, fetch the inner value and then pass it along to the next IO computation.

```haskell
module IOMonad where

import System.Random

generateAndPrintRandom :: IO ()
generateAndPrintRandom = (randomIO :: IO Int) >>= \r ->
    print r

generateAndPrintRandom2 :: IO ()
generateAndPrintRandom2 = do
  r <- (randomIO :: IO Int)
  print r
```

Listing 2.15: IO Monad Example

In the listing 2.15 there is a simple function that generate some random value and print it on standard output. In this example two effect-full and indeterministic functions are chained together with the bind operation. The functions ‘generateAndPrintRandom‘ and ‘generateAndPrintRandom2‘ do the same exact job, but in the second case the do notation was used to show how this syntactic sugar hides

a lot o boiler code as well as the application of bind, or other Monad primitives, in order to provide a more familiar imperative style, but remember that underneath all is implemented in terms of function composition.

Finally in the figure 2.14 there is some runs of the previous functions. The results are different at every execution as expected.

```
> generateAndPrintRandom
-2267795738622210889
> generateAndPrintRandom
-7462780386227065499
> generateAndPrintRandom
5691078975036211449
> generateAndPrintRandom2
1469964269743484687
> generateAndPrintRandom2
4229643549926998197
> generateAndPrintRandom2
8871010534753583822
```

Figure 2.14: IO Monad Example Execution

2. The Maybe Monad

   We saw the Maybe also in the examples of Functor and Applicative type-class, but it is a Monad as well. In fact, the return operation is the exactly same as the Applicative pure, and the bind operation is quite similar to the apply. In fact if the previous computation result in a 'Nothing', then the successor function will not be executed and the result of bind will still be 'Nothing', otherwise the value inside the maybe is passed to the following computation. This respect the Monad laws and allow to easily manage failures.

   An easy example of Maybe Monad can be found in the listing 2.17 alongside the equivalent do notation. Notice how in this case the flow of the computation, based on the existing value inside the maybe, is managed entirely by the bind operation.

```
1  instance Monad Maybe where
2      (Just x) >>= k = k x
3      Nothing  >>= _ = Nothing
4
5      return = Just
```

Listing 2.16: Maybe Monad Definition

```
1  module MaybeMonad where
2
3  isEven :: Int -> Maybe Int
4  isEven x = if (x `mod` 2 == 0) then Just x else Nothing
5
6  positive :: Int -> Maybe Int
7  positive x
8    | x > 0      = Just x
9    | otherwise  = Nothing
10
11 subtractionEvenAndPositive :: Int -> Int -> Maybe Int
12 subtractionEvenAndPositive x y = return (x - y) >>= \x1 ->
       isEven x1 >>= \x2 -> positive x2
13
14 subtractionEvenAndPositive2 :: Int -> Int -> Maybe Int
15 subtractionEvenAndPositive2 x y = do
16   r <- return (x - y)
17   r1 <- isEven r
18   positive r1
```

Listing 2.17: Maybe Monad Example

```
> subtractionEvenAndPositive 4 2
Just 2
> subtractionEvenAndPositive 0 2
Nothing
> subtractionEvenAndPositive 5 2
Nothing
> subtractionEvenAndPositive2 4 2
Just 2
> subtractionEvenAndPositive2 0 2
Nothing
> subtractionEvenAndPositive2 5 2
```

Figure 2.15: Maybe Monad Example Execution

3. The List Monad

   The same reasoning of the Maybe Monad is valid of the List Monad too. The return still be a simple creation of a new list with the passing parameter and the bind is the application of map with the input function and value followed by a join. In fact, in many others programming language the bind operation is synonym of 'flatmap' operation.

```haskell
instance Monad [] where
  xs >>= f      = [y | x <- xs, y <- f x]
  return x      = [x]
```

Listing 2.18: List Monad Definition

In the example 2.19 and the related execution we can see how, starting from a specific number the computation inside the Monad chain the function 'divisors' and 'tenMultipliers' getting the multiplication table of the divisors of the input number. For the body of the functions operating on list, the list comprehension is used, this is also a syntactic sugar for mapping operation provided by the language, Haskell or Scala for instance.

```haskell
module ListMonad where

divisors :: Int -> [Int]
divisors n = [x | x <- [1..(n-1)], n `rem` x == 0]

tenMultipliers :: Int -> [Int]
tenMultipliers n = [ f * n | f <- [1..10]]
```

```haskell

divisorsMultiplicationTable :: Int -> [Int]
divisorsMultiplicationTable n = divisors n >>= \x ->
    tenMultipliers x

divisorsMultiplicationTable2 :: Int -> [Int]
divisorsMultiplicationTable2 n = do
  d <- divisors n
  tenMultipliers d
```

Listing 2.19: List Monad Example

```
> divisorsMultiplicationTable 10
[1,2,3,4,5,6,7,8,9,10,2,4,6,8,10,12,14,16,18,20,5,10,15,20,25,30,35,40,45,50]
> divisorsMultiplicationTable2 10
[1,2,3,4,5,6,7,8,9,10,2,4,6,8,10,12,14,16,18,20,5,10,15,20,25,30,35,40,45,50]
> divisorsMultiplicationTable 13
[1,2,3,4,5,6,7,8,9,10]
> divisorsMultiplicationTable2 13
[1,2,3,4,5,6,7,8,9,10]
```

Figure 2.16: List Monad Example Execution

4. The State Monad

   We know that in functional programming all is immutable (2.2.3), so we require a way to perform state-full computations. The State Monad comes in rescue. The intuition is to store inside a Monad a function as a value with the signature $s \to (a, s)$, in particular that function is wrapped to a type called *State*. When this function will be executed with an input state it will provide a new value and a new state as a result. At each step of the state-full computation an intermediate state is passed alongside the actual value and, using specific functions, the developer can act on both state and value. Thanks to the Monad abstraction we can chain different state machines and use recursion. In addiction, the library that provide the State Monad also provide several operation to manage the state and value of the Monad.

   In Haskell the State Monad is defined in terms of its Monad transformer(2) StateT, but it can be hard to understand this Monad starting from this definition. Then, in the listing 2.20 there is a more simpler, but almost equivalent, definition of the Monad taken from the Haskell wiki [54].

```haskell
newtype State s a = State { runState :: s -> (a, s) } -- State type wrapping around the state function

instance Monad (State s) where
  return :: a -> State s a
  return x = state (\ st -> (x, st) )

  (>>=) :: State s a -> (a -> State s b) -> State s b
  pr >>= k = state $ \st ->
              let (x, st') = runState pr st -- Running the first processor on st.
              in runState (k x) st'         -- Running the second processor on st'.

-- Some tipical primitives to manage state, change it, exec it.

put newState = state $ \_ -> ((), newState)
get          = state $ \st -> (st, st)

evalState :: State s a -> s -> a
evalState pr st = fst (runState pr st)

execState :: State s a -> s -> s
execState pr st = snd (runState pr st)
```

Listing 2.20: State Monad Definition

The most simpler state machine is one with a single state where the input state comes in, some instant transformation happens and then the result in returned. This can also be done directly by a pure function, because without side effects the State Monad is a pure function, but here the purpose is to show a very basic usage.

```haskell
module StateMonad where

import Control.Monad.Trans.State

-- The signature says that the final value is Unit, but the State computed in S
-- In fact we don't have a value alongside the state.
addOne :: State Int ()
addOne = get >>= \v -> put (v+1)

addOne2 :: State Int ()
```

```haskell
addOne2 = do
  v <- get
  put (v + 1)
```

Listing 2.21: State Monad Example

```
> import Control.Monad.Trans.State
> execState addOne 2
3
> execState addOne2 2
3
```

Figure 2.17: State Monad Example Execution

In an imperative program often the state management is strictly bounded to some side effects in order to drive the state based of external sources. This is why the State Monad is implemented in terms of Monad transformers(2).

5. The Reader Monad

   Working with functional programming often we can turn up to a situation where the same parameter is passed along to several functions and if the number of those parameters grows than it can be a serious problem. For instance, when an application has some settings that effect the behaviour of it. To solve that the Reader Monad act as a way to share the same environment to different function. That get rid of those ugly parameters and configurations hiding them in the Monadic computation.

   The Reader Monad is more simpler than the State Monad because the value is a function with this simple signature $e \to a$. This means that the computation under the Reader Monad expect the environment e to be executed and when it happens return a value of type a. Notice in the definition of bind in the listing 2.22.

```haskell
newtype Reader e a = Reader { runReader :: (e -> a) }

instance Monad (Reader e) where
  return a = Reader $ \e -> a
  (Reader r) >>= f = Reader $ \e -> runReader (f (r e)) e

```

```
7
8     -- | Retrieves the Monad environment.
9     ask :: m r ask = reader id
10
11    -- | Executes a computation in a modified environment.
12    local :: (r -> r) ^
13    --  The function to modify the environment. -> m a -- ^
      @Reader@ to run in
14    --  the modified environment. -> m a
15
16    -- | Retrieves a function of the current environment.
17    reader :: (r -> a) -- ^ The selector function to apply
      to the environment.
18            -> m a
19    reader f = do
20      r <- ask
21      return (f r)
```

Listing 2.22: Reader Monad Definition

In the example code we can see how the string passed to the Reader Monad, when it is executed, is hiding inside the Monad and fetched when it is needed. Then, in the 'convo' function(listing 2.23), there is a simple way to combine multiple reader computation. Those computations could also be written directly inside the 'convo' function body nesting another layer of do notation. Finally, also a main is provided showing how to mix different Monads.

```
1  module ReaderMonad where
2
3  -- Idea Taken from the https://gist.github.com/egonSchiele
       /5752172
4
5  import Control.Monad.Reader
6
7  hello :: Reader String String
8  hello = do
9      name <- ask
10     return ("hello, " ++ name ++ "!")
11
12 hello2 :: Reader String String
13 hello2 = asks $ \name -> ("hello, " ++ name ++ "!")
14
15
```

```haskell
bye :: Reader String String
bye = do
    name <- ask
    return ("bye, " ++ name ++ "!")

bye2 :: Reader String String
bye2 = asks $ \name -> ("bye, " ++ name ++ "!")


convo :: Reader String String
convo = do
    c1 <- hello
    c2 <- bye
    return $ c1 ++ c2

convo5 = do
    c1 <- do    name <- ask
                return ("hello, " ++ name ++ "!")
    c2 <- bye
    return $ c1 ++ c2

convo2 = hello >>= \h -> (bye >>= \b -> return $ h ++ b) -- Using the bind
convo3 = hello >>= \h -> (\b -> h ++ b) <$> bye          -- Using the fmap
convo4 = asks (const (++)) <*> hello <*> bye             -- Using the apply

readerMain = print . runReader convo $ "adit"
```

Listing 2.23: Reader Monad Example

```
> readerMain
"hello, adit!bye, adit!"
```

Figure 2.18: Reader Monad Example Execution

6. The Writer Monad

The Writer Monad is dual to the Reader Monad. In the Reader Monad we have a common environment for the whole computation, in the Writer Monad instead we want to add some value to the given value. It is often used to logging or produce a result along side. What a

Monad wrap is a tuple of type (a, w) where the a is the type of the computation result and w the one of the "on the side" result.

In the listing 2.25 there is an example of the usage of the Writer Monad. As you can see, when the whole computation is executed the result would be the expected one plus an additional log showing every operation. Then, in the listing 2.24 there is the definition of the Writer Monad. It is based on the concept of Monoid, check out the section 2.3.4 to understand it well. Both example and definition come from the book "Learn You a Haskell for Great Good".

```
1
2 newtype Writer w a = Writer { runWriter :: (a, w) }
3
4 instance (Monoid w) => Monad (Writer w) where
5     return x = Writer (x, mempty)
6     (Writer (x,v)) >>= f = let (Writer (y, v')) = f x in
    Writer (y, v `mappend` v')
```

Listing 2.24: Writer Monad Definition

```
1
2 module WriterMonad where
3
4 -- From http://learnyouahaskell.com/for-a-few-monads-more
5 -- This example no longer works without tweaking - see
6 -- http://stackoverflow.com/questions/11684321/how-to-play-
    with-control-monad-writer-in-haskell
7 -- just replace the data constructor "Writer" with the
    function "writer" in the line marked "here"
8 -- That changed with mtl going from major version 1.* to
    2.*, shortly after LYAH and RWH were written
9
10 import Control.Monad.Writer
11
12 logNumber :: Int -> Writer [String] Int
13 logNumber x = writer (x, ["Got number: " ++ show x])  --
    here
14
15 -- or can use a do-block to do the same thing, and clearly
    separate the logging from the value
16 logNumber2 :: Int -> Writer [String] Int
17 logNumber2 x = do
18     tell ["Got number: " ++ show x]
19     return x
```

```haskell

multWithLog :: Writer [String] Int
multWithLog = do
    a <- logNumber 3
    b <- logNumber 5
    tell ["multiplying " ++ show a ++ " and " ++ show b ]
    return (a*b)

main :: IO ()
main = print $ runWriter multWithLog -- (15,["Got number: 3","Got number: 5","multiplying 3 and 5"])
```

Listing 2.25: Writer Monad Example

7. The Continuation Monad

   If you need the continuation passing style than this is the Monad for you. The continuation Monad enable to develop in this style in functional programming using a Monad structure. Then, the computation can be built from a function that start from an intermediate result to a final result. Chaining functions we can build up the whole computation and manipulate the flow at each step in a complex way, deciding to restarting or aborting the whole computation or only a specific portion of it for example.

   The continuation Monad embrace a function that require a continuation function and return the result of that function when it is executed with a proper input. Its definition and implementation is strictly related with the concept of currying (2.2.1).

   In the listing 2.26 there is the definition of the Continuation Monad showing how, in the bind the Monad wraps around the continuation function and combine different function in it. Then, in the listing 2.27 there is a simple usage of the Continuation Monad in the computation of the Pythagoras distance. Both example and definition comes from the Haskell wikibook.

```haskell
instance Monad (Cont r) where
    return x = cont ($ x)
    s >>= f  = cont $ \c -> runCont s $ \x -> runCont (f x) c
```

Listing 2.26: Continuation Monad Definition

```haskell
-- Using the Cont monad from the transformers package.
import Control.Monad.Trans.Cont

add_cont :: Int -> Int -> Cont r Int
add_cont x y = return (add x y)

square_cont :: Int -> Cont r Int
square_cont x = return (square x)

pythagoras_cont :: Int -> Int -> Cont r Int
pythagoras_cont x y = do
    x_squared <- square_cont x
    y_squared <- square_cont y
    add_cont x_squared y_squared
```

Listing 2.27: Continuation Monad Example

Combining Monads

Until now, we focus on combining functions together and how the type-classes can help in doing it. Moreover, these type-classes describe different contexts like: IO, failures, state and so on. Going a step further, we need to find a way to combine these context together as well because often we can end up with situations where we need to perform some state-full IO computations or handle failures inside a specific environment for instance. In following subsections there are a couple of solution to this need.

1. Nested Monads

   No one forbid us to nest a Monad inside another Monad. For instance, we can create a State Monad inside an IO function, run it and get the result wanted and then perform the side effect needed. This is the most simpler and fancy solution to the problem of combining Monads, but it can lead to unreadable code very quickly. It reminds the famous callback hell Javascript problem. Anyway, it still a good solution for simple situations.

   As an example of that check out the listing 2.23. It shows the nesting do notation of multiple Reader Monad computation in the function ‘convo5‘.

   Another interesting example of combining nested Monad is in the listing 2.28, here the function starts with and IO computation printing

a message and requiring the pressed button, then if that button is ‘x‘ the computation terminate returning Unit, otherwise a new IO computation starts performing some other effects and finally a recursion is fired restarting the computation from the beginning.

```
1
2 -- The details of the Subject Type, notify function and
    setSubject are
3 --   skipped for brevity You can check out the full code at
    :
4 --   https://github.com/benkio/ButtonLedSystemFP/tree/
    master/BLSHaskell
5
6  ledStateMachine' :: Subject IO a -> IO()
7  ledStateMachine' s = do
8    putStrLn "press enter to switch the led(ditig x + enter
     to exit)"
9    x <- getChar
10   if (x == 'x')
11     then return ()
12     else do
13       l' <- notify s
14       s' <- setSubject s (head l')
15       ledStateMachine' s'
```

Listing 2.28: Nesting IO Monads

2. Monad Transformers

What is usually preferred to Monad nesting are the Monad transformers. The idea is to build a new type-class with a type constructor parameterized over a Monad m, that provide a specific operation(lift) to perform computation of the inner Monad m. In the definition of the Monad transformer type-class 2.29 there is the signature of the lift operation that moves the inner Monad ‘m‘ to the Monad transformer ‘t‘.

```
1
2 class MonadTrans t where
3   -- | Lift a computation from the argument Monad to the
      constructed Monad.
4   lift :: (Monad m) => m a -> t m a
```

Listing 2.29: Monad Transformer Type Class

For example, if we want to compose the State Monad and the IO Monad we can use the StateT Monad passing the IO Monad as inner Monad and then perform IO operations inside StateT computation using the lift function. In the listing 2.30 we can see how the stateT is a Monad and it is very similar to the previous State Monad, but the return of the internal operation is wrapped in the input inner Monad. An example is provided(2.31) showing how to build the "guess number" game using the StateT Monad.

```
1
2  newtype StateT s m a = StateT { runStateT :: s -> m (a,s)
     }
3
4  instance (Monad m) => Monad (StateT s m) where
5      return a = StateT $ \ s -> return (a, s)
6
7      m >>= k  = StateT $ \ s -> do
8          ~(a, s') <- runStateT m s
9          runStateT (k a) s'
10
11 instance MonadTrans (StateT s) where
12     lift m = StateT $ \ s -> do
13         a <- m
14         return (a, s)
```

Listing 2.30: StateT Monad Definition

```
1 --
2 -- another example: a guessing game
3 -- (from http://scsibug.com/2006/11/28/a-simple-game-with-
    statet/)
4 --
5
6 module StateTMonad where
7 import System.Random
8 import Control.Monad.State
9
10 stateTMain :: IO ()
11 stateTMain = do answer <- getStdRandom (randomR (1,100)) --
     think of a number
12                putStrLn "I'm thinking of a number between
    1 and 100, can you guess it?"
13                guesses <- execStateT (guessSession answer)
     0
```

```
14                putStrLn $ "Success in " ++ (show guesses)
    ++ " tries."
15
16 guessSession :: Int -> StateT Int IO ()
17 guessSession answer =
18     do gs <- lift getLine     -- get guess from user
19        let g = read gs        -- convert to number
20        modify (+1)            -- increment number of guesses
21        case compare g answer of
22               LT -> do lift $ putStrLn "Too low"
23                        guessSession answer
24               GT -> do lift $ putStrLn "Too high"
25                        guessSession answer
26               EQ -> lift $ putStrLn "Got it!"
27
28 -------------------- 110 recognizer
    ---------------------------
29
30 data OneOneZeroState = S1 | S2 | S3 | S4
31
32 machineFunction :: OneOneZeroState -> IO ()
33 machineFunction S1 = putStrLn "0"
34 machineFunction S2 = putStrLn "0"
35 machineFunction S3 = putStrLn "0"
36 machineFunction S4 = putStrLn "1"
37
38 stateFunction :: Int -> OneOneZeroState -> OneOneZeroState
39 stateFunction x S1 = if x == 0 then S1  else S2
40 stateFunction x S2 = if x == 0 then S1  else S3
41 stateFunction x S3 = if x == 0 then S4  else S1
42 stateFunction x S4 = if x == 0 then S1  else S2
43
44 recognizer :: StateT OneOneZeroState IO Int
45 recognizer = lift getLine >>=
46   \input -> if (input == "x")
47               then return 0
48               else  modify (stateFunction (read input))
    >>
49                     get
    >>=
50                     \s -> lift (machineFunction s)
    >>
51                     recognizer
52
53 recognizerMain :: IO ()
```

```
54 recognizerMain = execStateT recognizer S1 >> return ()
```

Listing 2.31: Guess Number and 110 recognizer StateT Example

When we saw the State Monad in previous sections a simple definition was provided, but going to analyze the actual implementation of the State Monad inside the Haskell library we will find that this is actually defined using its Monad Transformer version. In those case, the inner monad would be the Identity Monad. The Identity Monad was not discussed previously because it simply wraps around a value performing nothing. The Identity Monad seems useless, but its origin comes again from the category theory.

Finally also the previous discussed Monads have their own transformer version, such as: ReaderT, WriterT, ContT, MaybeT and so on.

2.3.4 Monoid

In the formal definition of the Monad we found that a Monad is a Monoid. Then, in theory this section would be placed before the Monad one, but the Monoid idea is not strictly required to explain the Monad, then here we can see briefly what a Monoid is in order to have a little more understanding of the Monad.

Monoids [55] are a well known structure to everyone because there are many examples of Monoids in elementary math for instance. The intuition behind a Monoid is to find something of every kind that has particular properties regard some operation. Those properties are:

Associativity This is one of the most well known math properties and we found it also in previous Applicative Laws for instance (section 2.3.2).

Identity Element The analyzed type, in order to be a Monoid, must have an element that is an identity related to the correspondent operation. This seems to be very complicated, but with some examples all would be clear.

```
1 class Monoid a where
2   mempty  :: a
3   mappend :: a -> a -> a
4
```

```
5  mconcat :: [a] -> a
6  mconcat = foldr mappend mempty
```

Listing 2.32: Monoid Definition

In the Monoid definition 2.32 there are two main operation supported by the Monoid type-class, the first represent the Identity element of the Monoid and the second a way to chain two Monoid together, with the associativity property defined as law.

Moreover there is also an idea of concatenation of Monoid, deriving from the append operation. Sometimes, the concatenation idea can fit well the actual result, thinking of lists as a Monoid for instance, but in other cases it is more difficult to map the appending idea to the result of the operation.

Monoid Laws

The Monoid laws follow strictly the properties introduced in the starting of this chapter, those are:

Associativity `(x <> y) <> z = x <> (y <> z)`

where the '<>' symbol is the infix operator for the 'mappend' operation.

Left Identity `mempty <> x = x`

Right Identity `x <> mempty = x`

The right and left identity shows how the identity element is transparent to the append operation.

Examples

Going straight to examples, the most famous one is the relation between numbers and the sum and product operations. In fact, both operations supports their identity elements, zero and one, and the appending is the operation itself, because them has the associativity property.

Using Haskell to express formulaically this simple concept we get:

```
1 -- | Monoid under addition.
2 newtype Sum a = Sum { getSum :: a }
3
```

```haskell
-- | Monoid under multiplication.
newtype Product a = Product { getProduct :: a }

instance Num a => Monoid (Sum a) where
    mempty = Sum 0
    Sum x `mappend` Sum y = Sum (x + y)

instance Num a => Monoid (Product a) where
    mempty = Product 1
    Product x `mappend` Product y = Product (x * y)
```

Listing 2.33: Numbers as Monoids for Sum and Product

Here the demonstration of this in the GHCI interpreter:

```haskell

> import Data.Monoid
> Sum 5 <> Sum 6 <> Sum 10
Sum {getSum = 21}
> mconcat [Sum 5, Sum 6, Sum 10]
Sum {getSum = 21}
> getSum . mconcat . fmap Sum $ [5, 6, 10]
21
> getProduct . mconcat . fmap Product $ [5, 6, 10]
300
```

Listing 2.34: Sum Monoid Example

Another well known example is the List one, where the mempty element is the list constructor and the append is the list append.

```haskell

instance Monoid [a] where
  mempty  = []
  mappend = (++)
  mconcat xss = [x | xs <- xss, x <- xs]
```

Listing 2.35: Monoid List Definition

```haskell

> import Data.Monoid
> mappend "Hello " "World!!!"
"Hello World!!!"
> "Hello" `mappend` "World"
"HelloWorld"
```

Listing 2.36: List Monoid Example

Finally, there are also others kind of monoids from the everyday life, like the clock. In a clock, adding hours can be the Monoid operation where, the associativity property holds from the classical sum operation and the twelve number is considered the identity element.

2.4 Mutable State Solutions

Going back to the problem of state management, in the section about Monads and in particular with the State Monad; the classical way to solve this problem in functional programming was introduced. Anyway, sometimes it is not enough and a specific mutable state is required, in a state-full GUI that requires event handlers for instance. In these special cases, always taking Haskell as a reference, there are some ways to break the rule of immutability and have mutable state.

2.4.1 IORef

This is the simplest way in Haskell to create a mutable variable, it is just a box that contain a mutable variable with proper API for accessing and modifying the content. The usage of IORef is restricted to the IO Monad. The IORef can also be used for synchronization purposes through dedicated atomic operations, but in those case it is preferable other mechanisms like MVars or STM.

As always, in the listing 2.37 and 2.38 the definition of the IORef and its example can be found. In the example, a new IORef is created with a string inside it, then its content is changed through the right primitive and printed to show the value change.

```
1
2 -- IORefs
3
4 -- |A mutable variable in the 'IO' monad
5 newtype IORef a = IORef (STRef RealWorld a)
6
7 -- explicit instance because Haddock can't figure out a derived
    one
8 instance Eq (IORef a) where
9   IORef x == IORef y = x == y
10
```

```
-- |Build a new 'IORef'
newIORef    :: a -> IO (IORef a)
newIORef v = stToIO (newSTRef v) >>= \ var -> return (IORef var)

-- |Read the value of an 'IORef'
readIORef   :: IORef a -> IO a
readIORef  (IORef var) = stToIO (readSTRef var)

-- |Write a new value into an 'IORef'
writeIORef  :: IORef a -> a -> IO ()
writeIORef (IORef var) v = stToIO (writeSTRef var v)

atomicModifyIORef :: IORef a -> (a -> (a,b)) -> IO b
atomicModifyIORef (IORef (STRef r#)) f = IO $ \s ->
    atomicModifyMutVar# r# f s
```

Listing 2.37: IORef Definition

```
module IORef where

import Data.IORef

ioRefMain :: IO ()
ioRefMain = do
    -- create a new IORef
    stringRef <- newIORef $ ""
    string1 <- readIORef stringRef
    print string1

    -- change the value inside stringRef
    writeIORef stringRef "A"
    string2 <- readIORef stringRef
    print string2
    -- now it is apparent that the value inside of stringRef has
    changed
```

Listing 2.38: IORef Example

2.4.2 MVar

The MVar is very similar to the IORef, but it is specific for concurrent synchronization, so every modification to it is atomic. The MVar can be empty or full, with a value inside, and two operation of ‘take‘ and ‘put‘.

Both operations may block based on the current state of the MVar. The MVars ensure: fairness, laziness and access ordering.

The main primitives to create and modify an MVar are listed in the listing 2.39 with their signatures. This example is the same of the listing 2.38. The same thing done for the IO Ref can be easily rewrited in terms of MVar.

```
1
2 -- |Create an 'MVar' which is initially empty.
3 newEmptyMVar  :: IO (MVar a)
4
5 -- |Create an 'MVar' which contains the supplied value.
6 newMVar :: a -> IO (MVar a)
7
8 -- |Return the contents of the 'MVar'.  If the 'MVar' is
    currently
9 -- empty, 'takeMVar' will wait until it is full.  After a '
    takeMVar',
10 -- the 'MVar' is left empty.
11 takeMVar :: MVar a -> IO a
12
13 -- |Atomically read the contents of an 'MVar'.  If the 'MVar' is
14 -- currently empty, 'readMVar' will wait until its full.
15 -- 'readMVar' is guaranteed to receive the next 'putMVar'.
16 readMVar :: MVar a -> IO a
17
18 -- |Put a value into an 'MVar'.  If the 'MVar' is currently full
    ,
19 -- 'putMVar' will wait until it becomes empty.
20 putMVar  :: MVar a -> a -> IO ()
21
22 -- |Check whether a given 'MVar' is empty.
23 isEmptyMVar :: MVar a -> IO Bool
```

Listing 2.39: MVar Primitives Signature

2.4.3 State Thread Monad

The ST Monad has almost the same API of the IORef, but the advantage of using the ST Monad is the possibility to run computations also outside the IO Monad, using a mutable state. In particular, the ST Monad allow to encapsulate a computation, exactly like the IO Monad, and use mutable

memory location inside of it, then return a value back to the pure functional world and treat the ST computation like if it is actually pure.

The ST Monad is parameterized by: a type ‘a‘ that is the return value when the ‘runST‘ primitive is invoked and a type ‘s‘ representing the thread executing it. The type of the function used to escape ST is:

```
runST :: forall a. (forall s. ST s a) -> a
```

In the following example there is the implementation of the sum of a list using the ST Monad.

```
1
2  import Control.Monad.ST
3  import Data.STRef
4  import Control.Monad
5
6
7  sumST :: Num a => [a] -> a
8  sumST xs = runST $ do            -- runST takes out stateful code
       and makes it pure again.
9
10     n <- newSTRef 0              -- Create an STRef (place in
       memory to store values)
11
12     forM_ xs $ \x -> do          -- For each element of xs ..
13         modifySTRef n (+x)       -- add it to what we have in n.
14
15     readSTRef n                  -- read the value of n, and
       return it.
```

Listing 2.40: ST Monad Example

Chapter 3

Domain Specific Language: FPML

In previous chapters we analyze the main benefits of functional programming, how much it has influenced the main stream technologies and some of its adoptions in the industry. At this point, we want to discuss how we can design and model a problem with this programming style in mind. Watching at the object oriented paradigm and what we would do under those circumstances, we would typically try to: identify the main concepts and wrap them in objects, define their dependencies, how they interact, if they are simple or complex and at last we would formalize out thought using a modeling language, in most of the cases the UML. With a modeling language we can communicate efficiently to all the people who knows that language and even generate a large part of the project in some cases. In addiction, the UML for instance, embrace the meta-model of the object oriented paradigm, so the mapping to actual code become easy.

In functional programming there is not a standard modeling language that is widely recognized and adopted to design the problem solution, but in most cases the functional program itself can be directly used thanks to its conciseness, flexibility and support from the type system for correctness. In fact, we can design our types and function signatures, keeping in mind the most common design patterns introduced in section (2.3), and then reason on the actual composition of them using a top down or bottom up approach, divide big problems in sub problems and confine them in modules.

A true problem arise when we want ‘the best of both worlds‘ so we

are in a mixed context where we have functional and imperative/object oriented style code in our project. In those cases, we need a new design approach and tools that enables us to flawlessly integrate every style. If we decide to keep those style strictly separated we can still use both of the designing techniques in parallel and keep particular attention to the integration through specific API. For example, the functional and object oriented code can agree on some Interfaces for communications and then them can be developed each other independently. This idea can suite a micro-service architecture. If we pick Java and C# as technologies for the object oriented part, so our project is bound to a specific platform, the JVM or CLR in those cases, we can find different functional languages that runs on the same platform, like: Frege, Clojure and F#, as base for the functional section or take advantage of functional abstraction in java and C# we saw in previous chapters.

In case we want a close mix of paradigm instead of keep them separated, we need a modeling tool with:

- Fixed syntactic and semantic
- Unambiguous
- With the build-in abstractions from the chapter 2
- Extensible and easy to integrate with mainstream programming languages

This is what this chapter is about: try to build a simpler DSL language, called FPML(Functional Programming Modeling Language), that compiles in java with some of those features, check out the section 3.1. The result would be a tool that allow the design and build simpler algorithms with a specific architecture. Obviously, it will not be comprehensive of every possible computation done by a functional program, so this language can be a starting point for those who want to use a functional style generating all the code structure and providing a model of the system and than complete it.

Moreover, with a DSL we can export the functional abstractions to other technologies that do not have them from the start. The code that construct those concepts can be crafted in another paradigm and automatically generated when the model is defined.

3.1 Features

Software Architecture and Side Effect Separation

In order to explain some of the following features, an architecture for the DSL is required. Doing that require some reasoning about:

1. Context Separation: As we know, in functional programming all is a function, but we also saw that not all the functions compose in the same way depending on their nature: context-full or pure, so them will be separated.

2. Type System: Type system has an important role in system designing, so the DSL have to let the system designer declare its types to model the domain. Types can express effectfulness, or some computational context, marking the return type of functions, with the IO Type or Maybe Type for instance.

Driven by these thoughts, the new specific structure have to separate the side effects from the pure side of the program and, inside every part, some space is reserved for data and values declaration as well as functions. Values can be treated as functions that wants no arguments, so them could also simply be placed in the functions section, but keeping all divided is more clear and easy to read.

In the end there must be the entry point of our program. A program needs to perform some side effect to be useful, then the main function of our program needs to be a function with IO Type, exactly as in Haskell. See in figure 3.1 the actual empty code structure that reflect these considerations.

In the figure 3.1 there is also the ‘Undefined‘ keyword. This keyword stand for ‘unimplemented function‘ and is very useful in case we want generate the code from the DSL and then complete it. Thinking about what was said in the beginning of the chapter, this can be the actual place where a system designer planning for an integration with typical object oriented code for instance.

Basic Type System

A modeling language of a functional paradigm must have at last a basic type system because it carry a lot of benefits. This will helps us to increase

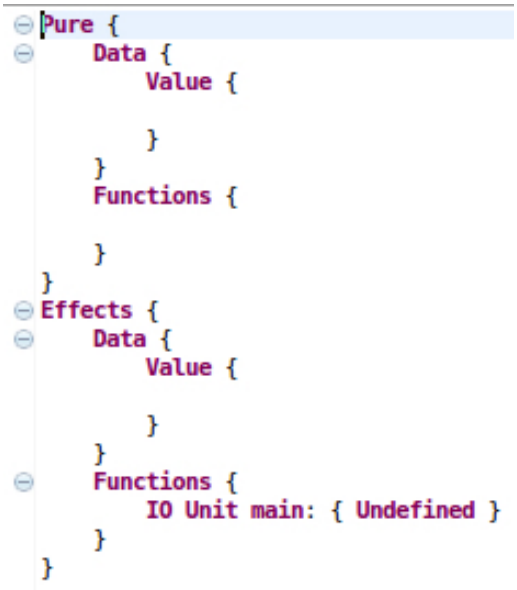


Figure 3.1: FPML Empty Code Structure

the correctness of the upcoming code, in particular we want the resulting code compile.

There are two groups of types: the pure types(Value Types) and the effect-full ones. Inspired by the IO Monad, to mark the effect-full function, the DSL has the IO Type, so all the types in the latter category will contain in some way the IO Type.

The value types are:

- Integer
- Boolean
- String
- Unit: it seems to be related to the ‘void‘ type of object oriented programming, but a pure function is not allowed to return Unit otherwise it will be useless due to referential transparency. Therefore was implemented a type check procedure that produce an error in these cases. Anyway, it is still needed because the IO Type would be a container, so the effect-full functions can return an ‘IO Unit‘ and perform some side effect alongside.

- Data: The system designer want to declare its type and use them in functions, so there must be a custom type with a proper ID and definition. There would be a data type for pure and effects contexts.

- Pure Function: In order to allow higher order functions and work with lambda expressions we need a way to type check and represent pure functions.

- Pure Algebraic: The same thoughts we did for data types applies to algebraic data types.

The effect-full value are:

- Effect-Full Function

- Effect-Full Data

- Effect-Full Algebraic

- IO: It is the type used to express side effects and in particular it wraps around another type, that would be the result of the effect-full computation. The inner type can be anyone listed in this section, but the Void type.

- Void: The previous type are well known at this point, but this one is a ghost type inserted to fail the type checking in special cases. It will never be used in an actual model.

Values and Expressions

In FPML, values are considered like functions with zero arguments, then them can be used everywhere a function can. Moreover, them has an ID for cross-referencing in other function bodies and in the end them are actually a way to reference its body expression.

An expression is the application of some functions and values that evaluates in a value of one of the types listed in section 3.1 and them can be used as well wherever a value can. Then, in order to be effective, the expressions have to allow a large variety of elements and with types like ADTs and IO Type them must be a recursive definition.

For brevity, here there is not the list of all possible values you can use in an expression, for detail check the 4 appendix.

Speaking of type checking, the type of the expressions and values are inferred by the system and in particular when a value is a data type its content is also checked for correctness. In addiction, you can place a pure value in the effect-full section of course, but a warning will rise to suggest a movement to the proper section. The goal of it is to guide the system designer to be consistent with the provided architecture.

Function Composition

In the chapter 2 we focused on the function composition, in FPML, it is the only way, except for the undefined keyword, to express function's bodies. As a result, the functions act as flow transformers: every step of the function body computes a new value that is passed to the next step. Then, in the end the function's body becomes streams of data and transformation of that data.

For function composition there are two different symbols stolen from Haskell and ML: the first is the `|>` forward pipe operator and the second is the `>>=` bind. In the chapter 2 the bind operator was defined as a Monad operator and here it is used to compose function in the effect-full realm. The forward pipe operator instead, is used in the pure context and simply apply the incoming result to the following function making the function application like a stream.

The function composition is possible because all the function are forced to have at least two argument. In particular, if a function has one argument is a simple function otherwise if the function has two arguments it is an higher order function that returns a function. The returning function of the higher order function is a function with the second argument of the outside function as a first argument and the result of the first function as a result. The fact that a function is an higher order function is also testified from the return type of the function.

With this mechanism we can actually perform any function with ‘n‘ arguments thanks to currying(2.2.1). What is missing is partial application, so designing a function that normally would take more then two arguments result in a model quite verbose.

There is a remarkable exception in the rules of "one or two arguments"

functions, the lambdas. Lambdas can have no arguments at all and in those cases they are treated as values because they are immediately executed. The only instance where a lambda expression without parameters is not executed in place is inside an If-then-else block, so we need to execute them after using an apply primitive. Check out the subsection 3.1 to learn about system primitives.

The bind operation works exactly as in Haskell, the the function on its left need to return an IO type and the function on the right must has an argument of the inner type of the previous IO type.

System Primitive

To ease the building of models and computation, the DSL has several primitives. Most of them are specific for Integer, String and Boolean types like: string concatenation, sum, multiplication, subtraction, equality, minor, major, and others. However, there are some primitives that deserve a little analysis:

Extract Enable to extract a value from a value of a custom data type. For example, if a function has in input a value of a custom data type and we want to get its internal value for the following computation chain, this is the primitive we needed.

If-Then-Else The if-then-else construct is a classic of every programming language and is core for computation based on some condition. In our case it allow to split the control flow. The main problem with this function is the return value because many times the types of the if branches are different. To overcome this problem the language has two different if-then-else form: the standard one that require the same type for both branches and the Either one (a.k.a. sum type), that return a sum type composed by both branches type. One side of the either would be empty and one full based on the if condition.

Lift We saw how the idea of lifting in the chapter 2, here this primitive has the same goal. If we have a pure function and we want to use it in effect-full context this primitive wraps that function to the IO context. Then, it can be used in a bind based chain.

Left/Right Algebraic If we have an algebraic data type like the Either or a tuple and we want to extract the left or right value this primitive allow that. The language cannot ensure that the data type has that value, the system designer can check with the proper primitive 'isLeft' or 'isRight'. If the primitive is used on an empty ADT, some exception could happen.

Return This is likely the return in a typical object oriented language and it is useful in case of the function needs only to preform a single computation because the forward pipe and bind operator always require two functions. This actually behaves like the return of the Monad type class, wrapping the incoming value from the function composition in the IO Monad.

Apply With higher order functions we can have a function as an input in the function chain. In those cases, we require a primitive that allow to apply that function with an arbitrary value and this is what the apply is for. This primitive require the type of the input function and an additional element, than apply that element to the function and return the proper result.

Often primitives needs some type annotation, this is because of type checking. It make it much more simpler, otherwise the type checking of the apply has to go back in the function chain and it would be very hard.

Every previous primitive has is dual in the effect-full realm, exception made for the lift one.

3.2 Language Design

This section will explore more the language design and reveal what is under the hood. The technology at the core of the language is the Xtext and Xtend framework[20]. The following subsections will briefly describe the actual grammar of the language with the details of the type system and code generation.

The language was strongly influenced by some of the language listed in chapter 1 in particular: Haskell, F# and Scala.

3.2.1 Grammar

The grammar follow the architecture idea described in section 3.1 and has, as first elements of the abstract syntax tree, the ones in the figure 3.1. In the following the remaining macro sections are:

Aggregate Rules These are some rules that group other rules together in order to easily refer to them in other rules or during the type checking. Them represent what is considered a function and the subdivision in pure and effect-full functions, also for the primitives.

Types Where all the types of the type system (3.1) are declared. A noticeable thing about those type organization is the IO Type that actually perform a recursion through the types wrapping around another type. In particular, the IO type is marked as an effect-full type, but inside it can encapsulate any other types. For the abstract data types the symbol '+' and '*' are used during the data declaration to recall the idea of algebra and sum/prod types. Instead, the type of a function recall the Java syntax, with its angle brackets, and also here there a separation between a pure function type and an effect-full one. The first must accept only pure types, the second instead accept any input types but the output type must be an IO type.

Expressions This section contains what is allowed as value body and functions in general. This section is one of the most complex because there are several case of recursion and wrapping in order to allow complex structures like abstract data type realization and lambdas. The out most rules are the *Expression* and *EffectFullExpression* that exactly match the *ValueType* and *EffectFullType* with the addiction of value references and the expansion of abstract data types into sum and prod types. Moreover, the effect-full expression has the ability to wrap in the IO type other expression and functions, we can think of it as the return function of the IO type and it is slightly different from the lifting function because in this case the result is not a function but a value.

Value, Data and Function declarations The actual syntax to declare each one of them. In each declaration we can find the reference to other rules. In Here there is what compose the body of these abstractions.

For example, if we check the *PureData* rule we found that its body is a *ValueType* rule.

Function Body How to create a function body and what are the related rules to the function's body. In the function body definition there is the composition operator introduced in section 2.2.2. A special attention must be given in the usage of some rules in the function bodies, if them are a reference or not. For example, the composition of functions are done by reference, so we need only to insert their names, but this do not holds for primitives or values.

The grammar is quite complex in some parts, but the best way to learn it is by example. However, the full grammar can be found as a reference in the appendix 4.

3.2.2 Validation

The validation and type checking has the same structure of the abstract syntax tree analyzing each element and implementing specific controls for each one. All the validation is done using the Xtend language[19]. In the following, there is a list of the files involved and what is their content:

FPMLValidator It is the entry point of the validation process and gather all the error messages and warning of the type checking. Thanks to the Xtext/Xtend platform we only need to add an annotation method '@Check' and the grammar rule type in input to that method then every time the model is saved all the elements of that type will be checked through that method. In our case, only few checks was added because almost all the rules are children of the function one, so checking the function rule directly enable the type checking to reach almost all the needed sub-rules. Inside of that check there is the delegation to the actual check based on the concrete type of the object. This was possible because of the switch of Xtend that allow pattern matching on types(3.2).

Checks This file regroup all the method that return a Boolean. Most of them are related to type checking, for example: the ones that given two types checks if they are equal, the type checking of data and values and the validation of function chain inside function's bodies.

```
36  @Check
37  def typeCheck(Function f){
38       switch f {
39           PureFunction: {
40               switch f {
41                   PureValue: typeCheckPureValue(f)
42                   PureLambda: typeCheckLambda(f)
43                   PrimitivePureFunction: typeCheckPurePrimitive(f)
44                   PureFunctionDefinition: typeCheckPureFunction(f)
45               }
46           }
47           EffectFullFunction: {
48               switch f {
49                   PrimitiveEffectFullFunction: typeCheckEffectFullPrimitive(f)
50                   EffectFullLambda: typeCheckEffectFullLambda(f)
51                   EffectFullValue: typeCheckEffectFullValue(f)
52                   EffectFullFunctionDefinition: typeCheckEffectFullFunction(f)
53               }
54           }
55       }
56  }
```

Figure 3.2: Validator of Function Type

GetArgTypes & GetReturnTypes The role of this two classes is straight forward, them provides methods to fetch the return type and the argument type of a specific rule. Also the content of them is organized based on the hierarchy of the grammar rules. Moreover, the organization of these files derive also from the reuse of the same code in the code generation.

Others This is the generic class that store methods with mainly two purposes: extracting the inner object due to left recursion grammar or from complex grammar rules in general and creating brand new object of a specific kind or wrap existing one inside an IO type or lambdas for instance.

Now focusing on the actual checking done in FPML we will go through all the grammar rules of the language and list all the checking in the language:

Pure Value For pure values whats matter is the type checking with custom data. Then, if a value is an expression but is not referring to a custom data then it will not type checked. In those cases its type will be inferred directly during the type checking of the function body where it is used. In particular, it will be simply replaced by its body.

However, if the value is referring to a custom data types the validator will find the type of the expression passed to the data type constructor and the type of the data type and then check if they are compatible.

Effectfull Value The effect-full values has the same control as the pure value, but for effect-full types of course. In addiction, there is the warning introduced in section 3.1 for the consistency of the architecture.

Pure Function Definition This is about the functions declared by the system designer and for them there are multiple type checks that implies the function type declaration. In particular, the check for the input type of the function and the return type. Both of them can navigate through the body of the function, once moving forward from the beginning of the stream and the other moving backward from the end, to figure out the actual type. Because of recursion and cycles in function calls the check on the return type of the function happens only when it is needed. In the cases, when a pure function is referenced and it is actual a function definition, the type checking simply trust the type definition without performing the checking and escaping the loop. The correctness of it will be checked by the function itself, otherwise the system will try to discover the type of the target function chain element. In the end, there are another simple check that ensure that the argument type is not an Unit type.

EffectFull Function Definition Effect-full Function Definition follows the exact same ideas of pure functions with addiction of the effect-full types and the absence of the unit check because it can be allowed in an effect-full context.

Pure Lambda We know that a pure lambda is an anonymous pure function, so the only check is about the correctness of the function chain in its body. Specifically, the same code of the function argument check is used and for the exception of the zero argument lambda, check out the section 2.2.2, the Unit type is used instead.

EffectFul Lambda As always it is the exact dual of the pure lambda check, but with the additional types.

Primitive Pure Function Most of the primitives has a fixed type signature, then a specific type check is not required, them are hard coded inside the type checker, but some of the primitives listed in section 3.1 needs type checking. In particular:

- IsLeft and IsRight can be only applied to sum types.
- The standard if branches must have the same types.
- The value passed to the apply primitive must match with the argument type of the input function from the function chain.

Primitive EffectFul Functions Here are performed the same check of primitive pure functions, but for the effect-full primitives.

Main Function This function is treated differently from the others because it has a fixed signature, so it reused the same checks for effect-full functions, but the expected return and argument type was hard coded.

EffectFul Data For the effect-full data the system search for the presence of IO Type and rise the warning previously described.

3.2.3 Code Generation

The code generation of the FPML is quite simple and maps all the elements in Java object or methods with the help of the FunctionalJava library[57] that already implements some of the abstractions introduced in the chapter 2. The FunctionalJava library has the IO type build-in, several types like the ‘Either‘ and ‘pairs‘ as well as a rich API to manage them. This made the code generation a lot easier and in addiction it can be useful as an example of how this library can be used inside a project. The Functional Java library is not the only library that export some of the functional concepts in Java, for example there are the BGGA project[5] or JavaSlang[40], but it seems to be the most mature and used. Anyway, this is another proof of how much functional style is becoming important in the software field.

In order to simulate the idea of independent functions in Java the DSL use static methods, so every function and value is encoded like that. This decouple the function from the object and it is not bound to a object life

cycle. The only requirement is to import the right package, and enable the scope of the needed function.

Static Code Generation

The simpler and static code generation is the one for the primitives. Them are hard coded into a specific classes, one for pure(Primitives.java) and one for effect-full(PrimitivesEffectFull.java). Every time the model is saved those class are generated and every time a primitive is used the specific static method is called. The packages organization is also fixed and based on previous experiments in functional programming and considerations illustrated in section 3.1. There are four packages:

EffectFull Contains the primitives effect-full class, the entry point of the program with the main and the effect-full function defined in the model.

EffectFull.Data Contains: The class Value.java with all the static methods related to effect-full values defined in the model, a class for every custom data type and the interface bounded to every custom data type. That generic interface is statically generated as well and is very useful when we need to extract the value from a custom data type. With every custom data type implementing that interface we know that each one of them obey to a specific API, in particular the interfaces for data types has only a getter to retrieve the inner value. Then, the implementation of extract primitives for instance becomes very simple.

Pure Has the exact content of the Effect-full package, but for pure functions. Obviously, the pure package do not contain the entry point.

Pure.Data Has the exact content of the Effectful.Data package, but for pure data and Values.

In addiction to Primitives and packages also in the main function the computation starts with the IO Unit value, this because an empty main imply some IO operation itself wrapped around an Unit value, so this is why this main's starting code is hard coded and static.

The last part of the static code generation is the skeleton of every class and interface in the code generation: in every file, Java require the right package name as well as imports and class declaration. At every code generation them are hard coded.

Dynamic Code Generation

The dynamic side of code generation can be divided to: functions, data and values.

Starting from the values, them are pretty simple. Values are like functions with no arguments, so them are encoded exactly like that in the EffectFullValues.java and PureValue.java files. When the code is generated each value in the model is mapped in a static method with no arguments, the same name of the value and the proper return type. Then, the body of the value is dynamically generated using the expression rule code generation method where the body content type is checked and a specific code is generated based on it.

For data types, a Java class per type is generated with a private constant field with the proper type, a constructor and the getter from the data type interface. In the code generation of these types you can see how actually the algebraic data types are mapped to Either and Pair types.

Looking at the code generation of functions it seems exactly like the values one, but this time them has one argument. What is interesting is the way function compose based on their nature, pure and effect-full: in pure functions each element of the body's function chain is wrapped one inside the other in reverse order, with special exception for lambdas, and in the effect-full realm the API from FunctionalJava library was used to compose them using mostly the bind and map operations.

When in the model a lambda is used a specific generic type from Functional Java called 'F' is generated. Its purpose is to simulate a function object in order to treat it as a first class citizen. This allow, at the creation of the object, to specify the body of the lambda and then execute it with a specific method 'f'. It turns out very useful for the generation of higher order too because we can build a deeper nesting of F objects when we specify the arguments and return types of F in its generic type parameters.

3.3 Examples

Until now we saw how the FPML works, what are the concepts inside it and the implementation of them. In this section, some simple examples will be exposed to show in practice what is the output of the language and how especially encode the architecture and function composition.

Hello World and Greetings

The 'Hello World' and 'Greetings' are the first examples in every languages and here is not exception.

In the Java's Hello World we only need a simple line of code inside the main method, but like this we do not decouple the pure data from the part that perform the input-output. The FPML instead make this clear separation. Moreover, notice how also in a functional language like Haskell no one force the developer to keep this separation, all is left to the developer itself. Therefore, with FPML we have done a step further because, defining a model through this DSL the upcoming project becomes consistent with a specific architecture idea. The model of the 'Hello World' is:

```
Pure {
    Data {
        Value {
            HelloWorldMessage: "hello World!!!"
        }
    }
    Functions {

    }
}
Effects {
    Data {
        Value {

        }
    }
    Functions {
        IO Unit main: { IO(PureRef(HelloWorldMessage)) >>= print }
    }
}
```

Figure 3.3: FPML Hello World's Model

As you can see the structure is almost blank as the figure 3.1, but the pure data for the string and the body of the main. Generating the code from this model we get the main method implemented using the primitive ‘print‘ in the function body and the value ‘HelloWorldMessage‘.

```
public class PureValue{

    public static String HelloWorldMessage() {
        return "hello World!!!";
    }
}
```

Figure 3.4: FPML Hello World’s Generated Value

```
public static void main(String[] args){
    IOW.lift(IOFunctions.ioUnit).append(IOFunctions.unit(PureValue.HelloWorldMessage()))
        .bind(PrimitivesEffectFull::primitivePrint)
    .safe().run().on((IOException e) -> { e.printStackTrace(); return Unit.unit(); });
}
```

Figure 3.5: FPML Hello World’s Generated Main

Analyzing the code generated for the main function we can see the IO Unit start code, and then the ‘HelloWorldMessage‘ is appended to the flow, after the lifting of the pure value to the IO type. Finally, the value is passed using the bind operation to the ‘primitivePrint‘ that actually perform the side effect. Pay serious attention, at this point nothing is truly executed, only the IO computation is built, all the computation is performed during the last line when the ‘run‘ method is called. We can see here how the lazy evaluation is implemented and how we can, skip the run code line and move the computation around if we want. Notice how, in this base example, there are a lot of code noise. This is an effect of porting the functional style to a technology based on another paradigm. Moreover, a lot of implementation details of the IO Type and its API are hiding behind the FunctionalJava library. In the following examples we will see how things get worse when complexity increases.

Moving to the greetings example, we have the same structure of the hello world example but this time we have to get the name to greet from standard in, To do that there is an effect-full primitive called ‘getLineStdIn‘, and create a specific function that concatenate the fixed greetings message with the input.

The main implementation will be skipped because there is nothing more than what was showed in the the previous example. What it is interesting is the additional function for concatenation. In particular, form the model you can see how the ‘+‘ function that actually concatenate the two string together is an higher order function so the result immediately after that chain’s step is a function that require a string and return a string. In this case a way to apply that is needed, so here the ‘Apply‘ primitive is used, passing in the argument of the function. Finally, here there is an actual example of type annotation noise introduced previously: you find in the model a lot of types also in the body of the function, this because it simplify a lot the type checking instead of type inference all type along the function chains.

This is the generated function from the model that does the concatenation and it is a pure function indeed. The plus primitive return a ‘F‘ type and the apply primitive in this case consist in the execution of the returned function with the outer function’s parameter.

```
Pure {
    Data {
        Value {
            AskMessage: "Hello, who you are?"
            GreetingsMessage: "Greetings, "
        }
    }
    Functions {
        def String greetingsConcat(String name): {
            GreetingsMessage |>
            + String         |>
            applyF F<String,String> name
        }
    }
}
Effects {
    Data {
        Value {

        }
    }
    Functions {
        IO Unit main: {
            IO(PureRef(AskMessage)) >>=
            print                   >>=
            getLineStdIn            >>=
            Lift(greetingsConcat)   >>=
            print
        }
    }
}
```

Figure 3.6: FPML Greeting's Model

```
public static String greetingsConcat (String name ){
    return Primitives.plus(PureValue.GreetingsMessage()).f(name);
}
```

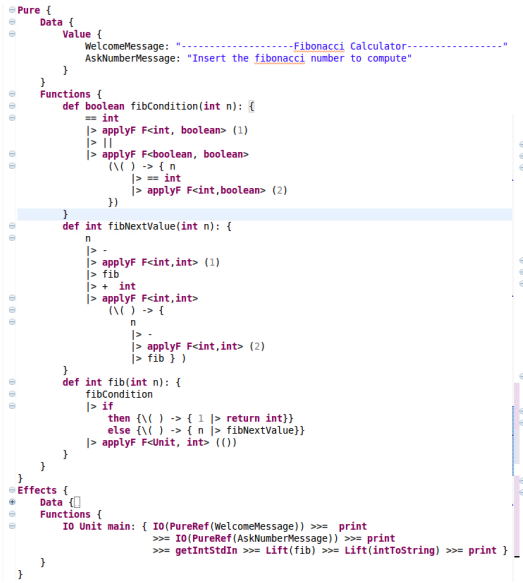
Figure 3.7: FPML Greetings's Generated Concat Function

Fibonacci and Button-Led

In these two examples all becomes unfortunately more complex and verbose, but it is useful looking at them to analyze some weaknesses of the language as well as how the chosen architecture evolves.

Looking to the Fibonacci solution in Functional style for example, we require to build a recursive function with a way to express a base case to exit the recursion and returning the result. In Haskell there is the concept of guards and pattern matching that helps a lot in the conciseness of the code, but the same can be achieve with a classical if expression. Unfortunately FPML lack the guards concept, so the if primitive comes in rescue. Moreover, in Fibonacci's model we have a lot of higher order function to manipulate the integers, adding and subtracting, as well as dealing with checks for the base condition. This make the model a lot more extended and is one of the possible future works of the language. In the figure 3.8a you can see the actual Fibonacci's model.

In figure 3.8b there is also the generated code for the custom pure function for Fibonacci's model. In our model there are a lot of lambdas due to the needs of apply and if primitive, then the generate code has a bunch of anonymous 'F' types. For generation purposes was preferred to be a little more explicit using the old Java 7 anonymous objects because give me more controls on the types.



```
Pure {
    Data {
        Value {
            WelcomeMessage: "--------------------Fibonacci Calculator----------------"
            AskNumberMessage: "Insert the fibonacci number to compute"
        }
    }
    Functions {
        def boolean fibCondition(int n): {
            == int
            |> applyF F<int, boolean> (1)
            |> ||
            |> applyF F<boolean, boolean>
                (\( ) -> { n
                    |> == int
                    |> applyF F<int,boolean> (2)
                })
        }
        def int fibNextValue(int n): {
            n
            |> -
            |> applyF F<int,int> (1)
            |> fib
            |> +  int
            |> applyF F<int,int>
                (\( ) -> {
                    n
                    |> -
                    |> applyF F<int,int> (2)
                    |> fib } )
        }
        def int fib(int n): {
            fibCondition
            |> if
                then {\( ) -> { 1 |> return int}}
                else {\( ) -> { n |> fibNextValue}}
            |> applyF F<Unit, int> (())
        }
    }
}
Effects {
    Data {}
    Functions {
        IO Unit main: { IO(PureRef(WelcomeMessage)) >>=  print
                        >>= IO(PureRef(AskNumberMessage)) >>= print
                        >>= getIntStdIn >>= Lift(fib) >>= Lift(intToString) >>= print }
    }
}
```

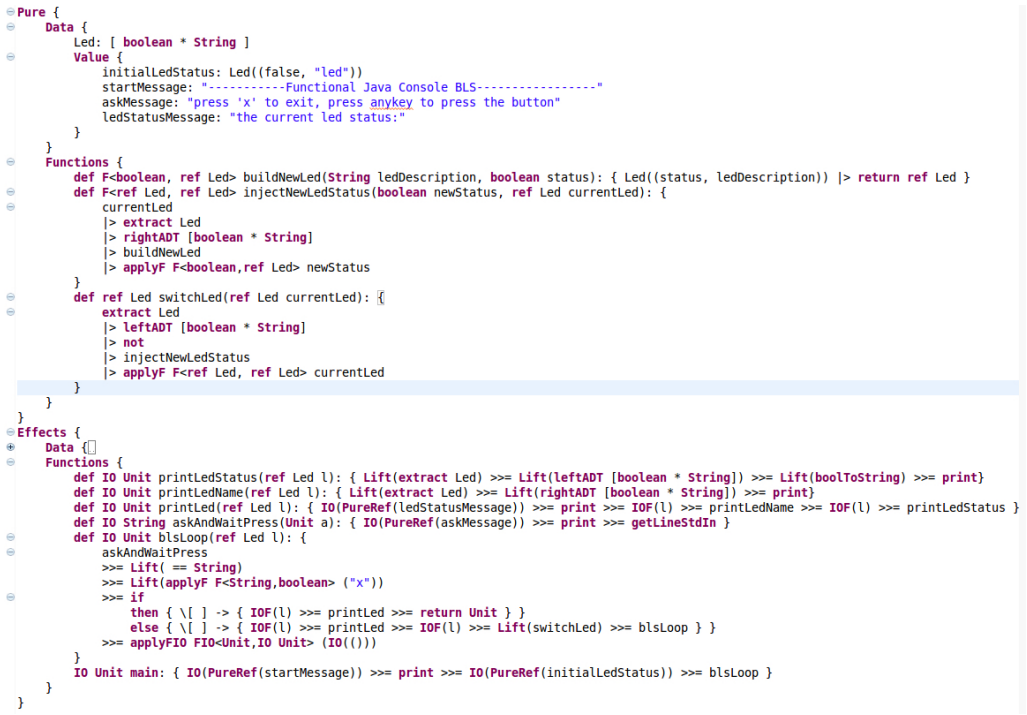
(a) FPML Fibonacci's Model

```
public class PureFunctionDefinitions {

    public static Boolean fibCondition (Integer n ){
        return Primitives.logicOr(Primitives.equalsCurrying(n).f(1)).f(new F0<Boolean>() {
                @Override
                public Boolean f() {
                    return Primitives.equalsCurrying(n).f(2);
                }
            }.f()
        );
    }

    public static Integer fibNextValue (Integer n ){
        return Primitives.plus(fib(Primitives.minus(n).f(1))).f(new F0<Integer>() {
                @Override
                public Integer f() {
                    return PureFunctionDefinitions.fib(Primitives.minus(n).f(2));
                }
            }.f()
        );
    }

    public static Integer fib (Integer n ){
        return Primitives.<F0<Integer>>pureIf(
                fibCondition(n),
                new F0<Integer>() {
                    @Override
                    public Integer f() {
                        return 1;
                    }
                }
                , new F0<Integer>() {
                    @Override
                    public Integer f() {
                        return PureFunctionDefinitions.fibNextValue(n);
                    }
                }
                ).f();
    }
}
```

(b) FPML Fibonacci's Pure Functions

What we did not see yet is a custom data type. To show an example with that concept we can see the Button-Led example. This is a very simple example that imply a button and a led. When the button is pressed the led turn its state to on or off depending on its previous state. The typical functional implementation of this type of problem would use the State Monad(section 4), but the FPML does not implement it, the only Monad abstraction in the language right now is the IO one. Therefore, the problem can be solved using recursion functions. In this case, the custom data type is the led that is an product type formed by a Boolean, representing its status, and a String, for its name. What we need in terms of pure functions is a function that switch the led status. Moreover, every data is immutable(2.2.3), so, to actually change the state of the led, we have to build a new led from the previous one, but with a twisted status. Finally, for the side effect part of the solution, we need some function to: print the current state of the led, ask the used if he wants to terminate the program or turn the led status and restart all the computation if the led status is changed. See the resulting FPML model in figure 3.8.

The model of figure 3.8 is very interesting and show a lot of details and features of the DSL. Starting from the top there is the actual custom data



```
Pure {
    Data {
        Led: [ boolean * String ]
        Value {
            initialLedStatus: Led((false, "led"))
            startMessage: "----------Functional Java Console BLS----------------"
            askMessage: "press 'x' to exit, press anykey to press the button"
            ledStatusMessage: "the current led status:"
        }
    }
    Functions {
        def F<boolean, ref Led> buildNewLed(String ledDescription, boolean status): { Led((status, ledDescription)) |> return ref Led }
        def F<ref Led, ref Led> injectNewLedStatus(boolean newStatus, ref Led currentLed): {
            currentLed
            |> extract Led
            |> rightADT [boolean * String]
            |> buildNewLed
            |> applyF F<boolean,ref Led> newStatus
        }
        def ref Led switchLed(ref Led currentLed): {
            extract Led
            |> leftADT [boolean * String]
            |> not
            |> injectNewLedStatus
            |> applyF F<ref Led, ref Led> currentLed
        }
    }
}
Effects {
    Data {
    Functions {
        def IO Unit printLedStatus(ref Led l): { Lift(extract Led) >>= Lift(leftADT [boolean * String]) >>= Lift(boolToString) >>= print}
        def IO Unit printLedName(ref Led l): { Lift(extract Led) >>= Lift(rightADT [boolean * String]) >>= print}
        def IO Unit printLed(ref Led l): { IO(PureRef(ledStatusMessage)) >>= print >>= IOF(l) >>= printLedName >>= IOF(l) >>= printLedStatus }
        def IO String askAndWaitPress(Unit a): { IO(PureRef(askMessage)) >>= print >>= getLineStdIn }
        def IO Unit blsLoop(ref Led l): {
            askAndWaitPress
            >>= Lift( == String)
            >>= Lift(applyF F<String,boolean> ("x"))
            >>= if
                then { \[ ] -> { IOF(l) >>= printLed >>= return Unit } }
                else { \[ ] -> { IOF(l) >>= printLed >>= IOF(l) >>= Lift(switchLed) >>= blsLoop } }
            >>= applyFIO FIO<Unit,IO Unit> (IO(()))
        }
        IO Unit main: { IO(PureRef(startMessage)) >>= print >>= IO(PureRef(initialLedStatus)) >>= blsLoop }
    }
}
```

Figure 3.8: FPML Button-Led's model

type for the led and in pure value section of the model the initial led. It represent the state of the led at the beginning of the program's execution. Then, analyzing the functions we see the 'buildNewLed' function that is an higher order function because of its return type and the double arguments. Moreover, This function show how to build a new data value from scratch using the same name of the custom data type and providing a proper value that fit the inner type declaration. In addiction, what is new in those function is the usage of the 'ref' keyword to refer to the new custom data type and the primitives to extract the needed value from the algebraic data type. Finally, what is noticeable from the effect-full functions is the usage of primitives for lifting functions and values to the IO context through the 'Lift', 'IO' and 'IOF' keywords.

The very important thing to analyze at this point is the actual code generated from the DSL, in particular the effect-full code. In the code listing 3.1 there is the generated code for the 'bslLoop' function. Is very difficult to figure out what this function does and especially would be quite impossible for every programmer to write a function like that from scratch. Keep in mind that this still be a very simple example of a program.

```
1  public static IO<Unit> blsLoop (Led l){
2      return IOFunctions.bind(
3          IOFunctions.bind(
4              IOFunctions.unit(
5                  IOFunctions.runSafe(
6                      IOFunctions.map(
7                          IOFunctions.bind(
8                              IOFunctions.unit(l),
9                              ignored -> EffectFullFunctionDefinitions.askAndWaitPress(
      Unit.unit()))
10                         , Primitives::equalsCurrying)
11                     ).f("x"))
12             , (Boolean c) ->  IOFunctions.unit(PrimitivesEffectFull.<F0<IO<Unit>>>
      effectFullIf(
13                 c,
14                 new F0<IO<Unit>>() {
15                     @Override
16                     public IO<Unit> f() {
17                       return IOFunctions.bind(IOFunctions.unit(l)
18                         , EffectFullFunctionDefinitions::printLed)
19                       ;
20                     }
21                 }
22                  , new F0<IO<Unit>>() {
23                     @Override
24                     public IO<Unit> f() {
25                       return IOFunctions.bind(
26                           IOFunctions.map(
27                               IOFunctions.left(
28                                   IOFunctions.unit(l),
29                                   IOFunctions.bind(
30                                       IOFunctions.unit(l)
31                                       , EffectFullFunctionDefinitions::printLed)
32                                   )
33                               , PureFunctionDefinitions::switchLed)
34                         , EffectFullFunctionDefinitions::blsLoop);
35                     }
36                   })
37                 )
```

```
38                 )
39         , (F0<IO<Unit>> f) -> f.f());
40   }
```

Listing 3.1: Button Led's Loop Function

3.4 Future Works

Previous sections shows the actual state of the language and a couple of simpler examples to begin with. In this section there is a simple list and discussion of possible future works of this project.

Missing Functional Design Patterns

In section 2.3 a lot of different abstractions were introduced, discussing where they are useful and why. In particular, in a typical programming language a designer can build them up respecting the proper laws and definition. This cannot be done in the DSL, but, as a future work, them can be directly embedded inside the grammar and let the platform generate all the infrastructure and the proper API.

The first idea can be to add the most common Functors, Applicative and Monad instances directly in order to enable computations in different contexts. Then, the next step could be allowing the system designer to add custom instances with a generated test plan for each of them based on the laws.

At the moment, in the DSL there is the IO type that simply try to mirror the way in which the IO Monad works and is used to divide the pure functions from the effect-full ones.

Generics and Type Inference

The type system of the FPML is very basic and so it did not comprehend generics. A possible evolution can be the addiction of generics as well as type inference to ease the usage and the verbosity of the language itself.

In addiction, the presence of generics will enable the polymorphic functions and ADTs. As an effect there will be much more flexibility and reuse of the existing code. Moreover, the following code generation size can shrink considerably.

OOP and Imperative Abstractions Integration

At the moment the FPML enable the OOP integration with an existing system through the ‘undefined‘ word that leave a place in which the designer can place whatever he wants to perform a computation. In the future the DSL can evolve embracing the OOP abstractions in a more deeper way.

An idea can be a sort of configuration file where the API, with related signatures, of an external system are specified and that can be used directly inside the modeling language as normal primitives. This can turn useful also in case the system is distributed: let's consider a system composed by micro-services for instance.

Actor Model

The FPML born like a way to export the functional style across different environments, but the interaction model at the core of functional programming still be the function call and it does not suite a distributed context.

To solve this problem, the FPML can integrate also with another model like the actor one to gain another interaction model that is more correct with a distributed scenario. Moreover, the actor model can also provide a way to manage concurrent computations.

Asynchronous Programming

Another way in which the interaction model of the FPML can be extended is adding the possibility to perform asynchronous calls. This can be done adding the most common ‘async‘-‘await‘ primitives and concepts like Promises and Futures. Furthermore, promises are also often related to the concept of Monad as well, so this fine to consider the addiction of promises directly as an additional Monad.

Function n-arity and Partial Application

A functional feature left out from the modeling language is the possibility of using function with more than one parameter enabling the n-arity and partial application. This result in a more verbose language that, in complex cases, can lead to a very confused modeling. The extension of the language with this simple characteristic can enhance a lot the readability of the outcoming model itself, making it also more easy to modify.

3.5 Source Code

The source code of the FPML project can be found at the following link:
`https://github.com/benkio/FPML`

Chapter 4

Conclusions

This thesis, in the first place, showed the adoption degree of functional programming in the industry presenting several successful use cases and where this style influenced other technologies. Then, the main question was presented: what are the main ideas, benefits, drawbacks of the paradigm and what means design a system using these abstraction. Here, the goal was to introduce this approach to system designers and teach them the basics. Finally, a proposal was made to formalize the main abstractions in a simple tool that enable to export them to multiple environments.

The main contribution of this thesis was to show how FP paradigm is not anymore an option, but a need. This because of the pervasiveness of functional abstractions and the growing complexity of problems that developers have to face. Nowadays, a system is by default distributed and complex, with countless components interacting with each others. In these scenarios the Imperative and Object Oriented Paradigms needs some contamination from other models. For this reason frameworks and libraries are used to fill this abstraction gap. Different kinds of architectures arises to solve those problems and avoid monolithic solutions, like *micro services*. In particular, a system needs, more than ever, to be able to flawlessly compose and integrate multiple components together and separate computations by context other than by responsibility. This means to have a way to rigorously divide operations by their result, context and effect. It's also important to have this separation reflected into code: have a way, looking into the code, to identify if an operation can fail, provide multiple results or perform some side effects for instance. As a result: code reasoning, testing, refactoring

and bug fixing becomes more easily. Furthermore, with hardware's increasing computational power, a functional approach can be exploited in a very efficient way.

We saw in Chapter 2 how the functional style addresses these problems and embrace the needed properties. These are the main reasons at the basis of the functional growth in the industry too. It is also remarked the importance of a type system as a tool for correctness support and first-hand design. In fact, in section 1.3.3 there are multiple solutions with the purpose to bring types to Javascript. With a rich type system and a compiler, the system designer can guide the implementation through type constraints. This will decrease run-time errors, require less tests and try to move the responsibility for correctness from the developer to the technology itself.

When the concepts of functional programming are well known there is another main problem to face: *How to use them in any environments?*. Sometimes, a project has dependencies from technologies and that cannot provide functional features. If this happens, the main approach would be: to search for a library that hide the building complexity of functional abstractions and supply how to use these concepts to the developer team. In this case developers has to know how to apply the functional style and this is not obvious, especially for junior developers. In the last chapter a possible solution of that through a DSL was presented. With a DSL, the same benefits of a library are obtained, but also the usage of the library itself is encoded in the code generation ensuring the consistency of style and architecture. Moreover, it better fills the abstraction gap from the underneath technology and the functional paradigm.

Finally, the DSL can also be extended and integrated with other models in order to provide different alternatives of interaction to the system designer, as described in section 3.4.

FPML Grammar

In the following listing there's the grammar of the FPML DSL:

```
grammar it.unibo.FPML with org.eclipse.xtext.common.Terminals

generate fPML "http://www.unibo.it/FPML"

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

////////////////////////////////////////////////////////
// Entry element
////////////////////////////////////////////////////////
Model:
    elements+=PureBlock
    elements+=EffectFullBlock;

////////////////////////////////////////////////////////
// Outer Blocks
////////////////////////////////////////////////////////

PureBlock:
    'Pure' '{'
        elements+=PureDataBlock
        elements+=PureFunctionBlock
    '}';

PureFunctionBlock:
    'Functions' '{'
        (features+=PureFunctionDefinition)*
    '}';

PureDataBlock:
    'Data' '{'
        (elements+=PureData)*
        value=PureValueBlock
```

```
33      '}';
34
35  PureValueBlock:
36      'Value' '{'
37          (elements+=PureValue)*
38      '}';
39
40  EffectFullBlock:
41      'Effects' '{'
42          elements+=EffectFullDataBlock
43          elements+=EffectFullFunctionBlock
44      '}';
45
46  EffectFullDataBlock:
47      'Data' '{'
48          (elements+=EffectFullData)*
49          value=EffectFullValueBlock
50      '}';
51
52  EffectFullValueBlock:
53      'Value' '{'
54          (elements+=EffectFullValue)*
55      '}';
56
57  EffectFullFunctionBlock:
58      'Functions' '{'
59          (features+=EffectFullFunctionDefinition)*
60          main=MainFunc
61      '}';
62
63  ////////////////////////////////////////////////////
64  // Outer Block Elements
65  ////////////////////////////////////////////////////
66
67  PureData: name=ID ':' content=ValueType;
68
69  PureValue returns PureFunctionDefinition: {PureValue} name=ID
        ':' value=Expression;
70
71  EffectFullData: name=ID ':' content=EffectFullType;
72
73  EffectFullValue returns EffectFullFunctionDefinition: {
        EffectFullValue} name=ID ':' value=EffectFullExpression;
74
75  PureFunctionDefinition:
```

```
76      'def' returnType=ValueType name=ID '(' arg=PureArgument (',' higherOrderArg=AdditionalPureArgument)? ')' ':' '{' functionBody=FunctionBodyPure '}';
77
78  EffectFullFunctionDefinition:
79      'def' returnType=IOType name=ID '(' arg=Argument (',' higherOrderArg=AdditionalEffectFullArgument)? ')' ':' '{' functionBody=FunctionBodyEffectFull '}';
80
81  MainFunc:
82      'IO' returnType=UnitType 'main' ':' '{' functionBody=FunctionBodyEffectFull '}';
83
84  AdditionalPureArgument: arg2=PureArgument;
85  AdditionalEffectFullArgument: arg2=Argument;
86
87  ////////////////////////////////////////////////////
88  // Aggregate Types
89  ////////////////////////////////////////////////////
90
91  Function: EffectFullFunction | PureFunction;
92
93  PureFunction: PureFunctionDefinition | PrimitivePureFunction | PureArgument | Expression | PureValue;
94
95  EffectFullFunction: EffectFullFunctionDefinition | PrimitiveEffectFullFunction | EffectFullValue | EffectFullArgument;
96
97  PrimitiveFunction: PrimitivePureFunction | EffectFullPrimitive;
98
99  EffectFullPrimitive: PrimitiveEffectFullFunction | PrimitiveEffectFullValue;
100
101 EffectFullBodyContent: EffectFullFunction | EffectFullPrimitive | EffectFullExpression;
102
103 ////////////////////////////////////////////////////
104 // Function Body Elements
105 ////////////////////////////////////////////////////
106
107
108 Argument: EffectFullArgument | PureArgument;
109
110 EffectFullArgument: type=EffectFullType name=ID;
```

```
111
112 PureArgument: type=ValueType name=ID;
113
114 FunctionBodyPure: EmptyFunctionBody |
    CompositionFunctionBodyPure;
115
116 FunctionBodyEffectFull: EmptyFunctionBody |
    CompositionFunctionBodyEffect;
117
118 EmptyFunctionBody: {EmptyFunctionBody} 'Undefined';
119
120 CompositionFunctionBodyPure:
121       referenceElement=[PureFunction] (functionChain+=
    CompositionFunctionBodyPureFactor)+
122     | primitiveElement=PrimitivePureFunction (functionChain+=
    CompositionFunctionBodyPureFactor)+
123     | expressionElement=Expression (functionChain+=
    CompositionFunctionBodyPureFactor)+;
124
125 CompositionFunctionBodyPureFactor:
126       ('|>' (referenceElement=[PureFunction]))
127     | ('|>' (primitiveElement=PrimitivePureFunction))
128     | ('|>' (expressionElement=Expression));
129
130 CompositionFunctionBodyEffect:
131       referenceElement=[EffectFullFunction] (functionChain+=
    CompositionFunctionBodyEffectFullFactor)+
132     | primitiveElement=EffectFullPrimitive (functionChain+=
    CompositionFunctionBodyEffectFullFactor)+
133     | expressionElement=EffectFullExpression (functionChain+=
    CompositionFunctionBodyEffectFullFactor)+;
134
135 CompositionFunctionBodyEffectFullFactor:
136       ('>>=' (referenceElement=[EffectFullFunction]))
137     | ('>>=' (primitiveElement=EffectFullPrimitive))
138     | ('>>=' (expressionElement=EffectFullExpression));
139
140 ////////////////////////////////////////////////////
141 // Types
142 ////////////////////////////////////////////////////
143
144 IOType: 'IO' type=Type ;
145
146 ValueType:
```

```
147     IntegerType | StringType | BooleanType | DataType |
        PureFunctionType | PureAlgebraicType
148     | UnitType;
149
150 Type: ValueType | EffectFullType;
151
152 EffectFullType:
153     EffectFullFunctionType | EffectFullDataType |
        EffectFullAlgebraicType | IOType | VoidType;
154
155 VoidType: {VoidType};
156
157 IntegerType: {IntegerType} type="int";
158
159 StringType: {StringType} type="String";
160
161 BooleanType: {BooleanType} type="boolean";
162
163 UnitType: {UnitType} type="Unit";
164
165 DataType: {DataType} 'ref' type=[PureData];
166
167 EffectFullDataType:
168     {EffectFullDataType} 'refIO' type=[EffectFullData];
169
170 PureFunctionType:
171     {PureFunctionType} 'F' '<' argType=ValueType ',' returnType=
        ValueType '>';
172
173 EffectFullFunctionType:
174     {EffectFullFunctionType} 'FIO' '<' argType=Type ','
        returnType=IOType '>';
175
176 PureAlgebraicType: '[' pureAdtElement1=ValueType pureAdtElement2
        =(PureSumTypeFactor | PureProdTypeFactor) ']';
177
178 PureSumTypeFactor: '+' adtElement=ValueType;
179 PureProdTypeFactor: '*' adtElement=ValueType;
180
181 EffectFullAlgebraicType: '{' effectFullAdtElement1=Type
        effectFullAdtElement2=(EffectFullSumTypeFactor |
        EffectFullProdTypeFactor) '}';
182
183 EffectFullSumTypeFactor: '+' adtElement=Type;
184 EffectFullProdTypeFactor: '*' adtElement=Type;
```

```
185
186 ////////////////////////////////////////////
187 // Values
188 ////////////////////////////////////////////
189
190 EffectFullExpression:
191     {IOEffectFullExpression} 'IO' '[' innerValue=
    EffectFullExpression ']'
192     | {IOExpression} 'IO' '(' innerValue=Expression ')'
193     | {IOPureFunction} 'IOF' '(' pureFunction=[PureFunction] ')'
194     | {IOPureFunction} 'IOF' '(' purePrimitive=
    PrimitivePureFunction ')'
195     | {IOEffectFullFunction} 'IOF' '[' effectFullFunction=[
    EffectFullFunction] ']'
196     | {IOEffectFullFunction} 'IOF' '[' effectFullPrimitive=
    PrimitiveEffectFullFunction ']'
197     | EffectFullFunctionValue
198     | EffectFullDataValue
199     | EffectFullProdValue
200     | EffectFullSumValue
201     | EffectFullValueRef;
202
203
204 UnitValue returns UnitType: {UnitType} '()';
205
206 EffectFullFunctionValue returns EffectFullFunctionType: value=
    EffectFullLambda;
207
208 EffectFullValueRef: {EffectFullValueRef} 'EffectFullRef' '['
    value=[EffectFullValue] ']' ;
209
210 EffectFullLambda returns EffectFullFunctionDefinition:
211     {EffectFullLambda} '\\' '[' arg=Argument ']' '->' '{'
    functionBody=CompositionFunctionBodyEffect '}'
212     | {EffectFullLambda} '\\' '['']' '->' '{' functionBody=
    CompositionFunctionBodyEffect '}' ;
213
214 Expression:
215     IntValue
216     | StringValue
217     | BooleanValue
218     | DataValue
219     | FunctionValue
220     | UnitValue
221     | PureValueRef
```

```
222     | PureSumValue
223     | PureProdValue;
224
225 IntValue returns IntegerType: {IntegerType} value=INT;
226
227 StringValue returns StringType: {StringType} value=STRING;
228
229 BooleanValue returns BooleanType: {BooleanType} value=BOOLEAN;
230
231 FunctionValue returns PureFunctionType: value=PureLambda;
232
233 DataValue returns DataType:
234     {DataValue} type=[PureData]'(' value=
    PureExpressionAndPureFunctionReference ')';
235
236 EffectFullDataValue returns EffectFullDataType:
237     {EffectFullDataValue} type=[EffectFullData]'[' value=
    EffectFullExpressionAndEffectFullFunctionReference ']';
238
239 PureValueRef: {PureValueRef} 'PureRef' '(' value=[PureValue] ')
    ';
240
241 PureLambda returns PureFunctionDefinition:
242     {PureLambda} '\\' '(' arg=PureArgument ')' '->' '{'
    functionBody=CompositionFunctionBodyPure '}'
243     | {PureLambda} '\\' '('')' '->' '{' functionBody=
    CompositionFunctionBodyPure '}';
244
245
246 PureProdValue returns PureAlgebraicType:
247     {PureProdValue} '(' prodAdtElement1=
    PureExpressionAndPureFunctionReference ','prodAdtElement2=
    PureExpressionAndPureFunctionReference ')';
248
249 PureExpressionAndPureFunctionReference:
250     prodAdtElementExpression=Expression
251     | prodAdtElementFunction=[PureFunction];
252
253 PureSumValue returns PureAlgebraicType:
254     {PureSumValue} 'Left' '(' sumAdtElement1=
    PureExpressionAndPureFunctionReference ')'
255     | {PureSumValue} 'Right' '(' sumAdtElement2=
    PureExpressionAndPureFunctionReference ')';
256
257 EffectFullExpressionAndEffectFullFunctionReference:
```

```
258     prodAdtElementExpression=EffectFullExpression
259     | prodAdtElementFunction=[EffectFullFunction];
260
261
262 EffectFullProdValue returns EffectFullAlgebraicType:
263     {EffectFullProdValue} '[' prodAdtElement1=
        EffectFullExpressionAndEffectFullFunctionReference ','
        prodAdtElement2=
        EffectFullExpressionAndEffectFullFunctionReference ']';
264
265 EffectFullSumValue returns EffectFullAlgebraicType:
266     {EffectFullSumValue} 'Left' '[' sumAdtElement1=
        EffectFullExpressionAndEffectFullFunctionReference ']'
267     | {EffectFullSumValue} 'Right' '[' sumAdtElement2=
        EffectFullExpressionAndEffectFullFunctionReference ']';
268
269
270 ////////////////////////////////////////////
271 // Primitives
272 ////////////////////////////////////////////
273
274 PrimitivePureFunction:
275     IntToString | BoolToString | IntPow | Plus | Minus | Times |
        Mod | ApplyF | LeftAlgebraic | RightAlgebraic
276     | Equals | MinorEquals | MajorEquals | Minor | Major |
        LogicAnd | LogicOr | LogicNot
277     | ExtractPure | IsLeftPure | IsRightPure | PureIf |
        PureEitherIf | PureReturn;
278
279 IntToString: {IntToString} 'intToString';
280 BoolToString: {BoolToString} 'boolToString';
281 IntPow: {IntPow} 'intPow';
282 Plus: {Plus} '+' type=(IntegerType | StringType);
283 Minus: {Minus} '-';
284 Times: {Times} '*';
285 Mod: {Mod} 'mod';
286 LeftAlgebraic: {LeftAlgebraic} 'leftADT' type=PureAlgebraicType;
287 RightAlgebraic: {RightAlgebraic} 'rightADT' type=
        PureAlgebraicType;
288 ApplyF: {ApplyF} 'applyF' functionType=PureFunctionType value=
        ApplyFFactor;
289 ApplyFFactor:
290     valueReference=[PureFunction]
291     | '(' valueExpression=Expression ')';
292
```

```
293 Equals: {Equals} '==' type=(IntegerType | StringType |
    BooleanType);
294 MinorEquals: {MinorEquals} '<=';
295 MajorEquals: {MajorEquals} '>=';
296 Minor: {Minor} '<';
297 Major: {Major} '>';
298 LogicAnd: {LogicAnd} '&&';
299 LogicOr: {LogicOr} '||';
300 LogicNot: {LogicNot} 'not';
301 ExtractPure: {ExtractPure} 'extract' data=[PureData];
302 IsLeftPure: {IsLeftPure} 'isLeft' type=PureAlgebraicType;
303 IsRightPure: {IsRightPure} 'isRight' type=PureAlgebraicType;
304 PureIf: {PureIf} 'if' 'then' '{' then=PureIfBody '}' 'else' '{'
    else=PureIfBody '}';
305 PureIfBody: functionReference=[PureFunction]
306     | functionExpression=Expression;
307
308 PureEitherIf: {PureEitherIf} 'ifEither' 'then' '{' then=
    PureIfBody '}' 'else' '{' else=PureIfBody '}';
309
310 PureReturn: {PureReturn} 'return' type=ValueType;
311
312 PrimitiveEffectFullFunction:
313     Print | ApplyFIO | EffectFullReturn | LeftAlgebraicIO |
    RightAlgebraicIO | ExtractEffectFull
314     | LiftPureFunction | LiftEffectFullFunction |
    IsLeftEffectFull | IsRightEffectFull | EffectFullIf
315     | EffectFullEitherIf | GetLineStdIn | GetIntSdtIn;
316
317 Print: {Print} "print";
318
319 LeftAlgebraicIO: {LeftAlgebraicIO} 'leftADT' type=
    EffectFullAlgebraicType;
320 RightAlgebraicIO: {RightAlgebraicIO} 'rightADT' type=
    EffectFullAlgebraicType;
321
322 PrimitiveEffectFullValue: Random  | Time;
323
324 Random: {Random} "randomInt";
325
326 EffectFullReturn: {EffectFullReturn} "return" type=Type;
327
328 Time: {Time} "currentTime";
329
```

```
330 ApplyFIO: {ApplyFIO} 'applyFIO' functionType=
    EffectFullFunctionType value=ApplyFIOFactor;
331 ApplyFIOFactor:
332       valueReference=[EffectFullFunction]
333     | valuePrimitive=EffectFullPrimitive
334     | '(' valueExpression=EffectFullExpression ')';
335
336 ExtractEffectFull: {ExtractEffectFull} 'extractEffectFull' data
    =[EffectFullData];
337
338 LiftPureFunction: 'Lift' '(' functionRef=[PureFunction] ')'
339     | 'Lift' '(' functionPrimitive=PrimitivePureFunction ')';
340
341 LiftEffectFullFunction: 'Lift' '[' functionRef=[
    EffectFullFunction] ']'
342     | 'Lift' '[' functionPrimitive=PrimitiveEffectFullFunction
    ']';
343
344 IsLeftEffectFull: {IsLeftEffectFull} 'isLeft' type=
    EffectFullAlgebraicType;
345 IsRightEffectFull: {IsRightEffectFull} 'isRight' type=
    EffectFullAlgebraicType;
346
347 EffectFullIf: {EffectFullIf} 'if' 'then' '{' then=
    EffectFullIfBody '}' 'else' '{' else=EffectFullIfBody '}';
348 EffectFullEitherIf: {EffectFullEitherIf} 'ifEither' 'then' '{'
    then=EffectFullIfBody '}' 'else' '{' else=EffectFullIfBody
    '}';
349 EffectFullIfBody: functionReference=[EffectFullFunction]
350     | functionExpression=EffectFullExpression;
351
352 GetLineStdIn: {GetLineStdIn} 'getLineStdIn';
353 GetIntSdtIn: {GetIntStdIn} 'getIntStdIn';
354
355 ////////////////////////////////////////////////
356 // TERMINALS
357 ////////////////////////////////////////////////
358
359 terminal BOOLEAN returns ecore::EBoolean: 'true' | 'false';
```

Listing 1: FPML Grammar

Projects Setup

This reports comes with some embedded source code for examples and projects, so it is useful to provide to the reader a way to setup and try it directly. In this appendix, there are the instructions and tools used to craft that code. In case of any problem feel free to contact me directly, you can find me on Github (`https://github.com/benkio`).

The first step is to setup a few develop tools to grab, build and execute the actual code:

Git[10] This is a famous source version control used to update and fetch the code from the Github site. Its installation process is straight forward for all the main platforms. To test if it is correctly installed simply check the presence of the ‘git‘ command in the terminal.

Java Development Kit The only operation to setup this tools is again download and install it from the Oracle site (`http://www.oracle.com/technetwork/java/index.html`). Again, to test if it is working check in the terminal the presence of the ‘java‘ and ‘javac‘ command in the terminal.

Xtext and Xtend The DSL is based on this technology. The best way to setup it is to download directly the Eclipse IDE with the Xtext and Xtend plugins already installed from the Xtext website[20, 19].

Stack To setup the Haskell code and projects you need the stack tool. It helps in managing all the dependencies as well as the Haskell compiler as well.

The next step is "fetch the projects and code". The links to the Github repositories are the following:

Thesis's Examples `https://github.com/benkio/ThesisExamples`

Domain Specific Language(FPML) `https://github.com/benkio/FPML`

Based on the repository, to grab the code we have to clone the repository in a target machine. This can be done using git and the following command in the target folder:

```
git clone https://github.com/benkio/ThesisExamples.git
```

This will create a new folder with the same name of the cloned project inside the folder where the command is executed. From now on the procedure changes based on the underneath technology.

.1 Thesis's Examples

This project is a very simple one, so all you need are few stack commands:

1. Dependencies :: In order to try the code, the target machine must setup the environment properly. First go to the project folder with the terminal and than run this stack command:

   ```
   stack setup
   ```

   The tool will download the compiler and all the dependencies in a separate place creating a specific sandbox for the project.

2. Build :: This is the first step, after the previous point, in every project with compiled languages. Again, the process is very simple. Place with a terminal in the project folder and execute the command:

   ```
   stack build
   ```

3. Interpreter :: For the sake of simplicity, most of the examples are executed in the interpreter of the GHCI. The project provide the functions used in the interpreter. What we need to do in order to enter in the interpreter, and load all the code of the project doing that, is run the following command as previously:

```
stack ghci
```

Inside the interpreter we can execute and use all the code and get immediately the results. To better manage the interpreter check out the ‘help‘ inside the prompt typing ‘:h‘. To exit the interpreter type ‘:quit‘.

.2 Domain Specific Language

To open and run the DSL of this report, it must be imported in the IDE with xtext and xtend plugins installed, I suggest the Eclipse IDE. The common procedure to import a project in eclipse is to follow the tutorial accessible from the menu File->Import. Be sure to use the option "Existing project in the workspace" in order to select the cloned repository folder ‘FPML‘. Using Eclipse the projects will be immediately recognized and easily imported.

Once the ‘Package Explorer‘, search for it in the menu Window->Show View, is loaded with all the project related to the DSL, then the Xtext artifacts has to be generated. Go to the file ‘FPML.xtext‘ in the package ‘it.unibo‘ for the project ‘fpml‘, right click on it, select ‘Run As‘ in the menu and click on ‘Generate Xtext Artifacts‘. The Xtext plugin will run and generate all the classes and structure based on the grammar listed in the appendix 4.

Now all is ready for launching the environment where the system designer can create its model based on the DSL, just go to the project ‘fpml‘ on the ‘package explorer‘, right click on it, go to ‘Run As‘ and select the option ‘Eclipse Application‘. A new Eclipse instance will appear. Inside the new eclipse we have to create a new project. In order to work well this project must have:

- A new folder called ‘src-gen‘ where the generated code will be placed. This folder must be marked as a source folder in the project properties.
- The FunctionalJava library[57] dependency. At the moment of writing the latest version is ‘4.6‘.

Finally the only thing to do is the creation of the model’s file in the ‘src‘ folder with the extension ‘*.FPML‘. If all goes right the editor will provide

the intellisense and code highlighting. To get started, check out the section 3.3 and the model structure in figure 3.1.

Bibliography

[1] Ashish Agarwal. *Commercial Users of Functional Programming*. URL: http://cufp.org.

[2] Alvin Alexander. *Who's using Scala?* 2016. URL: http://alvinalexander.com/scala/whos-using-scala-akka-play-framework.

[3] Inc. Amazon.com. *SimpleDB*. 2007. URL: https://aws.amazon.com/simpledb/.

[4] Mahesh Balakrishnan et al. "Tango: Distributed Data Structures over a Shared Log". In: (). URL: http://www.cs.cornell.edu/~taozou/sosp13/tangososp.pdf.

[5] *BGGA Project Website*. URL: http://javac.info.

[6] TIOBE software BV. *Tiobe Index*. URL: http://www.tiobe.com/tiobe-index/.

[7] Ben Christensen and Jafar Husain. *Reactive Programming in the Netflix API with RxJava*. URL: http://techblog.netflix.com/2013/02/rxjava-netflix-api.html.

[8] Inc. Cognitech. *Datomic Website*. URL: http://www.datomic.com.

[9] Inc. Cognitect. *Clojure Success Stories*. URL: http://cognitect.com/clojure#successstories.

[10] Software Freedom Conservancy. *Git Version Control Website*. URL: https://git-scm.com.

[11] Intel Corporation. *Intel Website*. URL: http://www.intel.eu/content/www/eu/en/homepage.html.

[12] Linkedin Corporation. *Linkedin Website*. URL: https://www.linkedin.com.

[13] Evan Czaplicki. “Elm: Concurrent FRP for Funcitonal GUIs”. PhD thesis. 2012. URL: http://elm-lang.org/assets/papers/concurrent-frp.pdf.

[14] Evan Czaplicki. *Elm Website*. URL: http://elm-lang.org.

[15] *Expert to Expert: Brian Beckman and Erik Meijer - Inside the .NET Reactive Framework (Rx)*. URL: https://channel9.msdn.com/Shows/GoingDeep/Expert-to-Expert-Brian-Beckman-and-Erik-Meijer-Inside-the-NET-Reactive-Framework-Rx.

[16] Richard Feldman. *Walkthrough: Introducing Elm to a JS Web App*. 2016. URL: http://tech.noredink.com/post/126978281075/walkthrough-introducing-elm-to-a-js-web-app.

[17] Apache Software Foundation. *Apache CouchDB Website*. URL: http://couchdb.apache.org.

[18] FSharp Software Foundation. *FSharp Testimonials*. URL: http://fsharp.org/testimonials/.

[19] The Eclipse Foundation. *Xtend Website*. URL: https://eclipse.org/xtend/.

[20] The Eclipse Foundation. *Xtext Website*. URL: https://www.eclipse.org/Xtext.

[21] Martin Fowler. “Event Sourcing: Capture all changes to an application state as a sequence of events”. In: (2005). URL: http://martinfowler.com/eaaDev/EventSourcing.html.

[22] Nick Gerakines and Mark Zweifel. *Developing Erlang at Yahoo*. 2008. URL: http://cufp.org/archive/2008/slides/GerakinesNick.pdf.

[23] Charlie Gower. *Functional Programmers are Better Programmers*. URL: http://blog.functionalworks.com/2016/01/28/functional-programmers-are-better-programmers/.

[24] Ossi Hanhinen. *How Elm Made Our Work Better*. 2016. URL: http://futurice.com/blog/elm-in-the-real-world.

[25] *Haskell in Industry*. URL: https://wiki.haskell.org/Haskell_in_industry.

[26] Rich Hickey. *Clojure Companies*. URL: https://clojure.org/community/companies.

[27] Paul Holser, Javier Fernandez-Ivern, et al. *JUnit QuickCheck Github*. URL: https://github.com/pholser/junit-quickcheck.

[28] J. Hughes. "Why Functional Programming Matters". In: *The Computer Journal* 32.2 (Jan. 1989), 98–107. DOI: 10.1093/comjnl/32.2.98.

[29] Neil Hughes. *Netflix boasts 37% share of Internet traffic in North America, compared with 3% for Apple's iTunes*. Jan. 2016. URL: http://appleinsider.com/articles/16/01/20/netflix-boasts-37-share-of-internet-traffic-in-north-america-compared-with-3-for-apples-itunes.

[30] Facebook Inc. *Facebook Website*. URL: https://www.facebook.com.

[31] Jet.com Inc. *Jet Website*. URL: https://jet.com.

[32] Jet.com Inc. *Using FSharp at Jet.com*. 2016. URL: https://channel9.msdn.com/Events/FSharp-Events/fsharpConf-2016/INTERVIEW-Using-F-at-Jetcom.

[33] Lightbend Inc. *Akka Library*. URL: http://www.lightbend.com/platform/development/akka.

[34] Lightbend Inc. *Langom*. URL: http://www.lightbend.com/platform/development/lagom-framework.

[35] Lightbend Inc. *Lightbend Website*. URL: http://www.lightbend.com.

[36] Lightbend Inc. *Play Framework*. URL: http://www.lightbend.com/platform/development/play-framework.

[37] Lightbend Inc. *Spark*. URL: http://www.lightbend.com/platform/development/spark.

[38] Netflix Inc. *Netflix Website*. URL: https://www.netflix.com.

[39] Twitter Inc. *Twitter Website*. URL: https://twitter.com.

[40] Javaslang. *Java Slang Website*. URL: http://www.javaslang.io.

[41] Phil Johnson. *Interest in F# trending (slowly) upwards*. 2015. URL: http://www.itworld.com/article/2895038/interest-in-f-trending-slowly-upwards.html.

[42] École Polytechnique Fédérale de Lausanne (EPFL). *Scala Website*. URL: https://www.scala-lang.org.

[43] Lightbend. *Lightbend Case Study*. URL: http://www.lightbend.com/resources/case-studies-and-stories.

[44] Hai Liu et al. "The Intel labs Haskell research compiler". In: *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell - Haskell '13* (2013). DOI: 10.1145/2503778.2503779.

[45] Simon Marlow. *Fighting Spam with Haskell*. URL: https://code.facebook.com/posts/745068642270222/fighting-spam-with-haskell/.

[46] Simon Marlow, Jon Coens, Louis Brandy, et al. *The Haxl Project at Facebook*. URL: https://wiki.haskell.org/wikiupload/c/cf/The_Haxl_Project_at_Facebook.pdf.

[47] Simon Marlow and Jonf Purdy. *Open-sourcing Haxl, a library for Haskell*. URL: https://code.facebook.com/posts/302060973291128/open-sourcing-haxl-a-library-for-haskell/.

[48] Conor Mcbride and Ross Paterson. "Applicative programming with effects". In: *Journal of Functional Programming* 18.01 (2007). DOI: 10.1017/s0956796807006326.

[49] Katie Miller. *Haskell is not for production and other tales*. URL: http://www.codemiller.com/talks/.

[50] Alan Morrison. "The Rise of Immutable Data Stores". In: (2015). URL: http://www.pwc.com/us/en/technology-forecast/2015/remapping-database-landscape/immutable-data-stores--rise.html.

[51] Henrik Nilsson, Antony Courtney, and John Peterson. "Functional reactive programming, continued". In: *Proceedings of the ACM SIGPLAN workshop on Haskell - Haskell '02* (2002). DOI: 10.1145/581690.581695.

[52] Richard Nilsson. *ScalaCheck*. URL: http://www.scalacheck.org.

[53] Oracle. *Stream (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html.

[54] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. 1st. O'Reilly Media, Inc., 2008, pp. 346–354. ISBN: 0596514980, 9780596514983.

[55] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. 1st. O'Reilly Media, Inc., 2008, p. 320. ISBN: 0596514980, 9780596514983.

[56] Alex Payne. *The How and Why of Scala at Twitter*. 2010. URL: http://www.slideshare.net/al3x/the-how-and-why-of-scala-at-twitter.

[57] Mark Perry et al. *Functional Java Library Website*. URL: http://www.functionaljava.org.

[58] Leaf Petersen et al. "Measuring the Haskell Gap". In: *Proceedings of the 25th symposium on Implementation and Application of Functional Languages - IFL '13* (2014). DOI: 10.1145/2620678.2620685.

[59] Christopher Piro and Eugene Letuchy. "Functional Programming at Facebook". In: *Proceedings of the 2009 Video Workshop on Commercial Users of Functional Programming: Functional Programming As a Means, Not an End*. CUFP '09. Edinburgh, Scotland: ACM, 2009. ISBN: 978-1-60558-943-5. DOI: 10.1145/1668113.1668120. URL: http://doi.acm.org/10.1145/1668113.1668120.

[60] Christopher Piro and Eugene Letuchy. *Functional Programming at Facebook*. 2009. URL: http://cufp.org/archive/2009/slides/PiroLetuchy.pdf.

[61] Inc Pivotal Software. *RabbitMQ Website*. URL: https://www.rabbitmq.com.

[62] Iustin Pop. "Experience report". In: *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming - ICFP '10* (2010). DOI: 10.1145/1863543.1863595. URL: http://k1024.org/~iustin/papers/icfp10-haskell-reagent.pdf.

[63] Jovan Popovic. *Functional Programming in C#*. 2012. URL: https://www.codeproject.com/Articles/375166/Functional-programming-in-Csharp.

[64] Rick Reed. *The WhatsApp Architecture Facebook Bought For $19 Billion*. 2014. URL: http://highscalability.com/blog/2014/3/31/how-whatsapp-grew-to-nearly-500-million-users-11000-cores-an.html.

[65] Goswin Rothenthal. *The 3D Geometry of Louvre Abu Dhabi*. 2016. URL: https://channel9.msdn.com/Events/FSharp-Events/fsharpConf-2016/The-3D-Geometry-of-Louvre-Abu-Dhabi.

[66] *Scala in the Enterprise*. 2012. URL: www.scala-lang.org/old/node/1658.

[67] Kurt Schelfthout, Mark Seemann, et al. *FsCheck*. URL: https://fscheck.github.io/FsCheck.

[68] Nick Smallbone, Koen Claessen, Oleg Grenrus, et al. *Quickcheck Library Github*. URL: https://github.com/nick8325/quickcheck.

[69] Michael Snoyman, Mathieu Boespflug, Greg Weber, et al. *Commercial Haskell Github*. URL: https://github.com/commercialhaskell/commercialhaskell.

[70] Michael Snoyman, Emanuel Borsboom, Michael Sloan, et al. *Stack Building Tools*. URL: https://github.com/commercialhaskell/stack.

[71] Oliver Sturm. *Functional Programming in C#: Classic Programming Techniques for Modern Projects*. Wiley, John and Sons, 2011.

[72] D. A. Turner. "Some History of Functional Programming Languages". In: *Lecture Notes in Computer Science Trends in Functional Programming* (2013), 1–20. DOI: 10.1007/978-3-642-40447-4_1.

[73] *Typeclassopedia*. URL: https://wiki.haskell.org/Typeclassopedia.

[74] Tim Watson. *CloudHaskell*. URL: http://haskell-distributed.github.io.

[75] Scott Wlaschin. *Why use F#?* URL: https://fsharpforfunandprofit.com/why-use-fsharp/.

[76] Functional Works. *Functional Works Site*. URL: https://jobs.functionalworks.com/.