Evaluating Functional Programming for Maintainable REST APIs

A thesis presented
by
Marc Coquand
to
The Department of Computer Science

for a degree of
Master of Science in Engineering
in the subject of
Interaction Technology and Design

Umeå University
Supervised by Anders Broberg
June 26, 2019

Abstract

This study's goal is to investigate the use of functional programming as a measure for reducing software defects and how it affects readability. It does so by looking at a specific case and introducing a REST framework called Cause which is completely type safe and helps creating type safe REST frameworks.

Acknowledgements

TODO I want to thank myself for my amazing work

Contents

Chapter 1

Introduction: Software paradigms and complexity

Different schools of thoughts have different approaches when it comes to building applications. There is one that is the traditional, object oriented, procedural way of doing it. Then there is a contender, a functional approach, as an alternative way to build applications. Early programming languages were based around procedure calls. Procedures are the series of steps the computer needs to perform the computations desired. [1] These programming languages were turing complete. A programming language is Turing complete if is able to calculate everything that is calculatable. [2] Object-oriented programming is used to make code more reusable and structured. Java, amongst most languages, is a popular language that is turing complete, object-oriented and imperative. [3]

Functional programming originates from 1936 from Lambda calculus. Lambda calculus is a theory for functions and evolved from Church and Curry to create an alternative foundation for mathematics. [4] Even though Turing machines and Lambda calculus developed separately, the church-turing thesis proves that any computational problem that is solvable with Lambda Calculus is also solveable for Turing machines and vice versa. [5] What this means in practice is any program written in the functional paradigm can be written in the procedural, object-oriented paradigm. However it does not mean that the paradigms are the same as one solution might be very complex in a procedural language and simple in a functional language and vice versa as long as they are turing-complete.

As software engineers, one factor of concern is software quality. Software quality can be divided into two different subparts: software functional quality and software structural quality. Software functional quality reflects how well our system conforms

to given functional requirements or specification and the degree of which we produce correct software. To check that the software is correct, software engineers create tests. [6] In order to create tests, the engineer employs various patterns in the code to make the code easier to test. One approach is functional programming. However Functional programming is not the most popular approach.

Software structural quality refers to how well the software adheres to non-functional requirements such as robustness and maintainability. [6] Some of the maintainability aspects, such as readability, is hard to measure quantitatively. By performing semi-structured interviews, it is possible to investigate how well the code is understood. Since functional programming is not the most popular approach to software engineering today, it is worth taking into consideration how employing functional programming might affect the readability and maintainability of the software. If no one understands the code, how can they be expected to maintain the software?

The goal of this thesis is therefore to investigate how readability with Functional programming in a common area of software development, servers. Or more specifically, REST servers.

Chapter 2

Background

This chapter will look at how concerns and requirements made impact the maintainability and testability. It aims to establish what are the current methods of developing applications and how software engineers today approach testing.

2.1 Introduction to REST servers

Servers are applications that provide functionality for other programs or devices, called clients. Services are servers that allow sharing data or resources among clients or to perform a computation.

REST (Representational State Transfer) is a software architecture style that is used to construct web services. A so called RESTful web service allow requesting systems to access and manipulate textual representations of web services by using a set of stateless operations. The architectual constraints of REST are as follows:

Client - Server Architecture Separate the concerns between user interface concerns and data storage concerns.

Statelessness Each request contains all the information neccessary to perform a request. State can be handled by cookies on the user side or by using databases. The server itself contains no state.

Cacheability As on the World Wide Web, clients and intermediaries can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable or not to prevent clients from getting stale or inappropriate data in response to further requests. Well-managed caching partially or completely

eliminates some client–server interactions, further improving scalability and performance.

Layered system A client can not tell if it is connected to an end server or some intermeditary server.

Code on demand Servers can send functionality of a client via exectuable code such as javascript. This can be used to send the frontend for example.

Uniform interface The interface of a RESTful server consists of four components. The request must specify how it would like the resource to be represented; that can for example be as JSON, XML or HTTP which are not the servers internal representation. Servers internal representation is therefore separated. When the client holds a representation of the resource and metadata it has enough information to manipulate or delete the resource. Also the REST server has to, in it's response, specify how the representation for the resource. This is done using Media type. Some common media types are JSON, HTML and XML.

A typical HTTP request on a restful server consists of one of the verbs: GET, POST, DELETE, PATCH and PUT. They are used as follows:

GET Fetches a resource from the server. Does not perform any mutation.

POST Update or modify a resource.

PUT Modify or create a resource.

DELETE Remove a resource from the server.

PATCH Changes a resource.

A request will specify a header "Content-Type" which contains the media representation of the request content. For example if the new resource is represented as Json then content-type will be "application/json". It also specifies a header "Accept" which informs which type of representation it would like to have, for example Html or Json.

A request will also contain a route for the resource it is requesting. These requests can also have optional parameters called query parameters. In the request route:

```
1 /api/books?author=Mary&published=1995
```

the ? informs that the request contains query parameters which are optional. In the example above it specfies that the request wants to access the books resource with the parameters author as Mary and published as 1995.

When a request has been done the server responds with a status code that explains the result of the request. The full list of status codes and their descriptions can be found here: `https://en.wikipedia.org/wiki/List_of_HTTP_status_codes`

2.1.1 Implementation concerns for REST apis

A REST api has to concern themselves with the following:

- Ensure that the response has the correct status code.
- Ensure that the correct representation is sent to the client.
- Parse the route and extract it's parameters.
- Parse the query and extract it's parameters.
- Handle errors if the route or query are badly formatted.
- Generate the correct response body containing all the resources needed.

Every type of error has a specific status code, these need to be set correctly.

2.2 Architecture

When developing large scale server applications, often the requirements are as follows:

- There is a team of developers
- New team members must get productive quickly
- The system must be continuously developed and adapt to new requirements
- The system needs to be continuously tested
- System must be able to adapt to new and emerging frameworks

Two different approaches to developing these large scale applications are microservice and monolithic systems. The monolithic system comprises of one big "top-down" architecture that dictates what the program should do. This is simple to develop using some IDE and deploying simply requires deploying some files to the runtime.

As the system starts to grow the large monolithic system becomes harder to understand as the size doubles. As a result, development typically slows down. Since there are no boundaries, modularity tends to break down and the IDE becomes slower over time, making it harder to replace parts as needed. Since redeploying requires the entire application to be replaced and tests becomes slower; the developer becomes less productive as a result. Since all code is written in the same environment introducing new technology becomes harder.

In a microservice architecture the program comprises of small entities that each have their own responsibility. There can be one service for metrics, one that interacts with the database and one that takes care of frontend. This decomposition allows the developers to easier understand parts of the system, scale into autonomous teams, IDE becomes faster since codebases are smaller, faults become easier to understand as they each break in isolation. Also long-term commitment to one stack becomes less and it becomes easier to introduce a new stack.

The issue with microservices is that when scaling the complexity becomes harder to predict. While testing one system in isolation is easier testing the entire system with all parts together becomes harder.

2.2.1 Unit testing

Unit testing is a testing method where the individual units of code and operating procedures are tested to see if they are fit for use. A unit is informally the smallest testable part of the application. To deal with units dependence one can use method stubs, mock objects and fakes to test in isolation. The goal of unit testing is to isolate each part of the programs and ensure that the individual parts are correct. It also allows for easier refactoring since it ensures that the individual parts still satisfy their part of the application.

To create effective unit tests it's important that it's easy to mock examples. This is usually hindered if the code is dependant on some state since previous states might affect future states.

2.2.2 Property-based testing

Property-based testing tests the properties that a function should fulfill. A property is some logical condition that a specification defines the function should fulfill. Compared to unit testing where the programmer creates the mock values; property based testing generates values automatically to find a contradiction. For example a function $reverse : [a] \to [a]$, which takes a list and reverses it's items, should have the property $reverse \circ reverse\ x = x$. A unit test would check that $reverse[1, 2, 3] = [3, 2, 1]$; a property-based test would instead generate a list of values randomly and check it's properties hold.

2.2.3 Integration testing

Whereas unit testing validates that the individual parts work in isolation; integration tests make sure that the modules work when combined. The purpose is to expose faults that occurs when the modules interact with each other.

2.2.4 End-2-End Tests

An End-2-End test (also known as E2E test) is a test that tests an entire passage through the program, testing multiple components on the way. This sometimes requires setting up an emulated environment mock environment with fake variables.

2.2.5 Challenges

When writing unit tests that depend on some environment, for example fetching a user from some database, it can be difficult to test without simulating the environment itself. In such cases one can use dependency injections and mock the environment with fake data. Dependency injection is a method that substitutes environment calls and returns data instead. The issue with unit tests is that even if a feature works well in isolation it does not imply that it will work well when composed with other functions.

The challenge in integration and E2E-tests comes with simulating the entire environments. Given a server connected to some file storage and a database it requires setting up a local simulation of that environment to run the tests. This results in slower execution time for tests and also requires work setting up the environment. Thus it ends up being costly. Also the bigger the space that is being tested the less close the test is to actually finding the error, thus the test ends up finding some error but it can be hard to track it down.

Thus to mitigate these issues the correct architecture needs to be created to make it easier to test. However if there is nothing forcing the programmer to develop software in this way it creates the possibility for the programmer to "cheat" and create software that is not maintainable.

To mitigate the programmer from making mistakes, some languages feature a type system. The type system is a compiler check that ensures that the allowed values are entered. Different strengths exist between various programming languages with some featuring higher-kinded types (types of types) and other constructs.

It is possible to combine the type system with design patterns to force the developer to create the right thing. Such constructs that exists are monads, which can be used to force the separation between pure and impure functionality. Later chapters will introduce a REST framework named Cause, which has been created to force the developer to create REST compliant servers.

However these patterns make heavy use of functional programming. Functional programming as a software paradigm is not popular, with the preferred software paradigm being Object-oriented programming. Thus this thesis aims to investigate the effects of introducing functional constructions to programmers with little familiarity with functional programming when it comes to understandability. Understandability is important to reduce the learning time for programmers and cut down learning costs.

2.3 SOLID principles

A poorly written system can lead to rotten design. Martin Robert, a software engineer, claims that there are four big indicators of rotten design. Rotten design also leads to problems that were established in Chapter 2, such as making easy unit tests. Thus a system should avoid the following.

Rigidity is the tendency for software to be difficult to change. This makes it difficult to change non-critical parts of the software and what can seem like a quick change takes a long time.

Fragility is when the software tends to break when doing simple changes. It makes the software difficult to maintain, with each fix introducing new errors.

Immobility is when it is impossible to reuse software from other projects in the new project. So engineers discover that, even though they need the same module that was in another project, too much work is required to decouple and separate the desirable parts.

Viscosity comes in two forms: the viscosity of the environment and the viscosity of the design. When making changes to code there are often multiple solutions. Some solutions preserve the design of the system and some are "hacks". The engineer can therefore easily implement an unmaintainable solution. The long compile times affect engineers and makes them attempt to make changes that do not cause long compile times. This leads to viscosity in the environment.

To avoid creating rotten designs, Martin Robert proposes the SOLID guideline. SOLID mnemonic for five design principles to make software more maintainable, flexible and understandable. The SOLID guidelines are:

Single responsibility principle Here, responsibility means "reason to change". Modules and classes should have one reason to change and no more.

Open/Closed principle States we should write our modules to be extended without modification of the original source code.

Liskov substitution principle Given a base class and an derived class derive, the user of a base class should be able to use the derived class and the program should function properly.

Interface segregation principle No client should be forced to depend on methods it does not use. The general idea is that you want to split big interfaces to smaller, specific ones.

Dependency inversion principle A strategy to avoid making our source code dependent on specific implementations is by using this principle. This allows us, if we depend on one third-party module, to swap that module for another one should we need to. This can be done by creating an abstract interface and then instance that interface with a class that calls the third-party operations. [7]

Using a SOLID architecture helps make programs that are not as dependent on the environments, making them easier to test (swapping the production environment to a test environment becomes trivial). When investigating the testability, it is important to look at programs that are written in such a way that all parts are easy to test. Thus choosing a SOLID architecture for programs will allow making more testable software. These concepts were designed for Object-oriented programming but can be translated to functional programming which will be demonstrated after introducing the functional concepts needed.

2.4 Readability in code

When evaluating the readability of code, companies can use Code reviews. A code review is an activity in which humans check how well the code can be understood by reading it. Thus similarity it can be used to evaluate how well users without experience with functional code understand functional programs.

By creating an semi-structured interview, where the programmers is asked open questions about how the code works it can give insights about the defects of the software and if there is something fundamental about functional programming that makes it harder to understand.

Chapter 3

Theory

Based on the challenges outlined in Chapter 2, the goal now becomes to construct a maintainable library for REST apis. The approach in this thesis is by using constructs from Functional programming. This chapter will introduce the fundamentals of functional programming to then move on and use that to construct a server library with automatic error handling and type safety.

3.1 Concepts from Functional Programming

While different definitions exist of what Functional programming means, we define functional programming as a paradigm that uses of pure functions, decoupling state from logic, using trait-based polymorphism and immutable data.

Purity When a function is pure it means that calling a function with the same arguments will always return the same value and that it does not mutate any value. For example, given $f(x) = 2 \cdot x$, then $f(2)$ will always return 4. It follows then that an impure functions is either dependant on some state or mutates state in some way. For example, given $g(x) = currenttime \cdot x$, $g(5)$ will yield a different value depending on what time it is called. This makes it dependant on some state of the world. Or given $x = 0$, $h() = x + 1$. Then $h()$ will yield $x = 1$ and $(h \circ h)()$ will yield $x = 2$, making it impure. [8]

Trait-based polymorphism OOP inherits classes that contain methods and attributes. [9] Functional programs instead define classes that describe the actions that are possible. For example, a class `Equality` could contain a function `isEqual` that checks if two data types are equal. Then any data type that implements the interface Equality, for example lists or binary trees, would be able

to use the function isEqual. This is known as type-classes in Haskell[1], mixins in Javascript[2] or traits in Scala[3].

Immutable data by deafult Immutable data is data that after initialization can not change. This means if we initialize a record, `abc = {a: 1, b: 2, c: 3}` then `abc.a := 4` is an illegal operation. Immutable data, along with purity, ensures that no data can be mutated unless it is specifically created as mutable data.

Higher-order functions Higher-order functions are functions which either return a function or take one or more functions as arguments. A function $twice : (a \to a) \to (a \to a)$, $twice\ f = f \circ f$, takes a function as an argument and returns a new function which performs given function twice on the argument. So for example $addOne = (+)1$, $addTwo = twiceaddOne$.

Decoupling state from logic Even if functional programs emphasise purity applications still need to deal with state somehow. For example a server would need to interact with a database. Functional programs solve this by separating pure functions and effectful functions. Effects are observable interactions with the environment, such as database access or printing a message. While various strategies exist, like Functional Reactive Programming[4], Dialogs[5] or uniqueness types[6], the one used in Haskell (the language used in this thesis to construct the programs) is the IO monad. For the uninitiated, one can think of Monads as a way to note which functions are pure and which are effectful and managing the way they intermingle. It enables handling errors and state.[7].

As a strategy to further separate state and logic, one can construct a three-layered architecture, called the three layer Haskell cake. Here, the strategy is that one implements simple effectful functions, containing no logic as a base layer. Then on a second layer one implements an interface that implements a pure solution and one effectful solution. Then on the third layer one implements the logic of the program in pure code. The way the second layer is implemented is explained further in Section **??**.

[1] `www.learnyouahaskell.com/types-and-typeclasses`
[2] `www.typescriptlang.org/docs/handbook/mixins.html`
[3] `https://docs.scala-lang.org/tour/traits.html`
[4] Read more: `en.wikipedia.org/wiki/Functional_reactive_programming`
[5] Read more: `stackoverflow.com/questions/17002119/haskell-pre-monadic-i-o`
[6] Read more: `https://en.wikipedia.org/wiki/Clean_(programming_language)`
[7] This is simplified as Monads are notoriously difficult to explain.

So while no exact definition of Functional programming exist, this thesis defines it as making functions pure and inheritance being based around functionality rather than attributes.

More advanced constructs also exists for functional programming that need to be introduced in order to construct a maintainable rest api.

3.1.1 ADTs: Sum types and product types

A type is in Haskell a *set* of possible values that a given data can have. This can be *int*, *char* and custom defined types. A *sum type* or *union type* is a type which is the sum of types, meaning that it can be one of those it's given types. For example the type `type IntChar = Int | Char` is either an Int or a Char. A useful application for sum types are enums such as `type Color = Red | Green | Blue`, meaning that a value of type Color is either red, green or blue. A sum type can be used to model data which may or may not have a value, by introducing the Maybe type: `type Maybe value = Just value | Nothing`.A product type is a type which is the product of types, for example `type User = User Name Email`. Informally, a product type can be likened to a record in Javascript. This allows us to model computations that might fail. For example given $sqrt(x) = \sqrt{x}$, $x \in \mathbb{Z}$ then $sqrt(-1)$ is undefined and would cause Haskell to crash. Instead by introducing a function `safeSqrt`, where `safeSqrt x = if x > 0 then Just (sqrt x) else Nothing`, the program can force the developer to handle the special case of negative numbers. Sum types are useful for implementing the Interpreter pattern, explained in Section **??**.

3.1.2 GADT

a GADT is a *generalized abstract data type*. They specify, depending on the input, what the output should be of that type. GADT enables implementing *domain-specific languages* (DSL). A DSL is a language with a limited scope for specific applications. For example a parsing library or a calculator.

```
1          data Calculator = Number Int
2            | Add Calculator Calculator
3            | Multiply Calculator Calculator
```

Figure 3.1: A Calculator GADT with two operations add and multiply.

```
1          mathExpression = (Number 5 `Add` Number 3) `Multiply` (
               Number 4 `Add` Number 3)
```

Figure 3.2: A mathematical expression constructed using the GADT in figure 3.1

Figure 3.1 defines a GADT for a calculator. The calculator can do two operations, add and multiply. This allows us to construct mathematical expressions. The expression in Figure 3.2 can translates to $(5+3)*(4+3)$ by defining a way to evaluate the expression. Figure 3.3 defines an evaluation for the program.

```
1          evaluate :: Calculator -> Int
2          evaluate (Add expr1 expr2) = evaluate expr1 + evaluate expr2
3          evaluate (Multiply expr1 expr2) = evaluate expr1 + evaluate
               expr2
```

Figure 3.3: Evaluator for the calculator

3.1.3 Type classes

A type class is a construct that allows for ad hoc polymorphism. This allows to create constraints to type variables in parametrically polymorphic types. In English, that means that it allows creating interfaces that must be implemented for the types. For example the equality type class, defined in Figure 3.4

```
1          class Eq a where
2            (==) :: a -> a -> Bool
3            (/=) :: a -> a -> Bool
```

Figure 3.4: Equality type class in Haskell.

By defining an Equality type class one can create general functions that can be used for anything that is "equalable". For example Figure 3.5 is a function that prints a text if two items are equal. This function can be used for floats, ints, tuples and everything else that implements the `Eq` type class. Other uses for type classes is Num which implements numeric operations for floats and integers. This is useful for implementing the MTL technique which will allow us to implement the Interpreter pattern which will be described in the following sections.

```
printIfEqual :: Eq a => a -> a -> IO ()
printIfEqual a b =
    if a == b then
        putStrLn "They are equal"
    else
        putStrLn "They are not equal"
```

Figure 3.5: A function that prints a text if the two items are equal.

3.1.4 Functors and Contravariant Functors

A useful type class is the Functor type class. It defines a function $map : (a \to b) \to m\ a \to m\ b$. So every type that can be mapped over is a Functor. Examples of this are lists, where map morphs every value in the list from a to b. Another example is for Maybe, defined in 3.1.1. A Functor for Maybe checks if the value is $Just\ a$, if so it morphs that value to $Just\ b$, otherwise it returns $Nothing$.

Not every type with a type parameter is a Functor. For example the type $Predicate = a \to Boolean$, is a function that when given some value a returns a boolean. This type can not be a Functor due to the type parameter being the *input* of the function. When the type parameter of the type is the input, it is in negative position and the type is *contravariant*. When the type parameter is the output of a function, it is in positive position and the type is covariant. A type can be a Functor only if it is covariant.

Contravariant Functors are type classes that define a function $contramap : (a \to b) \to m\ b \to m\ a$. These are useful for defining how the value should be *consumed*. So for example a type $encoder : a \to encoded$, defines an encoder. The contravariant functor would allow transforming the encoder into intermediate value.

3.1.5 Brief introduction to Monads for side effects

Monads[8] are a way to sequence computations that might fail while automating away boilerplate code. Figure 3.6 shows how Monads are implemented as a typeclass in Haskell. It implements the function `return`, the function bind `(>>=)`, the function sequence `(>>)` which is bind whilst ignoring the prior argument and `fail` which handles crashes.

[8] `en.wikipedia.org/wiki/Monad_(functional_programming)`

```haskell
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    fail :: String -> m a
    fail msg = error msg
```

Figure 3.6: Monad type class in Haskell.

Informally, Monads are as a design pattern that allows us to sequence different computations. Without them the developer would have to explicitly check if a computation has failed. For example, given the function $unsafeSqrtLog = sqrt \circ log$, then $unsafeSqrtLog(-1)$ would throw an error since log and $sqrt$ are undefined for -1. Section 3.1.1 showed how the `Maybe value` type could be used to create a safe computation `safeSqrt`. To sequence that computation with a function `safeLog`, the user would have to manually check that `safeSqrt` returned a value `Just result` and not `Nothing`. Monads allows sequencing these computations without explicitly writing this check, so composing `safeSqrt` and `safeLog` using bind becomes `safeSqrtLog n = safeSqrt n >>= safeLog`. The same idea applies for effectful computations such as fetching data from a database.

3.1.6 Strong Profunctors

Some types are both contravariant and covariant, such as the type $computation\ a\ b = a \to Result\ b$, since computation has a type parameter in positive and negative position. A Profunctor is a type class that contains the function $dimap : Profunctor\ p \Rightarrow (a \to b) \to (c \to d) \to p\ b\ c \to p\ a\ d$. $dimap$ both contramaps and maps the type at the same time. So since *computation* is both covariant and contravariant it is also a profunctor.

A Strong Profunctor defines two functions, $first : Profunctor\ p \Rightarrow p\ a\ b \to p\ (a, c)\ (b, c)$, $second : Profunctor\ p \Rightarrow p\ a\ b \to p\ (c, a)\ (c, b)$. This reveals the core strength of Profunctors, to compute two separate values and then merge those together. By defining a function $merge : Strong\ p \Rightarrow p\ a\ (b \to c) \to p\ (a, b) \to c$, a Strong Profunctor gives a "memory" to the function, so that values can be computed both on the input and the output of the function. How this is useful will be demonstrated in the functional REST server chapter.

3.2 SOLID principles with functional programming

To create maintainable functional programs the concepts from SOLID architecture must be translated over to the functional paradigm where applicable.

3.2.1 Single Responsibility Principle

A function takes a single input and produces a single output. If file structure is centered around the morphisms of a single type. The responsibility of a file is to morph that type into some other value. Thus it keeps the modules focused and simple.

3.2.2 Liskov Substitution Principle

Liskov's Substitution Principle are used for reasoning about subtyping among objects. Since objects do not exist for functional programs some translating is needed of those concepts. The formal requirements of Liskov's Substitution Principle are as follows:

- Contravariance of method arguments should be in the subtype.
- Covariance of method arguments in the subtype.
- No new exceptions should be thrown by each subtype, except where those exceptions are themselves subtypes of exceptions thrown by the supertype.

Let's break each of these down to see how they translate in functional programming:

Contravariance of method arguments should be in the subtype. In functional programming, given a type which is contravariant. In order to morph that into a subtype the function contramap must be used. Recall that $contramap : (a \to b) \to t\ b \to t\ a$. For a function $from \to to$ to exist $from$ must be a subtype of to! Thus this is guaranteed.

Covariance of method arguments in the subtype. Same logic as for contravariance applies for covariance. In order to morph a subtype using map there must be a function $subtype \to type$.

No new exceptions should be thrown by each subtype, except where those exceptions ar Since exceptions are represented by a sum type *Result*, described in previous chapters, it is impossible for subtypes to throw new exceptions.

3.2.3 Dependency Inversion Principle

Dependency Inversion Principle states that the logic should not depend on it's environment. To achieve that in functional programming the environment can be abstracted and taken as parameters of the program. For instance given the program readNPrint in Figure 3.7, this program depends on the computer IO, making it difficult to extend it to different environments, such as databases.

```
1         readNPrint : IO ()
2         readNPrint = readLine >>= putStrLn
```

Figure 3.7: A program that reads input from the computer and then prints it.

Instead, Figure 3.8 shows how the parameters are abstracted and readNPrint is a higher order function instead that takes some function that can generate a string and some function that can print a string.

```
1         readNPrint : (IO String) -> (String -> IO ()) -> IO ()
2         readNPrint reader printer = reader >>= printer
3
4         -- and then later
5         consoleIO : IO ()
6         consoleIO = readNPrint readLine putStrLn
```

Figure 3.8: A program that reads input from the computer and then prints it, where the logic is separated from it's environment.

3.2.4 Interface Segregation Principle

Interface Segragation Principle states that no client should be forced to depend on methods it does not use.

This can be enforced by the use of GADTs

3.3 Functional servers

3.3.1 RESTful servers

Servers are applications that provide functionality for other programs or devices, called clients. Services are servers that allow sharing data or resources among clients or to perform a computation.

REST (Representational State Transfer) is a software architecture style that is used to construct web services. A so called RESTful web service allow requesting systems to access and manipulate textual representations of web services by using a set of stateless operations. The architectual constraints of REST are as follows:

Client - Server Architecture Separate the concerns between user interface concerns and data storage concerns.

Statelessness Each request contains all the information neccessary to perform a request. State can be handled by cookies on the user side or by using databases. The server itself contains no state.

Cacheability As on the World Wide Web, clients and intermediaries can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable or not to prevent clients from getting stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.

Layered system A client can not tell if it is connected to an end server or some intermeditary server.

Code on demand Servers can send functionality of a client via exectuable code such as javascript. This can be used to send the frontend for example.

Uniform interface The interface of a RESTful server consists of four components. The request must specify how it would like the resource to be represented; that can for example be as JSON, XML or HTTP which are not the servers internal representation. Servers internal representation is therefore separated. When the client holds a representation of the resource and metadata it has enough information to manipulate or delete the resource. Also the REST server has to, in it's response, specify how the representation for the resource. This is done using Media type. Some common media types are JSON, HTML and XML.

A typical HTTP request on a restful server consists of one of the verbs: GET, POST, DELETE, PATCH and PUT. They are used as follows:

GET Fetches a resource from the server. Does not perform any mutation.

POST Update or modify a resource.

PUT Modify or create a resource.

DELETE Remove a resource from the server.

PATCH Changes a resource.

A request will specify a header "Content-Type" which contains the media representation of the request content. For example if the new resource is represented as Json then content-type will be "application/json". It also specifies a header "Accept" which informs which type of representation it would like to have, for example Html or Json.

A request will also contain a route for the resource it is requesting. These requests can also have optional parameters called query parameters. In the request route:

```
1 /api/books?author=Mary&published=1995
```

the ? informs that the request contains query parameters which are optional. In the example above it specfies that the request wants to access the books resource with the parameters author as Mary and published as 1995.

When a request has been done the server responds with a status code that explains the result of the request. The full list of status codes and their descriptions can be found here: `https://en.wikipedia.org/wiki/List_of_HTTP_status_codes`

3.3.2 Implementation concerns for REST apis

A REST api has to concern themselves with the following:

- Ensure that the response has the correct status code.
- Ensure that the correct representation is sent to the client.
- Parse the route and extract it's parameters.
- Parse the query and extract it's parameters.
- Handle errors if the route or query are badly formatted.

- Generate the correct response body containing all the resources needed.

Every type of error has a specific status code, these need to be set correctly.

3.4 Architect of a sustainable server

3.5 Modeling a server in functional programming

3.5.1 Constraints for good architecture

3.5.2 Look

3.5.3 Implementation

3.5.4 Definitions

Verbs and Media types

A verb is either get, post, put, patch or delete, represented as a sum type $typeVerb = Get \mid Post \mid Put \mid Patch \mid Delete$. Content types are harder to represent as a sum type due to there being so many and are thus represented as a string instead.

Request and Response

A Request is a product type consisting of a url, queries, accept header (Media), headers (a list of string tuples), accept media, verb, encodded body and an encoding. The correct encoder can be deduced from the accept header.

A Response is a product type consisting of a body, status code, content type Media, headers (a list of string tuples) and an *encoder for the body*. An encoded response instead contains an encoded body. These are separated because the body will be constructed separately from the encoder.

A server of parameter b is a type that takes a request and transforms it into a response of parameter b. I.E. $server : Request \rightarrow Response\ encoded$. A $Response\ b$ is a contravariant product type consisting a status code, a set of headers, a content type, a function $body : a \rightarrow encoded$ and encoding.

```
1   type Response a = {
2     code: StatusCode,
3     headers: Header ,
4     contentType: MediaType,
5     body: a -> encoded,
```

```
6       encoding: Encoding.t,
7     };
```

When constructing a Response a series of computations need to be done. If one of these fails, due to internal implementation errors or invalid requests there needs to be an error handling system in place. To account for this, a type $computation\ a\ b = a \rightarrow Result\ b$, where result is a sum type $Result\ a = Ok\ a \mid Fail\ Code\ Media\ Message$, is introduced. Computation is also a Monad, meaning that computations can be composed together, where if one computation should fail it skips executing the rest and returns a failure containing the correct status code.

An endpoint is a computation of a *Request* to a tuple of function handler h and a *Response body*. An endpoint $e\ a\ b$ is complete if $e \sim computation\ Request\ (Result\ b, Response\ b)$. For all complete endpoints, there exists a function $\forall b.part\ b\ b \sim Request \rightarrow (Result\ b, Response\ b) \Rightarrow part\ b\ b \rightarrow server$. This function applies the result of the handler to the Response encoder to create *Response encoded*. Thus to construct a server from an endpoint, the endpoint must be complete and for all complete endpoints, handler is $Result\ a$.

Since a computation is a strong profunctor and a monad, non-complete endpoints can be composed together to produce a complete endpoint. Thus we introduce combinators as follows.

Extract the encoder from the request and contramap that encoder to the response:

$$\begin{aligned} accept : [Media, a \rightarrow encoded] \\ \rightarrow endpoint\ h\ encoded \\ \rightarrow endpoint\ h\ a \end{aligned} \tag{3.1}$$

Extract a query parameter and apply that to the handler:

$$\begin{aligned} query : parameter \\ \rightarrow (value \rightarrow Maybe(a)) \\ \rightarrow endpoint\ (Maybe(a) \rightarrow h)\ b \\ \rightarrow endpoint\ h\ b \end{aligned} \tag{3.2}$$

Depending on the URI of the request, a different endpoint should be

- An ordered set of required parameters to execute a handler.
- An ordered set of optional query parameters to execute a handler.
- A set of accepted representations for a resource as well as an accompanying function that encodes the resource to the represenation.

- A handler to execute, where a handler must take the correct parameters and return the correct resource.
- A set of http request methods supported by the endpoint.

Each computation in an endpoint will morph the handler or the Response. An endpoint consists of a composition of parts, where $part\ h\ b = computation\ Request\ (h, Response\ b)$ and h is a handler. A *part* is *complete* if $part \sim Request \to (Result\ b, Response\ b)$. When a part is complete it means that all parameters from the request have been extracted and appended to the handler and the Response has a way to encode the result of the handler. Thus a function $\forall b.part\ b\ b \sim Request \to (Result\ b, Response\ b) \Rightarrow part\ b\ b \to server$ exists.

The body of a response is a function that transforms the resource into it's requested representation. If the request specifies accept as `application/json` then the body function turns the body into a json format.

A *Request* is a monad that parses the incoming request. It transforms that into a request handler that then feeds the result into a response. A handle is a function $a \to Response\ b$. The extended definition of the server is then $Server : Request\ a \to (a \to Response\ b) \to Response\ b$.

3.6 Cause, a functional REST framework

Cause is a high-level web framework that allows writing composable REST frameworks in Reasonml. In Cause, a REST api is a *specification*, where the user specifies endpoints. An endpoint in a spec contains the following: An endpoint is composable, meaning that you can create a *connector endpoint* which consists of two *subendpoints*. The following operations exist for endpoints

oneOf : $list(endpoint)$ Creates a connector endpoint out of a list of subendpoints.

is : $string \to endpoint$ Checks that the request contains a given string in it's path.

int : $endpoint$ Extracts an integer from the request path and feeds it into the handler.

text : $endpoint$ Extracts a text from the request path and feeds it into the handler.

contentType : $list(MediaType, a \to encoded)$ Checks that the request contains one of the supported media types and sets the appropriate encoder.

accept : $list(MediaType, encoded \to a)$ Takes a set of supported mediatypes and a way to transform the encoded value into the value.

(↣) : $endpoint \to endpoint \to endpoint$ An operator for composing endpoints.

get : $handler \to endpoint \to endpoint$ Connects an endpoint to a handler and ensures that endpoint accepts only GET.

delete : $handler \to endpoint \to endpoint$ Connects an endpoint to a handler and ensures that endpoint accepts only DELETE.

put : $handler \to endpoint \to endpoint$ Connects an endpoint to a handler and ensures that endpoint accepts only PUT.

post : $handler \to endpoint \to endpoint$ Connects an endpoint to a handler and ensures that endpoint accepts only POST.

Example 3.6.1. A RESTful api User manages user data for a server at the path `/user`. It has the handlers $getUser : id \to list(User)$, $postUser : User \to result$, $deleteUser : id \to result$. The user endpoint then becomes

```
1 userSpec = is("user") ↣ oneOf(
2     [int ↣ contentType([(Json,encodeJson)]) ↣ get(getHandler),
3     ,accept([(Json,decodeJson)]) ↣ post(postHandler),
4     ,int ↣ delete(deleteHandler),
5     ])
```

Chapter 4

Method

To evalute if the functional approach to creating servers is more maintainable than existing solutions, a comparative study will be done. A popular library for developing server applications is by using an unopiniated solution using Express, which is a good candidate to compare to Cause.

Express is an unopiniated server framework written for Node.js for Javascript. That a framework is unopiniated means that it does not force you to architecture your code in any specific way.

4.1 Constructing the server in Express and Cause

To measure the maintainability, a comparison will be made by comparing a correct construction of an idiomatic server both made in Cause and the popular framework for Node Express. They will feature similar functionality which is a library api with the endpoints:

- `GET ‘‘api/books?released=int&author=string’’` Get a list of books and optionally ask for a specific author or a book from a specific year
- `DELETE ‘‘api/books/:id’’` Delete a book with a specified ID.
- `POST ‘‘api/books/:id’’ OR ‘‘api/books/’’` Create a new book or override a specific book

The server will also make use of a hashmap for database connection.

The accepted content types will be *application/json* and *www-url-formencoded* for all endpoints. Both implementations will handle all of the error cases. They will

also be written in an idiomatic way, that is they will not take the challenges outlined in Chapter 2 into consideration; the only requirement is that they compile.

4.2 Evaluating maintainability

The aspects that to be evaluated when measuring maintainability were discussed in Chapter 2. This study will focus on evaluating the following criterias:

- Testability
- Extendability
- Readability
- Error-proneness

Each of them require a different approach for evaluation.

4.2.1 Evaluating testability

The testability of the code is determined by it's aherence to SOLID principles, in particular of it's use of inversion of control. If a solution has less dependencies, it becomes easier to use unit tests to test the code and less resources are needed to create integration tests and E2E-tests.

Evaluation can be done then by counting the amount of mocked dependencies in the imperative solution and the functional solution.

4.2.2 Evaluating error-proneness and extendability

To evaluate error-proneness and extendability an expert analysis will be used. Cognitive Dimensions is a framework for evaluating the usability of programming languages and to find areas of improvements. [10] It allows us to evaluate the quality of a design and explore what future improvements could be made. As part of the Cognitive Dimensions, 14 different Cognitive Dimensions of Notation exist. A notation depends on the specific context, in this case the notation is the languages themselves and their architecture. The author of the framework recommends omitting the dimensions that are not applicable to the notation. This framework is used to evaluate the safety, error-proneness and extendability of both the Express and Cause solution.

Viscosity How much work does it take to make small changes? How easy is the code to refactor? If small changes requires consequent adjustments then that is a problem. As a viscous system cause a lot more work for the user and break the line of thought.

Visibility How easy is it to navigate the source code to find the parts that you want?

Hidden dependencies Are there hidden dependencies in the source code. Does a change in one part of the source code lead to unexpected consequences in another part of the code. Every dependency that matters to the user should be accessible in both directions.

Role-expressiveness How obvious is each sub-component of the source code to the solution as a whole?

Abstraction What are the levels of abstraction in the source code? Can the details be encapsulated?

Secondary notation Are there any extra information being conveyed to the user in the source code?

Closeness of mapping By looking at the source code, how close do we find it to be to the case we are solving?

Consistency How much of the rest can the user guess

Diffuseness or terseness How much space and symbols does the source code need to produce a certain result or express a meaning?

Hard mental operations Where does the hard mental processing lie? Is it more when writing the source code itself rather than solving the case, I.E. the semantic level? Does one sometimes need to resort to pen and paper to keep track of what is happening?

Provisionality How easy is it to get feedback of something before you have completed the entire system?

Progressive evaluation How obvious the role of each component of the source code in the solution as a whole?

Error proneness To what extent does the programming paradigm and language help minimise errors? Are there any structures or complexities that lead to it being easier to make errors?

For this study we will investigate the following dimensions:

- Diffuseness or terseness
- Closeness of mapping
- Hard Mental Operations
- Visibility
- Hidden dependencies
- Abstraction
- Error-proneness

We omit the other dimensions as related work concluded that the other dimensions did not bring much weight when evaluating the different paradigms. [11]

These aspects can also give us insights in the other aspects of mainatainability and will be used for discussion and evaluation.

4.2.3 Evaluating readability through code reviews

Code reviews, also known as peer reviews, is an activity where a human evaluates the program to check for defects, finding better solutions and find readability aspects.

To measure the readability of the REST library, a semi-structured code review is conducted on five different people with varying knowledge of REST apis and functional programming.

Semi-structured interviews

Semi-structured interviews diverges from a structured interview which has a set amount of questions. In a semi-structured interview the interview is open and allows for new ideas to enter the discussion.

Semi-structured interviews are used to gather focused qualitative data. It is useful for finding specific insights in regards to the readability of the code and provides insights as to wether or not the code can actually be understood by the general user.

To conduct an semi-structured interview, the interview should avoid leading questions and use open-ended questions to get descriptive answers rather than yes or no answers.

The questions that will be asked are presented below.

- What is your experience with Node and REST
- After being presented the code api, can you explain what it does?
- Which media types does the endpoint accept?
- Which media types representations can the endpoint show?
- Can you demonstrate how you would extend the api and add a new endpoint for a PUT request.

4.2.4 Evaluating the answers

After performing the interviews conclusions can be made by having the author interprate the answers to conclude if the code is readable or not. If the code is readable the users being interviewed should be able to explain to the author what the code does.

In order to reduce the bias in the experiments each user will be shown a different code base first. So the 3 users will be shown the implementation in ReasonML and 2 users will be shown the implementation in Express.

So in summary, the way each aspect of maintainability will be evaluated in both solutions by the following:

Testability Evaulated by comparing the number of dependencies that need to be mocked.

Extendability Evaluated by analysing the results from cognitive dimensions and the interviews.

Readability Evaluated by merging the results of cognitive dimensions and the interviews.

Error-proneness Evaluated by analysing the results from cognitive dimensions and the interviews.

Afterwards from there a discussion can be had about the strenghts and weaknesses of both solutions and the impacts of maintainability by using functional programming for developing REST servers.

Chapter 5

Results

Once these programs are constructed then cyclomatic complexity can be evaluated by summing the cyclomatic complexity of each function. We can also evaluate the cognitive complexity using the cognitive dimensions framework. Thus we get can see if smaller subproblems in big applications require more tests and have a bigger mental complexity depending on the different paradigms.

5.1 Limitations

TODO

5.1.1 Improvements to implementation

TODO

5.2 future work

5.2.1 Relations to cardinality

TODO

Bibliography

[1] T. Ishida, Y. Sasaki, and Y. Fukuhara, "Use of procedural programming languages for controlling production systems," in *[1991] Proceedings. The Seventh IEEE Conference on Artificial Intelligence Application*, vol. i, pp. 71–75, Feb 1991.

[2] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 1937.

[3] G. S. G. B. A. B. James Gosling, Bill Joy, "The java language specification," Febuary 2015. `https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf`, accessed Feb 2018.

[4] D. A. Turner, "Some history of functional programming languages," in *Proceedings of the 2012 Conference on Trends in Functional Programming - Volume 7829*, TFP 2012, (New York, NY, USA), pp. 1–20, Springer-Verlag New York, Inc., 2013.

[5] B. J. Copeland, "The church-turing thesis," in *The Stanford Encyclopedia of Philosophy* (E. N. Zalta, ed.), Metaphysics Research Lab, Stanford University, winter 2017 ed., 2017.

[6] R. S. Pressman, *Software Engineering: A Practitioner's Approach*. New York, NY, USA: McGraw-Hill, Inc., 6 ed., 2005.

[7] M. C. Robert, "Design principles and design patterns," September 2015. originally `objectmentor.com`, archived at `https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf`, accessed Feb 2018.

[8] "Pure function," Aug 2018. `https://en.wikipedia.org/wiki/Pure_function`, accessed Feb 2019.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[10] T. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131 – 174, 1996.

[11] E. Kiss, "Comparison of object-oriented and functional programming for gui development," master's thesis, Leibniz Universität Hannover, August 2014.