Evaluating Functional Programming for Quality 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
August 3, 2019

Abstract

Defects in Software engineering are a common occurrence. To mitigate defects the developers must create maintainable solutions. A maintainable solution is readable, extendable, not error-prone and testable. In order to make them so developers follow a guideline called SOLID principles. These principles are not enforced by the language but relies on the diligence of the developers, meaning there is nothing stopping them from writing unmaintainable code. In this study we translate these principles to Functional programming to investigate if Functional programming can be used to construct a library for servers that forces the developer to create correct code without incurring costs in maintainable and readability.

Acknowledgements

I want to thank my Anders Broberg for being my supervisor and Ola Ringdahl for his support and being quick to respond on email. I also want to thank all those who participated in the interviews.

Contents

Chapter 1

Introduction

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. Functional programming originates from 1936 from Lambda calculus [1] and even though functional programming is old, the industry most commonly use Object-oriented, imperative, languages such as Java. As of today, defects in software are still common place with the average defect rate being 15- 50 per 10000 lines of code. [2] This indicates that the tools used might be inefficient and improvements can be made. Also with the nature of defects being so common engineers partly need new tools that decrease defects but also need to ensure that the for future developers the code is easy to modify so that when defects show up these can easily be fixed. The software needs to be maintainble to be of good 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. [3] In order to create tests, the engineer employs various patterns and tools in the code to make the code easier to test, such as Object-oriented programming. They can also use static analysis and logical proofs to ensure correctness.

Software structural quality refers to how well the software adheres to non-functional requirements such as robustness and maintainability. [3] 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.

In this thesis, the focus will be on Functional programming as a potential solution to increase software quality. 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 aim is to investigate what makes software maintainable and can those criterias that make software maintainable be enforced and with Functional programming. Since software engineering is so general the focus will be especially on servers. In servers, it is common to use a protocol called REST to establish communication between servers and clients. These servers are called RESTful APIs and will be explained further in Chapter 2. As of today, it is up to the developer to manually ensure that this protocol is followed. Nothing enforces the developer to do it correctly unless the library implementation enforces it. Popular frameworks, such as Express, however do not enforce this protocol.

This thesis will show that it is possible using Functional programming to implement a server library that forces the user to implement a REST compliant server which would aid developers in improving the software functional quality. However functional programming is not commonplace and using such a library might have a negative impact when it comes to software maintenance and readability, thus impacting the software structural quality. By establishing the rules in Object-oriented programming for maintainable software and translating those to functional programming and by evaluating the readability of code written using the library, the study aims to find how software structural quality is affected through the use of functional programming.

Chapter 2

Background

This chapter aims to introduce us to REST servers and what are construction concerns when making them. Afterwards it will establish common practice in producing large scale software to ensure good software structural quality.

2.1 Introduction to REST servers

As mentioned in Chapter 1, servers need some protocol to communicate with the clients. One such protocol is REST. [4] Servers are applications that provide functionality for other programs or devices, called clients. [4] 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:

```
/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. [5] 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. Thus there are different types of tests that are conducted: unit tests, integration tests and E2E-tests.

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. (ADD_REFERENCE UNIT TEST) 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.

Since unit tests are the easiest form of testing, the developer should attempt to write code in such a way that it can unit test most of the code and not need to resort the upcoming test methods.

2.2.2 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.

2.2.3 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.

2.2.4 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. It also requires the diligence of the developer to enforce that code is written in units and that separation of logic and environment is done as otherwise E2E-tests and integration-tests need to be used.

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.

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 Section 2.2.4, such as making it hard to conduct unit tests. Thus Martin Robert states that 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 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. [6]

Using a SOLID architecture helps making programs that are not as dependent on the environments which makes them easier to test (swapping the production environment to a test environment becomes trivial). When investigating the testability, an important factor is that programs are written in such a way that all parts are easy to test. SOLID principles also helps ensuring that programs are extendable with Interface segregation principle, Open/Closed principle and Liskov substitution principle. Thus choosing a SOLID architecture for programs will allow making more testable software. These concepts were however designed for Object-oriented programming. In Chapter 4, these principles will be translated for Functional programming.

2.4 Functional programming for correct constructions

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. (ADD_REFERENCE type systems)

It is possible to combine the type system with design patterns to force the developer to create the right thing. Chapter 3 will introduce a REST framework named Cause, which has been created to force the developer to create REST compliant servers using Functional programming.

However as the REST library we in introduce in Chapter 3 makes heavy use of functional programming, a software paradigm that is not popular compared to paradigms such as Object-oriented programming (ADD_REFERENCE statistics), the hypothesis of this thesis is that readability might be affected. 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. Thus readability of software is an important criteria for good software structural quality.

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.

2.5 Conclusion

In summary there are four pillars of concern in maintainable software this thesis aims to address.

Testability Due to rotten design.

Extendability Due to rotten design.

Readability Multiple factors, this thesis will specifically look at inexperience as a factor of readability.

Error-proneness Due to rotten design.

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. Adherence to SOLID principles can be used to avoid rotten design. Thus a good REST library should be understood by other programmers and follow SOLID principles to good software structural quality. The REST library should also force coherence with REST protocol to have a good software functional quality. If the functional software can enforce that SOLID principles are followed through the type system, it would mean that a maintainable architecture would be forced by construction.

Chapter 3

Theory

Based on the challenges outlined in Chapter 2, the goal now becomes to construct a library for REST apis that is compliant by construction to ensure software structural quality, introduced in Chapter 1. This chapter will introduce the fundamentals of functional programming to then move on and use that to construct a server library which can be used to produce REST compliant servers.

3.1 Concepts from Functional Programming

While different definitions exist of what Functional programming means, here functional programming is a paradigm that uses of pure functions, decoupling state from logic 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 $g(x) = \mathit{currenttime} \cdot x$, $g(5)$ 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. [7]

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. Mutable data is an easy source of bug because it can cause two different functions to modify the same value, leading to unexpected results.

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.

Partial Application It is possible in functional languages to *partially apply* a function, meaning that we only supply some of the functions arguments, which yields a function instead of a value. For example, given a function $sumab = a + b$, we can partially apply this function to create a function $add3a = sum3a$.

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[1], Dialogs[2] or uniqueness types[3], 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.[4].

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.

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 for constructing a maintainable rest library.

[1]Read more: `en.wikipedia.org/wiki/Functional_reactive_programming`

[2]Read more: `stackoverflow.com/questions/17002119/haskell-pre-monadic-i-o`

[3]Read more: `https://en.wikipedia.org/wiki/Clean_(programming_language)`

[4]This is simplified as Monads are notoriously difficult to explain.

3.1.1 Functors and Contravariant Functors

The Functor type class 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 3.1.2. 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 $Predicate\ a = a \to Boolean$, 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 said to be in negative position and the type is *contravariant*. When the type parameter is the output of a function, it is said to be in positive position and the type is covariant. A type can be a Functor only if it is covariant.

Contravariant Functor type class 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.

3.1.2 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*, *Algebraic data type (ADT)* 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.

3.1.3 Domain-specific languages

A domain specific language (DSL) is a programming language specified for a specific domain. Typical examples of DSLs are HTML for designing web pages and SQL for making database calls. An eDSL is such a language embedded within the syntax of the language. EDSLs are useful due to the ability to separate the evaluation of the logic of the program to the logic itself. In the case of REST apis, this means we can develop an eDSL which can interpret REST apis and use them for different purposes. Since the logic is separated from the evaluation it also gives the advantage that testing is required for the evaluator and not for the logic itself (since testing it means just testing that it's equal to itself). To test evaluators can be done in separation using property-based testing (ADD_REFERENCE testing the hard stuff John hughes), thus allowing the developer to ensure that any program written using the GADT is correct by extension.

3.1.4 Generalized algebraic data type

One method for constructing eDSLs in functional programming is through the use of *generalized algebraic data type* (GADT).(ADD_REFERENCE GADT) They specify, depending on the input, what the output should be of that type. GADT enables implementing *domain-specific languages* (DSL) easily and ensure that the DSL is statically correct.

```
type Calculator
    Number : Int  -> Calculator Int
    Bool : Bool -> Calculator Bool
    Add : Calculator Int -> Calculator Int -> Calculator Int
    Multiply : Calculator Int -> Calculator Int -> Calculator Int
    Equal : Calculator a -> Calculator a -> Calculator Bool
```

Figure 3.1: A Calculator GADT with three operations add, eq and multiply.

```
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 demonstrates a minimal example GADT for a calculator. The calculator has five constructors: *Number*, *Bool*, *Equal Add* and *Multiply*. From this we can construct mathematical expressions and ensure that they are correct by constructions, or else they will not compile. If we attempt to construct an expression *Add* (*Bool False*) (*Number* 5) the compilation will fail as *Multiply* expects a number or an expression. However only having the expression is not very useful without some way of evaluating it. In Figure 3.3 we demonstrate how we can evaluate the expression using pattern matching.

```
evaluate : forall a. Calculator a -> Int
evaluate (Add expr1 expr2) = evaluate expr1 + evaluate expr2
evaluate (Multiply expr1 expr2) = evaluate expr1 + evaluate expr2
evaluate (Equal expr1 expr2) = (evaluate expr1) == (evaluate expr2)
evaluate (Number i) = i
evaluate (Bool b) = b
```

Figure 3.3: Evaluator for the calculator

Another example of GADTs is the creation of type safe lists, where we can be sure statically that performing *head* will yield an answer. This is demonstrated in Figure 3.4

```
type Empty
type NonEmpty

type SafeList a b =
     Nil : SafeList a Empty
     Cons : a -> SafeList a b -> SafeList a NonEmpty

safeHead : SafeList a NonEmpty -> a
safeHead (Cons x _) = x
```

Figure 3.4: Type safe list

By separating how the expression from it's evaluation, the expression can be reused for different purposes. For instance it would be possible to use the same logic for the calculators and implement them for different platforms and ensure statically that all platforms follow the same logic. So GADTs are useful for creating expressions that can later on be evaluated.

3.2 Servers using GADTs: Router

Using GADTs, we can construct a statically correct eDSL for server routers.(ADD_REFERENCE code from https://github.com/elm/package.elm-lang.org/blob/master/src/backend/Server/Router.hs) A server router parses incoming requests and extracts query parameters and parameters and executes a function depending based on the result. We first define a minimal example of a GADT *Router*, in Figure 3.5. The two constructors *Top* and *Exact* describe matching `/` or for a given string s, `/s` respectively. It should also be possible to link two *Router* together to allow us to match nested urls. Thus we can introduce *Compose* which allows composing routers together composed of multiple parts. For instance `/hello/world` corresponds to *Compose* (*Exact* "*hello*") (*Exact* "*world*").

```
type Router
    Top : Router
    Exact : String -> Router
    Compose : Router -> Router -> Router
```

Figure 3.5: Minimal router GADT

Definition in Figure 3.5 is not sufficient as a route can also contain parameters, such as integers for id or strings. These parameters need to be applied to a function which can handle them. We can use GADTs to also describe the transformation of the functions argument from some start parameter to some end parameter. In Figure 3.6 we extend the router with the constructors Integer and String. When evaluated later, Integer and String describe that they transform a function which takes an argument (*String* or *Int*) and returns result. *Compose* allows us to compose these parameters together, so an api for a user resource could be implemented as `Compose (Compose (Exact "user") String) Integer`, which could be interpreted to match on urls formatted as `/users/:string/:int` where `:string` is a valid string and `:int` is a valid int. These arguments get applied to the handler, thus the type of this value becomes `Router (String -> Int -> a) a`.

```
type Router start result
    Top : Router start result
    Exact : String -> Router start result
    Integer : Router (Int -> result) result
    String : Router (String -> result) result
    Compose : Router a b -> Router b c -> Router a c
```

Figure 3.6: Router GADT extended with Int and string

So *Router* can describe a specification for what the type signature of the handler must be and what it must then produce. Finally there needs to be a way to apply that handler to the arguments so that it can produce that value, so we add a constructor to *Router*, `Produce : function -> Router function output -> Router (output -> c) c`. So the final GADT for the router eDSL becomes the one in Figure 3.7. `Router (output -> c) c` informally translates to "give me something that can transform the output to c and I will give you c".

```
type Router start result
    Top : Router start result
    Exact : String -> Router start result
    Integer : Router (Int -> result) result
    String : Router (String -> result) result
    Compose : Router a b -> Router b c -> Router a c
    Produce : function -> Router function output -> Router (output
       -> c) c
```

Figure 3.7: Router GADT extended with Int and string

This section has demonstrated how GADTs are useful for constructing a eDSL for routers. What was not described is how to interpret the router which is omitted for brevity. However the source code for the final solution can be found in Appendix. Furthermore we will extend this functionality to implement all of the functionality of a REST server.

3.3 Functional servers

A *Server* is a type function *Request* $\rightarrow$ *Response*. Simply, given some Request it should produce some Response where Request is a product of the url, media type, accept header, content type header and a body. The Response is a product of status code, a set of headers where headers are a tuple of strings, a content type and an encoding. If the Response returns a successful code in the range of 2XX, 3XX we say it is a successful response. The goal is to ensure that values of *Server* are following the REST API specifications. Thus we define a function *make* that can construct a value of *Server* which follow REST specifications.

Based on the REST api description outlined in chapter 2, we define a type $Specification$ *input output*, which is a GADT for specifying how to transform input into output. We define $Specification$ *Request Response* to mean a *server specification*, since it defines how to turn a *Request* (input) into a *Response* (output). $Specification$ works as DSL for one or more *endpoints* within a REST api where an endpoint is one of the following:

- A correct URI to access the resource, such as `api/user:int`. While the final implementations support any type, this implementation will only support integers and strings for brevity.
- URI should also support query parameters as optional parameters for the handler defined below.
- A set of content types that it can represent the resource as.
- A set of content type representations for the resource that request can submit which can be parsed by the server.
- A map of the query parameter name and their respective parser, where parser is a function of type $string \rightarrow Maybe\ a$)
- An HTTP verb
- Status code on success
- A function, called a handler, which takes some parameters and returns either the resource or a failure message and failure code. This is used for side effects, for instance database access. We model the result as a sum type $Result\ a = Ok\ a \mid Fail\ Message\ Code$.
- A combination of one of these endpoints.

A difference from the *Router* defined in previous section is that each Endpoint need to produce *two* intermediate values. One value which is the handler and another value which is how to encode the result of the handler. We define the encoder as ResponseBuilder, which is a contravariant of type $a- > encoded$. Thus we modify the constructer `Exact : String -> Specification input output` to `Exact : String -> Specification (handler, responseBuilder) (handler, responseBuilder)`. The constructor `Integer` becomes `Integer : Specification (int -> a, r) (a, r)`.

We also introduce a few new constructor, one being `Accept` which takes a list of encoders and their corresponding accept header (`(Mediatype, [r -> encoded])`) and produces a `Specification (a, encoded) (a, r)`, or with responsebuilder's definition expanded it produces `Specification (a, encoded) (a, r -> encoded)` for clarity. When parsing a request, it can then check what available media types the endpoint can represent and pick the appropriate encoder without the programming needing to write it manually. The final GADT for specifications become as follows:

```
type Specification input output =
  Exact : String -> Specification (h, r) (h,r)
  QueryParam : string
          -> (string -> Maybe a)
          -> Specification (Maybe a -> b, r) (b, r)
  Slash : Specification (a, b) (c, d)
          -> Specification (c, d) (h, r)
          -> Specification (a, b) (h, r)
  IntegerParam : Specification (int -> a, r) (a, r)
  Verb : HttpMethod -> Specification (h, r) (h, r)
  Accept : [(MediaType, r -> encoded)] -> Specification (a,
      encoded) (a, r)
  ContentType : [((MediaType, string -> Maybe body)]
                -> Specification (body -> b, r) (b, r)
  Handler : StatusCode
      -> handlerFunction
      -> Specification (handlerFunction, noEncoder) (Result
         resource, resource)
      -> Specification Request (Maybe Response)
  Many :
        [Specification
          Request (Result Response)
        ] -> Specification Request Response
```

The constructor handler now takes as a parameter a `Specification (handlerFunction, noEncoder) (Result c, c)`, which describes that given a handler function that correctly handles the parameter values and a responseBuilder without an encoder, can produce a tuple of the resulting resource as well as an encoder of that resource to the appropriate media accept header. We enforce statically that the handlerFunction produces something which might fail (*Result resource*), so that we can automatically handle that error and throw the appropriate response.

Notice that the Many case is for transforming many endpoints into a single one. Also notice that Handler produces a $Specification\ Request\ (ResultResponse)$. If the parsing fails we want it to return Nothing so that $Many$ has a way of knowing if an endpoint failed to parse the request. If all endpoints fails the yield from applying a Request to a specification interpreter should be 404 - Not found.

From the $Specification$ GADT we can define a function $make : Specification\ Request\ Response \rightarrow Request \rightarrow Response$. $make$ works by evaluating the GADT to deduce how the request should be parsed. Since Functional programming is being used, then given a $Specification\ Request\ Response$ called s, we can partially apply the function $make$ to produce a Server, since Server is *type equal* to the type of $make\ s$. The full implementation of $make$, implemented in the functional programming language ReasonML, is available in the appendix.

3.4 Using the library

The library exposes the GADT specification and a set of functions that can be used to create specifications. Following is a minimalistic example of an endpoint:

$$spec = GET \triangleright\ Path.is\ "echo" \triangleright\ Path.takeText$$
$$echo = endpoint\ (\lambda s \rightarrow Ok\ s)\ Ok200\ spec$$

$echo$ is a $Server$ which "echoes" back the message that is entered on the URL `/echo/`, so `/echo/helloworld` would yield a response with the body "helloworld". It does this by using the function endpoint, which takes the handler, a status code on success and a specification. Specifications are combined using the ($\triangleright$) operator, which is implemented as $(\triangleright)\ a\ b = Compose\ a\ b$.

3.4.1 Defining an endpoint

The verb of the endpoint is set by using one of the functions `GET,POST,DELETE,PATCH`. This will make it so the endpoint only matches those requests containing the same verb as specified. Three operations exist for parsing URIs which are $Uri.is : String \rightarrow Specification$, $Uri.takeText : Specification$ and $Uri.takeInt : Specification$. $Uri.is$ parses exactly the given string and $e = Uri.takeInt$ will parse an integer from the path. These can be combined so $e = Uri.is\ "api" \triangleright\ Uri.is\ "user"\ \triangleright\ Uri.takeInt$ would parse `api/user/5` and extract 5 as a parameter which it applies to the handler.

Accept headers can be set using the *accept* function, which takes a list of tuples with the first element being an encoder and the second being it's associated content media representation. So the specification can use different encoders depending on the accept header of the request.

Content type headers can be set using the *contentType* function, which takes a list of tuples with the first element being a *decoder* and the second being it's associated content media representation. This way it can check what media representation the request content has and decode it and afterwards apply that to the handler.

Query parameters can be set using the *query* function. Query takes the name of the parameter and a decoder to use. The result of the decoder will be applied in the handler.

A get endpoint to a book API

Using these combinator functions defined earlier, we can define a server to access books. In this example we demonstrate how to create an endpoint that also uses the query parameters author, specifying the name of the author of the books we want to access; released, specifying the year we want the books to be published; as well as how to accept multiple accept headers (json and plain):

```
spec =
    GET
    |> uri Path.is "api" |> Path.is "books"
    |> query "author" (\name -> Just name)
    |> query "released" intFromString
    |> accept [
         (json, Encoders.jsonList),
         (plain, Encoders.plainList),
       ]
get = endpoint getFromDatabase Ok200 spec
```

In this example, `getFromDatabase`, is deduced from the specification to be a function with the signature `Maybe string -> Maybe int -> Result [book]`. Encoders are functions of type `book -> encoded`. We notice how the handler does not need to work anything with how to encode/decode books and only needs to handle the error of failure to fetch the resource, thus separating concerns.

3.5 Conclusion

In this chapter we have introduced the concepts of functional programming and then from those concepts created a library that can be used to construct REST specifications as a GADT. We have then demonstrated how a value of specification can be used to construct REST compliant server application, I.E. we constructed a function $make : Specification\ Request\ Response \rightarrow Request \rightarrow Response$.

Chapter 4

Method

Armed with the new library constructed in Chapter 3, which increases software functional quality, the aim now is to see how software structural quality is impacted. To evaluate 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.(ADD_REFERENCE UNOPINIATED)

An idiomatic server will be made in both 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 that is abstracted away in the implementation The accepted content types will be `application/json` and `www-url-formencoded` for all endpoints and the displayable content-types are `text/plain` and `application/json`. 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.

4.1 Evaluating maintainability

The aspects that to be evaluated when measuring maintainability were discussed in Chapter 2. To recap the important aspects were:

- Testability
- Extendability
- Readability
- Error-proneness

Chapter 2 established that the SOLID principles can be used to as guidelines for creating maintainable software. Those principles will therefore be used as criteria that Cause should be evaulated against. However these guidelines do not state anything about the readability of the software. Thus two different methods will be used to measure readability and to measure the testability, extendability and error-proneness.

4.2 SOLID principles in Functional programming

To evaluate the testability, error-proneness and extendability Deductive reasoning can be used to find if the solution follows each criteria. The SOLID guidelines were written for Object-oriented programming. The principles are however general and can be redefined for Functional programming as presented below.

4.2.1 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 then the responsibility of a file is to morph that type into some other value. Thus it keeps the modules focused and simple and would ensure that the single responsibility principle is held. It can also be thought of as "One function modifies one thing". So in summary, a program follows the Single Responsibility Principle if

1. Each functions performs only a morphism, which is guaranteed if the function is pure.

2. The file does not contain functions that do not contain any of the types declared within that file. However, this rule has an exception for functions that are only used by the other functions within that module (called a helper function). Helper functions can be merged into the function that uses it but we choose to split them up for readability purposes.

4.2.2 Liskov Substitution Principle

Liskov's Substitution Principle states how reasoning about subtyping among objects should be done. If S is a subtype of T, then the subtype relation mean that any term S can be safely used in a context where type T is expected. Since subtypes do not exist in classic functional programming some translation is needed. 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.

In functional programming, Liskov Substitution Principle is simply Contravariant Functors. In order to comply with the principle, argument types overriding a method must be contravariant and the reverse should be true for the return type, it should be covariant. A contravariant type can only be overridden by using *contramap*. And it's result is naturally in positive position hence it's covariant.

4.2.3 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 4.1, this program depends on the computer IO, making it difficult to extend it to different environments, such as databases.

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

Figure 4.1: A program that reads input from the computer and then prints it.

Instead, Figure 4.2 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.

```
readNPrint : (IO String) -> (String -> IO ()) -> IO ()
readNPrint reader printer = reader >>= printer

-- and then later
consoleIO : IO ()
consoleIO = readNPrint readLine putStrLn
```

Figure 4.2: A program that reads input from the computer and then prints it, where the logic is separated from it's environment.

This way, the dependencies can be mocked and replaced with different ones. So if we later want to create a *applicationIO* we can reuse *readNPrint* with the functions for printing in the application and reading input from the application.

Now for a REST api library, it means that the logic should not depend on it's environment means that the specification of the REST api should not depend on the server implementation. In other words, it should be trivial to port the server logic to another runtime if needed. To do this, GADTs can be used to separate the expression from it's evaluation. So the REST api is simply described as instructions of a GADT.

4.2.4 Interface Segregation Principle

Interface Segragation Principle states that no client should be forced to depend on methods it does not use. This translates to, in Functional programming, that the smallest set of data should be used for each function to work. Recall earlier that types can be thought of as sets. Recall also that the cardinality of a set is the amount of possible values that set can have. If the cardinality of a type is higher than expected it allows introducing illegal states. For example, $type\ Color = Blue|Green|Red$ has a cardinality of 3 (since it can either be Blue, Green or Red) whereas Fig. 4.3 has a cardinality of $2 \cdot 2 \cdot 2 = 8$ meaning that it has 5 states that are impossible! By choosing the right data structure it lowers the amount of possible values that are possible. So Interface Segragation Principle in Functional programming states that a function should not be able to produce values it does not use.

```
type Color = { Blue: Bool, Red: Bool, Green: Bool}
```

Figure 4.3: Product type Color with cardinality too high

Observe that in Fig 4.4, the type `IUserRepo` has two operations, one which is not needed by the `getUserEndpoint` (why would that function need to storeUser?). Thus the function is capable of having more values than it should be. This means it breaks the Interface segregation principle.

```
data IUserRepo =  {
        getUser : Id -> IO User,
        storeUser : User -> Id -> IO ()
}

-- Later on
getUserEndpoint : IUserRepo -> Request -> Response
-- ...
```

Figure 4.4: Normal interface for operations

So in summary, adherence to interface segregation principle means that the cardinality of the types are minimized.

4.2.5 Open/Closed principles

Open/Closed principle states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

The OCP is an advice on how to write modules in such a way that we have backwards compatibility and so that if extra functionality is needed, the modifier does not need to look at the class in order to make modifications. So if a class has some new requirements you do not need to modify the source code but can instead extend the superclass.

When this principle is applied into Functional programming, we run into the expression problem. The expression problem states that *"The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code and while retaining static type safety (e.g., no casts)."*

(ADD_REFERENCE http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt)

The similarity with expression problem and OCP is that you want to be able to extend the program (add new cases to the datatype) without recompiling existing code. Object-oriented programming uses classes that should be open for extension and closed for modification. In functional programming, when this principle is applied, new cases to datatype should be possible and new functions. OCP exists because modifying battle tested code is dangerous and might cause regressions. Thus a preferable solution is to extend the previous code instead.

Example: Creating a OCP compliant paint programming

In a paint program, various shapes should be possible to paint: circles, squares, stars and custom shapes. It should also have a custom menu depending on the shape, a circle should be able to set the radius, a square the area and stars the diameter.

A functional approach is to create a sum type of the shape, seen in Fig 4.5. To add more shapes, the original source code would need to be modified. This means that Open/Closed principle is not being followed. It can cause a lot of trouble down the line, one function is acceptable but what if we had thousands of functions that depended on shape. Adding one shape would mean changing thousands of lines of code scattered all over the place.

```
type Shape
  = Star size
  | Custom [vector]
  | Circle radius curvature
  | Square size

render : Shape -> IO ()
render shape =
        case shape of
                Star size = Star.render(size)
                ...

-- Do the same thing
renderMenu : Shape -> IO ()
renderMenu = ...
```

Figure 4.5: A sum type of shapes

A different approach is by using type classes and contravariance. In order to render shapes, there needs some general format which we can use to render them. Let's assume we have some function $render : Set\ Vector \rightarrow IO\ ()$ for rendering. This is great because we know that any shape can be represented as a set of vectors in the end. Let us define $type\ Renderable\ a = Renderable\ (a \rightarrow Set\ Vector)$. Now it becomes possible to define a render function $render : Renderable\ a \rightarrow a \rightarrow IO\ ()$, that works for all shapes. Shapes can be made in separation now by contramapping properties, seen in fig 4.6.

```
type Renderable a = Renderable (a -> Set Vector)
instance Contravariant a => Renderable a where
        contramap cf b = \a -> b $ cf a

type Circle = {radius: Int}
circle : Renderable Circle
circle = circleToVector . radius -- circleToVector turns it to vector

setRadiusFactor :  Int -> Renderable Circle -> Renderable Circle
setRadiusFactor factor = contramap ({radius = factor})

type Custom = {scale : Int, shape : Set Vector}
custom : Renderable Custom
custom = scale * shape

addVertex : Vector -> Renderable Custom -> Renderable Custom
addVertex vertex = contramap (Set.union vertex)
```

Figure 4.6: A contravariant approach to shapes

Contravariance forces adherence to a certain interface but leaves it open to extension, in spirit to the Open/Closed Principle. A separate part of the code can contain Renderable Square, Renderable Star without modifying the original code. Thus, in Functional programming, contravariance and type classes can enable OCP compliant code that solves the expression problem. So OCP compliant code is code that does not run into the expression problem.

4.2.6 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 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.

Q1 What is your experience with RESTful APIs?

Q2 What is your experience with Express?

Q3 What is your experience with ReasonML?

Q4 After being presented the code api, can you explain what it does?

Q5 Which media types does the endpoint post accept?

Q6 What is the uri of DELETE?

Q7 Which media types representations can the endpoint show?

Q8 Given a handler putInDatabase, Can you demonstrate how you would extend the api and add a new endpoint for a PUT request.

Q9 Looking at the javascript api, can you explain what it does?

Q10 Which media types does the endpoint get accept?

Q11 Which content type and accept does post have?

The interviewer will also be informed that the name of the file off the code is BookApi.re, re being the file extension of ReasonML, and BookApi.js, js being the file extension of javascript, respectively.

4.2.7 Evaluating the answers

After performing the interviews conclusions can be made by interpreting 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. This also shows gives insights as to the reason the users do not understand the solution.

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 comparing to SOLID principles.

Readability Evaluated by comparing to SOLID principles.

Error-proneness Evaluated by SOLID principles and the interviews where we ask to extend the solution with a PUT request.

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

Chapter 5

Results

Based on the specifications outlined in the introduction of Chapter 4 two server programs were constructed. The source code for the programs can be found in Appendix A.1 for the ReasonML implementation and in Appendix A.2 implementation.

5.1 Evaluating adherence to SOLID

We can through an expert analysis analyze the adherence to SOLID guidelines in Section 4.1 of the solution in ReasonML. It is the resulting code of using the library that is analyzed and not the library itself as the goal is to find if Functional programming constructs can be used to enforce an idiomatic solution. The solution was written by the author in a "as naive" approach as possible. That means that the author did not take any design guidelines into consideration but created the software in such a way that it would compile.

Single Responsibility Principle

Recall that the Single Responsibility Principle for functional programming states that all modules should revolve around one type. The file BookApi.re contains one product type Book. The modules Encoders and Decoders both use this type except for one helper function `int`. Afterwards we find that all the functions in the module make use of the book type with Lastly we find that the module Endpoint all revolve around the type `route (a)` and that they all make use book except the handler `delete`, which is a helper function for the function `router`.

Open/Closed Principle

OCP, as defined in Section 4.2.5, that the data structures should be open for extensions without modifying the previous code. The book api functionality can be extended with new endpoints without modifying any of the original code. The router is implemented as a list of endpoints, thus if the user wants to add a new endpoint it can append new endpoints to the list. However it is not possible to extend existing endpoints without modifying the code. For example should the user want to prepend so that each uri starts with `/new` then that is not possible, the existing code has to be modified.

Liskov Segregation Principle

Liskov Segregation Principle is not applicable to this solution.

Interface Segregation Principle

Interface segregation principle states that the cardinality should be as low as possible. While it is impossible to force the user to have the lowest cardinality possible the library encourages usage of the lowest possible cardinality by feeding the arguments into the handler and stating the return type. So in the BookApi.re that every $specification$ forces a contract on the function and states that to work they must take the specified arguments which it will extract from the request with the parser function. So it means that the cardinality of the handler must be according to the $specification$.

Dependency Inversion Principle

Dependency Inversion Principle is about separating the logic from it's environment. Since the $specification$ GADT separates the handler from the specification, it means that should the developer want to change the handler they can change the argument at one spot. If the developer should want to change the REST api library, handlers are separated from the $specfication$. Thus the developer would not need to change any of the logic of the handlers. Therefore the code follows the dependency inversion principle.

Also due to it's separation it also means that testing the logic of the api is easier. In order to test it you

5.1.1 Imperative solution

Since the solution was developed in a untyped language with no force of structure it makes sense that the SOLID principles will not be followed. But for the sake of the argument below we go through them as well.

Single Responsibility Principle The imperative solution breaks SRP in all handlers by having functions that both parse the requests and performs the side effects. Demonstrated in the first handler App.get

Open/Closed Principle N/A

Interface Segregation Principle N/A

Dependency Inversion Principle In the imperative solution, it is impossible to test the handler in isolation. All systems need to be emulated such as database and the router.

Liskov Segregation Principle N/A

5.2 Interviews

With the method outlined in the previous chapter, the interview was performed on 4 different people, which is a bit less than the recommended by Norman group of five people due to difficulty finding enough users (ADD_REFERENCE `https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/`). It was performed through the use of Skype, a communication tool[1]. The four respondents were graduated students of the Engineering Interaction Technology and Design programme at Umeå university. The programme is a five year degree that combines education in software engineering with studies in design. The questions were originally asked in swedish but translated to English by the author. The answers can be found in Table 5.1 and Table 5.2.

Person 1 and Person 4 understood correctly what the library does. However all four subjects where slightly confused as to what encoders and decoders were used for. Not one of the subjects could correctly guess what the accept and content types in the Javascript solutions. In Q5 for person 3 the question was omitted as the user could not correctly guess at all what the code was supposed to do and assumed that it a system for encrypting books.

[1]Skype's website: `https://www.skype.com`

	Person 1	Person 2
Q1	Implemented API that should follow REST. I assumet it's related to CRUD?	A little bit, REST is an API with endpoints containing method and headers ensuring you get the right data.
Q2	Little bit, it should be Javascript and Ocaml combined.	No experience
Q3	I've implemented an API in Express	I had a course where I used Express three years ago.
Q4	The title is BookApi.re which describes it quite well. It is a Book api that follows RESTful. It also manages encoding and decoding. It also checks that the requests are correctly formatted.	Encoders and Decoders extract data from the json and I understand the handler functions. However the module Endpoint is unclear. Especially $type\ a.\ route(a)$.
Q5	application json	Maybe string?
Q6	$api/books/ : int$	No clue
Q7	application/json	Content type? Json?
Q8	See Appendix B.0.1	See Appendix B.0.2
Q9	A book api	A Book REST api
Q10	text/plain and application/json	text/plain and application/json
Q11	Content type is json and Accept might be json?	Unsure

Table 5.1: Raw results interview one and two

	Person 3	**Person 4**
Q1	I know what it is. It specifies how to receive and send information to the client.	It is used for making HTTP requests and setting up a server using simple methods for changes.
Q2	I have seen it.	I have no experience but heard about it
Q3	It is my go to library for writing servers.	I have worked with it
Q4	Code to encode and decode so that people can not read the content of the books. Not sure what plain means in content type. Are modules objects?	First modules define encoding and decoding json data. Afterwards some helper functions. Lastly there are endpoints with router defined with different paths and query parameters. It is a REST api for adding, deleting and modifying books.
Q5	N / A	responds with json and accepts plain.
Q6	*/delete*	*/api/books.*
Q7	application/json	N/A
Q8	See Appendix B.0.3	See Appendix B.0.4
Q9	A book api to fetch books	A Book REST api
Q10	string	It can recieve text/plain and application/json but unsure what it can send
Q11	Application/json	Application/json but not specified what it accepts, assume plain/text.

Table 5.2: Raw results interview three and four

Chapter 6

Conclusion

In Section 2.5 we introduced the four pillars of maintainability that needed to be evaluted against the library. These were testability, extendability, readability and error-proneness. In order to do so we will review the results gathered in Chapter 5 and tie them together with the four pillars.

6.1 Evaluating the readability

We find that while everyone understood the Javascript version of the book api, two (P2 and P3) out of four faced difficulties with the ReasonML version. Encoders and decoders seemed to have confused P3, as they assumed it related to cryptography. P2 got confused by the type signature $type\ a.\ route(a)$, which is necessary for Ocaml to deduce the type signature as it otherwise can not generalize. It also seems that P4 was incapable of understanding the ReasonML version and assumed that endpoints were not functions but objects.

3 out of 4 users (P1, P2, P4) were able to extend the code with a new endpoint PUT and P3 was almost able to except that they used the wrong function for handlers.

Further research is needed to find how long it would take for the users to understand. We find that half of the users could understand the new code base without any form of introduction (P1 and P4). In production it might be valuable to find a more exact number how long it would. A future test could investigate if users understood the code after a short introduction. As even without introduction three out of four were still able to extend the code and two out of four were able to understand it.

The study indicates that there are costs in readability to the code for inexperienced users. It can be argued that these costs seem to be minor and that after a brief introduction the code would be understood. It can further be argued that some of those costs can be mitigated by adding comments and making the changes in Section 6.4, but more research is needed to prove this. For the manager, this becomes a question of costs vs benefits were they have to factor the increased cost in readability and possibly extension against the gains in ensured increases in testability.

6.2 Functional programming and SOLID

In this study we found that functional programming was capable of creating a library that was able aid in creating Single Responsibility principle, by encouraging the user to separate the REST specification from how the handler fetches the data. It also manages to enforce Interface Segregation Principle and Dependency Inversion Principle. This aids in reducing immobility, fragility and viscosity.

It was inconclusive as to if it enforces Liskov substitution principle as further work is needed to create examples where this principle is properly test. The functional solution also breaks Open/Closed principle for situations where the user wants to add more details to a specification.

6.3 Four pillars of concern

Recall that Chapter 2 introduced the four pillars of concern: testability, extendability, readability and error-proneness. From these studies we can conclude that

Testability Software became easier to test as it managed to invert the control so that unit tests can be made for testing specifications which in the imperative solution would require an integration test.

Extendability No gains are made in extendability when using the functional solution, in fact they might be a bit damaged by not properly following the OCP solution.

Error-proneness The functional solution marginally affects the error-proneness as we find that person 3 in Q8 was unable to extend the code without making an error.

Readability affected negatively since P3 and P2 were unable to comprehend the code.

Even though errors were made when extending the server with a PUT request. It might be mitigade by the static analysis featured in ReasonML which would have pointed out that there was an error.

In conclusion this thesis has demonstrated how software functional quality could be improved by the use of Functional programming by creating a library that enforces REST compliancy. We also demonstrated how it could also aid in certain software structural qualities and hurt in others such as readability.

6.4 Future work

SOLID principles places a lot of emphasis on extendability and that modifying original code is bad practice. It is questionable if these principles are relevant for strongly typed languages, as these indicate if there are any errors in the code, making it easier to refactor.

A lot of the errors in readability might be possible to fix and it might be that they are not inherenet to the language itself. However it would be necessary to make the changes below and then retry the experiment with five more people to find out if it is more comprehensible.

There is also more work to be done if Cause is to become useful for the industry. The inclusion streaming types would need to be implemented in the future.

6.4.1 Clarifying what is URL and what is not

Most users were uncertain what the URIs were with only Q1 correctly assuming that the URI for delete was $/api/books : id$. However it is possible to further clarify what is a URI by wrapping it in a function that takes an incomplete route. This might make it clearer for the user what the URL is but further research is needed.

6.4.2 Extendability for the URIs

There are some issues still with the server not being as extendable as possible for it to be OCP compliant. To extend it with the functionality of adding new details to an existing endpoint would make it more compliant as it would allow the user to add new functionality to code without recompiling the original code.

6.4.3 Functional programming for documentation

When creating software, engineers tend to also document the software for future use to make it more maintainable. In servers this is usually done manually. (ADD_REFERENCE) However if updates are made to the code, the engineer has to then also manually update the documentation which incurs maintenance costs. It is plausible that *specfication* can also be used for documentation. This should be as simple as creating an evaluator for the *specification* to generate the documentation. This would ensure that documentation stays in sync and minimizes maintenance costs of documentation by automating it.

6.5 Concluding remarks

I hope this thesis serves to underline the challenges in the software industry and demonstrate how the software industry can make use of functional programming to aid creating software that is maintainable in ways that are not possible in other paradigms. The techniques outlined in this thesis here can be applied to other protocols in other domains to ensure that certain restraints are held. My hope is in the future the software industry aims to write software as and specification combined rather than first writing the specification and then the software.

Bibliography

[1] 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.

[2] S. McConnell, *Code Complete, Second Edition.* Redmond, WA, USA: Microsoft Press, 2004.

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

[4] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, 2000. AAI9980887.

[5] L. Chen, "Microservices: Architecting for continuous delivery and devops," 03 2018.

[6] 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.

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

Appendices

Appendix A

Implementation

A.1 ReasonML REST implementation

```
type book = {
  title: string,
  author: string,
  year: int,
  id: int,
};

module Encoders = {
  let json = myBook => {
    open! Json.Encode;
    Json.Encode.(
      object_([
        ("title", string(myBook.title)),
        ("year", int(myBook.year)),
        ("author", string(myBook.author)),
        ("id", int(myBook.id)),
      ])
    );
  };
  let jsonList = Json.Encode.list(json);
  let plain = book => book.title;
  let plainList = books =>
    List.fold_right((book, str) => str ++ plain(book), books, "");
};
module Decoders = {
  let json = json => {
    open! Json.Decode;
```

```
    Json.Decode.{
      title: json |> field("title", string),
      year: json |> field("year", int),
      author: json |> field("author", string),
      id: json |> field("id", int),
    };
  };
  let safeJson = myBook =>
    try (Some(json(myBook))) {
    | Json.Decode.DecodeError(s) => None
    };
  let int = s =>
    try (Some(int_of_string(s))) {
    | Failure("int_of_string") => None
    };
  let jsonWithKey = Json.Decode.tuple2(Json.Decode.int, json);
};

let deleteFromDatabase = key => {
  let success = Database.delete(key);

  if (success) {
    Result.Ok("Deleted");
  } else {
    Result.Failed(
      "Entry does not exist",
      Status.BadRequest400,
      MediaType.Plain,
    );
  };
};

let replace = (key, book) => {
  let success = Database.replace(key, Encoders.json(book));
  if (success) {
    Result.Ok("Added");
  } else {
    Result.Failed("Database failure", Status.Error500, MediaType.
       Plain);
  };
};

let insert = book => {
  let success = Database.insert(Encoders.json(book));
  if (success) {
    Result.Ok("Added");
```

```
  } else {
    Result.Failed("Database failure", Status.Error500, MediaType.
       Plain);
  };
};

let getFromDatabase = (queryAuthor, queryReleased) => {
  let cmpAuthor = (author, book) => book.author == author;
  let cmpReleased = (released, book) => book.year == released;
  switch (queryAuthor, queryReleased) {
  | (Some(author), Some(released)) =>
    Result.Ok(
      Database.filter(Decoders.safeJson, book =>
        cmpAuthor(author, book) && cmpReleased(released, book)
      ),
    )
  | (Some(author), None) =>
    Result.Ok(Database.filter(Decoders.safeJson, cmpAuthor(author
       )))
  | (None, Some(released)) =>
    Result.Ok(Database.filter(Decoders.safeJson, cmpReleased(
       released)))
  | (None, None) => Result.Ok(Database.get(Decoders.safeJson))
  };
};

module Endpoint = {
  open Spec.Router;

  let replaceId: type a. route(a) =
    endpoint(
      ~handler=replace,
      ~success=Status.Ok200,
      ~spec=
        Method.post
        >- Path.is("api")
        >- Path.is("books")
        >- Path.takeInt
        >- contenttype([Spec.Accept.json(Decoders.json)]),
    );

  let post: type a. route(a) =
    endpoint(
      ~handler=insert,
      ~success=Status.Created201,
      ~spec=
```

```
        Method.post
        >- Path.is("api")
        >- Path.is("books")
        >- contenttype([Spec.Accept.json(Decoders.json)]),
    );

  let get: type a. route(a) =
    endpoint(
      ~handler=getFromDatabase,
      ~success=Status.Ok200,
      ~spec=
        Method.get
        >- Path.is("api")
        >- Path.is("books")
        >- Path.query(~parameter="author", ~decoder=s => Some(s))
        >- Path.query(~parameter="released", ~decoder=Decoders.
            int)
        >- accept([
             Spec.Contenttype.json(Encoders.jsonList),
             Spec.Contenttype.plain(Encoders.plainList),
           ]),
    );

  let delete: type a. route(a) =
    endpoint(
      ~handler=deleteFromDatabase,
      ~success=Status.Ok200,
      ~spec=
        Method.delete >- Path.is("api") >- Path.is("books") >-
            Path.takeInt,
    );
  let router = oneOf([get, post, replaceId, delete]);
};

Spec.listen(3000, Endpoint.router);
```

A.2 NodeJS REST implementation

```
const express = require("express");
const bodyParser = require("body-parser");
const app = express();
const database = require("database");
app.use(bodyParser.json()); // support json encoded bodies
app.use(bodyParser.urlencoded({ extended: true })); // support
    encoded bodies
```

```
const toString = book => book.author + " " + book.released + " "
    + book.name;

app.get("api/books", (req, res) => {
  const released = parseInt(req.query.released);
  const author = req.query.author;
  var books;
  if (released && author) {
    books = database.getByReleaseAndAuthor(released, author);
  } else if (released) {
    books = database.getByRelease(released);
  } else if (author) {
    books = database.getByAuthor(released);
  } else {
    books = database.get();
  }
  switch (req.header) {
    case "text/plain":
      res.send(books.map(toString).join(", "));
      break;

    case "application/json":
      res.send(books);
      break;

    default:
      res.status(405);
      res.send("Unsupported media type");
  }
});

app.delete("api/books/:bookId", (req, res) => {
  const id = parseInt(req.params.bookId);

  var success;
  if (id) {
    success = database.remove(id);
  } else {
    success = false;
  }

  if (success) {
    res.status(200);
    res.send("Deleted");
  } else {
```

```
    res.status(400);
  }
});

app.post("api/books", (req, res) => {
  const name = req.body.name;
  const released = parseInt(req.body.released);
  const author = req.body.author;
  const id = parseInt(req.body.id);
  if (name && released && author) {
    if (id) {
      database.update({
        key: id,
        book: {
          author: author,
          released: released,
          name: name
        }
      });
    } else {
      database.add({
        author: author,
        released: released,
        name: name
      });
    }
    res.status(201);
    res.send("success");
  } else {
    res.status(400);
    res.send("Bad format");
  }
});

app.listen(3000, function() {
  console.log("listening on 3000");
});
```

Appendix B

Interview answers for Q8

B.0.1 Person 1

```
let put: type a. route(a) = endpoint(
        ~handler=putInDatabase,
    ~success=Status.Created201,
    ~spec=
         Method.put
         >- Path.is("api")
         >- Path.is("books")
         >- contenttype([Spec.Accept.json(Decoders.jsonWithKey)])
         >- accept([Spec.Contenttype.plain(s => s)]), );
```

B.0.2 Person 2

```
let put: type a. route(a) =
  endpoint(
        ~handler=putInDatabase,
        ~success=Status.Created201,
        ~spec=
                Method.post
                >- Path.is("api")
                >- Path.is("books")
                >- accept([Spec.Contenttype.plain(s => s)]));
```

B.0.3 Person 3

```
let put: type a. route(a) =
    endpoint(
        ~handler=put,
        ~success=Status.Created201,
```

```
        ~spec=
        Method.put
        >- Path.is("api")
        >- Path.is("books")
        >- contenttype([Spec.Accept.json(Decoders.json)])
        >- accept([Spec.Contenttype.plain(s => s)]) );
```

B.0.4 Person 4

```
let put: type a. route(a) =
   endpoint(
       ~handler=putInDatabase,
       ~success=Status.Created201,
       ~spec=
   Method.put
       >- Path.is("api")
       >- Path.is("books")
       >- contenttype([Spec.Accept.json(Decoders.json)])
       >- accept([Spec.Contenttype.plain(s => s)]), );
```