Evaluating Functional Programming for Software Quality in 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 14, 2019

Abstract

Defects in Software engineering are a common occurrence. To mitigate defects the developers must create maintainable solutions and strive for good software quality. A maintainable solution is readable, extensible, 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 Anders Broberg for being my supervisor, my parents for their support and Michelle Neysa for listening to my non-stop ramblings about software quality and functional programming. 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 uses Object-oriented, imperative, languages. (ADD_REFERENCE usage statistics) As of today, defects in software are still commonplace 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 defects being so common engineers not only need new tools that decrease defects but also need to ensure that for future developers, the code is easy to modify so that when defects show up they can be fixed. The software needs to be maintainable to be of good quality.

Software quality can be divided into two different subparts: software functional quality and software structural quality. [3] Software functional quality reflects how well our system conforms to given functional requirements or specification and the degree of which correct software is produced. To check that the software is correct, software engineers create tests. To create tests, the engineer employs various patterns and tools in the code to make the code easier to test. These range from Test-driven development, Object-oriented programming, unit testing(ADD_REFERENCE TDD) to the use of static analysis and logical proofs.

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, are hard to measure quantitatively. By performing semi-structured interviews, it is possible to investigate how well the code is understood.

1.1 Objectives

This thesis aims to investigate how functional programming affects software quality when compared to imperative programming in server development. It will establish what constitutes good functional and structural quality in servers and then demonstrate how functional programming can be used to construct a library that forces good software functional quality.

Since software quality has two aspects, it will investigate afterward the impacts this functional solution has in software structural quality, which revolves around maintainability, testability, error-proneness, and readability. To do so it will construct two identical servers, one written in an imperative language and one in a functional language. These will then be compared using guidelines and interviews to find the impact in structural quality.

In servers, it is common to use a protocol called REST to establish communication between servers and clients.(ADD_REFERENCE REST) These servers are called RESTful APIs. In popular solutions, such as Express, developers are not forced to ensure that the server follows REST, which can potentially lead to errors and maintainability problems. This thesis is outlined as follows:

Chapter 2 explains RESTful APIs and establishes the challenges as well as current guidelines for ensuring good software structural quality in RESTful apis.

Chapter 3 explains how to quantitatively evaluate the software structural quality of REST apis by using interviews and analysing how well they follow guidelines.

Chapter 4 introduces a library for creating REST servers that adhere to the REST protocol by construction, ensuring increased software functional quality. The chapter also establishes the guidelines for evaluating software structural quality in Functional programming, using design patterns from Chapter 2 as a baseline.

Chapter 5 explains the results from using the created REST library for constructing server and comparing that to another imperative solution to compare the differences in software structural quality by performing the tests described in Method.

Chapter 6 analyses the impacts of software quality of the two solutions and concludes the pros and cons of functional programming as a solution for better software quality.

Chapter 7 Presents the future work and reflections about the thesis.

Chapter 2

Background

Software structural quality encompasses the maintainability aspects of the software, which includes aspects such as readability, error-proneness, extendability, and testability. The software usually evolves as new requirements come in, so even though it might be *functionally* good at a time, the developer will have to modify the code. Thus in the industry, it is not enough that software is only functionally correct, it also has to be structurally of good quality. There is also an importance in Q&A processes to be able to test that the software works correctly to check that the software also is functionally correct after modification. This chapter aims to introduce us to REST servers and what are construction concerns when making them, I.E. what is good software functional quality in REST servers. Then once we have established good functional quality, we introduce requirements in large scale software and the challenges that arise in the maintainability of the software so that we can establish guidelines of practices that allow for 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 protocol that is used to construct web services. A RESTful web service allows requesting systems to access and manipulate different representations of web services by using a set of stateless operations. The architectural constraints of REST are as follows:

Client - Server Architecture Separate the concerns between user interface concerns and data storage concerns. The server handles the management of resources and the client manages the display of those resources.

Statelessness Each request contains all the information necessary to perform a request. The 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.

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

Code on demand Servers can send functionality of a client via executable 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. The REST server has to, in its 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 optional query parameters. It also specifies 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`. Some common ones are 200, meaning successful; 404, meaning not found; 400, meaning badly formatted request.

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 its parameters.
- Parse the query and extract its parameters.
- Handle errors if the route or query is 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 Development concerns

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.(ADD_REFERENCE) 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 become 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 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.

2.3 Testing

As challenges arise in the development and as the software grows in scope, developers need to be able to verify that the software is correct. This helps to ensure that after modification the software works as expected and helps to catch defects.(ADD_REFERENCE test for bugs). In servers, three main types of tests are conducted: unit tests, integration tests, and E2E-tests.

2.3.1 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.(ADD_REFERENCE) 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.

2.3.2 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 occur when the modules interact with each other.

2.3.3 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.

2.3.4 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 find 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 nothing is 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.

2.4 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. [6] Rotten design also leads to problems that were established in Section 2.3.4, 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 make 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 SOLID guideline. SOLID is a mnemonic for five design principles to make the 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 source code.

Liskov substitution principle Given a base class and a derived class, 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.

Using a SOLID architecture helps to make 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 help to ensure that programs are extensible 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 3, these principles will be translated for Functional programming.

2.5 Functional programming for correct constructions

To mitigate the programmer from making mistakes, some languages feature a type system.(ADD_REFERENCE type systems) 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 higher kinded types)

It is possible to combine the type system with design patterns to force the developer to create the right thing. Chapter 4 will introduce a REST library, 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 4 makes heavy use of functional programming it might not be as understandable. Functional programming is not a popular software paradigm when compared to Object-oriented programming (ADD_REFERENCE statistics), thus readability might be affected. Understandability is important to reduce the learning time for programmers and cut down learning costs. Thus readability of software is an important criterion for good software structural quality.

2.6 Summary

This chapter introduced REST APIs and their requirements. We also established development concerns during the production of servers, which can be be summarised with these four points:

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 and lack of type system to enforce the right structure.

We went over the concerns of what happens when scaling software and that to ensure the quality then developers employ tests. We established that some of the quality can also be ensured by the type system, which aids in catching bugs and as

will be demonstrated in Chapter 4, enables us to ensure that servers are RESTful by construction.

To do unit tests, good architecture should also make it easier to create mocks. Thus we introduced SOLID principles, which works as a guideline for creating extensible software which can be modified over time and where dependencies are inverted making it easier to mock. However SOLID does not address readability of code. Even if the code is extensible it might not be understandable, meaning the developer will be incapable of extending it anyway. Thus when evaluating software structural quality it is important to both look at rotten design and readability.

Chapter 3

Method

Now that we have established what RESTful servers are and what the development challenges arise when creating them, the goal is to evaluate the potential of functional programming for better software quality. We established that two things need to be addressed for software structural quality: the avoidance of rotten design and good readability. By creating a semi-structured interview, where the subjects are asked open questions about how the code works then it can give insights about the readability of the source code. Adherence to SOLID principles can be used to avoid rotten design, thus we also need to evaluate the servers adherence to those principles. If the library can enforce that SOLID principles are followed through the type system, that the source code is readable and that the library forces the developers to follow the REST API then the servers produced by the library would have good software quality. In this chapter, we, therefore, describe how we can evaluate the software structural quality through interviews and using SOLID principles so that we can evaluate our solution's software quality. We then also describe how we can compare it to another solution written in imperative programming to see if it gives improvement over existing solutions.

3.0.1 Aims

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 unopinionated solution using Express, which is a good candidate to compare to a functional library which will be explained further in Chapter 4. Express is an unopinionated server framework written for Node.js for JavaScript. That a framework is unopinionated means that it does not force you to

architecture your code in any specific way.(ADD_REFERENCE UNOPINIATED) An idiomatic server was made using the library in Chapter 4 and the popular framework for Node Express. They feature similar functionality which is a REST 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 make use of a database that is abstracted away in the implementation. The supported content types will be `application/json` and for all endpoints and the displayable content-types are `text/plain` and `application/json`. They were written in an idiomatic way, that is they did not take the challenges outlined in Chapter 2 into consideration.

3.1 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 as guidelines for creating maintainable software. Those principles will be used as criteria that Cause should be evaluated 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.

3.1.1 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. (ADD_REFERENCE Code reviews)

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 diverge 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. (ADD_REFERENCE semi structured interviews) Semi-structured interviews are used to gather focused qualitative data. It is useful for finding insights about the readability of the code and if the code can be understood by others.

To conduct a 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 of the code is BookApi.re, re is the file extension of ReasonML, and BookApi.js, js is the file extension of javascript, respectively.

3.1.2 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.

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

Testability Evaluated 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.

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 4

Theory

With our research question now defined, the goal is to construct a library for REST APIs that is compliant by construction to ensure software functional quality. 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. The SOLID guidelines that Chapter 2 introduced were originally written for Object-oriented programming, so this chapter will also introduce SOLID principles but for functional programming.

4.1 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 the state from logic and immutable data.(ADD_REFERENCE)

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. [7]

Immutable data by default 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 function's 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 emphasize 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. (ADD_REFERENCE) 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 exists, this thesis defines it as making functions pure and inheritance being based around functionality rather

[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.

than attributes. More advanced constructs also exist for functional programming that needs to be introduced for constructing a maintainable rest library.

4.1.1 ADTs: Sum types and product types

A type is in Haskell is informally a *set* of possible values that a given data can have.(ADD_REFERENCE) 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 its 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 is similar to an immutable record in Javascript. (ADD_REFERENCE) *Maybe* 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.

4.1.2 Functors and Contravariant Functors

A Functor have 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 4.1.1. 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 $Predicate\ a = a \to Boolean$, 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 Functors have a function $contramap : (a \to b) \to m\ b \to m\ a$. (ADD_REFERENCE contravariacn) These are useful for defining how the value

should be *consumed*. For example, a *type encoder* = $a \rightarrow$ *encoded*, defines an encoder. The contravariant functor would allow transforming the encoder into intermediate value.

4.1.3 Domain-specific languages

A domain-specific language (DSL) is a programming language made for a specific domain.(ADD_REFERENCE DSL) 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.(ADD_REFERENCE edsl) 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.

4.1.4 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) and ensure that values of the DSL are statically correct.

```
1 type Calculator
2     Number : Int  -> Calculator Int
3     Bool : Bool -> Calculator Bool
4     Add : Calculator Int -> Calculator Int -> Calculator Int
5     Multiply : Calculator Int -> Calculator Int -> Calculator Int
6     Equal : Calculator a -> Calculator a -> Calculator Bool
```

Figure 4.1: A Calculator GADT with three operations add, eq and multiply.

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

Figure 4.2: A mathematical expression constructed using the GADT in figure 4.1

Figure 4.1 demonstrates a minimal example of 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 4.3 we demonstrate how we can evaluate the expression using pattern matching.

```
1 evaluate : forall a. Calculator a -> Int
2 evaluate (Add expr1 expr2) = evaluate expr1 + evaluate expr2
3 evaluate (Multiply expr1 expr2) = evaluate expr1 + evaluate expr2
4 evaluate (Equal expr1 expr2) = (evaluate expr1) == (evaluate expr2)
5 evaluate (Number i) = i
6 evaluate (Bool b) = b
```

Figure 4.3: 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 4.4

```
1 type Empty
2 type NonEmpty
3
4 type SafeList a b =
5      Nil : SafeList a Empty
6      Cons : a -> SafeList a b -> SafeList a NonEmpty
7
8 safeHead : SafeList a NonEmpty -> a
9 safeHead (Cons x _) = x
```

Figure 4.4: Type safe list

By separating how the expression from its 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. Since the logic is separated from the evaluation and

correct by construction it means that we only need to test the evaluator, which can be done using property-based testing (ADD_REFERENCE testing the hard stuff John hughes). Testing the logic means checking that its equal to its intended value, which requires manual review.

4.2 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 on the result. We first define a minimal example of a GADT *Router*, in Figure 4.5. 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 the constructor *Compose* allows composing routers together composed of multiple parts. For instance the URL `/hello/world` corresponds to *Compose* (*Exact* "*hello*") (*Exact* "*world*").

```
1 type Router
2     Top : Router
3     Exact : String -> Router
4     Compose : Router -> Router -> Router
```

Figure 4.5: Minimal router GADT

Definition in Figure 4.5 is not sufficient for a route parser 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. GADTs can also be used to also describe the transformation of the handler function's arguments, by extending Router with two parameters. In Figure 4.6 we extend the router with the constructors Integer, String and add two parameters input and output. The Integer and String constructors describe the application of an argument to a function. Together with *Compose*, 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 parameters get applied to the handler, thus the type of becomes `Router (String -> Int -> a) a`. Informally the type can be read as to give me a function which takes a String and an Int and I will give you 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 4.6: Router GADT extended with Int and string

So *Router* describes 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 4.7. `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 4.7: Router GADT extended with Int and string

This section has demonstrated how GADTs are useful for constructing an EDSL for routers. What was not described is how to interpret the router which is omitted for brevity. 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.

4.3 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 want to 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 composed of parts, where each part is one or more 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.
- Unlike the router defined in Section 4.2, query parameters also need to be supported.
- A set of content types that it can represent the resources.
- 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 $Result\ a = Ok\ a \mid Fail\ Message\ Code$.

An endpoint specification is defined as $Specification\ Request\ (Maybe\ Response)$, as it might fail to produce a response if it fails to parse the url. A difference from the *Router* in Section 4.2 is that parts need to produce *two* intermediate values. One value is the handler and another value 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 constructors, `Exact : String -> Specification input output` becomes `Exact : String -> Specification (handler, responseBuilder) (handler, responseBuilder)`. The constructor `Integer` gets modified to `Integer : Specification (int -> a, r) (a, r)`. The full GADT gets shown below.

We introduce a few new constructors, 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)`, with responsebuilder's definition expanded it produces `Specification (a, encoded) (a, r -> encoded)`. 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:

```
1   type Specification input output =
2     Exact : String -> Specification (h, r) (h,r)
3     QueryParam : string
4             -> (string -> Maybe a)
5             -> Specification (Maybe a -> b, r) (b, r)
6     Slash : Specification (a, b) (c, d)
7             -> Specification (c, d) (h, r)
8             -> Specification (a, b) (h, r)
9     IntegerParam : Specification (int -> a, r) (a, r)
10    Verb : HttpMethod -> Specification (h, r) (h, r)
11    Accept : [(MediaType, r -> encoded)] -> Specification (a,
         encoded) (a, r)
12    -- Attempt to extract body and apply it to the handler.
13    ContentType : [((MediaType, string -> Maybe body)]
14                  -> Specification (body -> b, r) (b, r)
15    Handler : StatusCode
16        -> handlerFunction
17        -> Specification (handlerFunction, noEncoder) (Result
             resource, resource)
18        -> Specification Request (Maybe Response)
19    Many :
20          [Specification
21            Request (Result Response)
22          ] -> 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.

The $Many$ constructor is for transforming multiple 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 return $Nothing$ then the specification interpreter should return a response with 404 - Not found.

From the $Specification$ GADT we can define a function
$make : Specification\ Request\ Response \rightarrow (Request \rightarrow Response) \sim Specification\ Request\ Response \rightarrow Server$ ($a \sim b$ means a is type equal to b). $make$ works by evaluating the GADT to deduce how the request should be parsed and how to produce a REST compliant response from that request. The full implementation of $make$, implemented in the functional programming language ReasonML, available in the appendix.

4.4 Using the library

The library exposes a set of functions that can be used to create specifications, which all make use of $specification$. 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 $(\triangleright)\ a\ b = Compose\ a\ b$.

4.4.1 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 its 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 its associated content media representation. This way it can check what media representation the request's content has and decode it and afterward 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]` and if successful returns a status code 200. Encoders are functions of type `book -> encoded`.

4.5 SOLID principles in Functional programming

To evaluate the library from a standpoint of testability, error-proneness and extendability using SOLID principles we first need to introduce how SOLID principles

work for functional programming. In the following sections, the five parts of SOLID principles have been translated to an equivalent for functional programming.

4.5.1 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. 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 function performs only a morphism, which is guaranteed if the function is pure.

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

4.5.2 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 means 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, the Liskov Substitution Principle is simply Contravariant Functors. 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 its result is in positive position hence its covariant. This principle does not apply to the application we will evaluate later.

4.5.3 Dependency Inversion Principle

Dependency Inversion Principle states that the logic should not depend on its 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 4.8, 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 4.8: A program that reads input from the computer and then prints it.

Instead, Figure 4.9 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 4.9: A program that reads input from the computer and then prints it, where the logic is separated from its 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. For a REST API library, it means that the logic should not depend on its 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 its evaluation. So the REST API is simply described as instructions of a GADT.

4.5.4 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.

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

Figure 4.10: Product type Color with cardinality too high

For example, $type\ Color = Blue|Green|Red$ has a cardinality of 3 (since it can either be Blue, Green or Red) whereas Fig. 4.10 has a cardinality of $2 \cdot 2 \cdot 2 = 8$ meaning that it has fives 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.

```
1 data IUserRepo =  {
2         getUser : Id -> IO User,
3         storeUser : User -> Id -> IO ()
4 }
5
6 -- Later on
7 getUserEndpoint : IUserRepo -> Request -> Response
8 -- ...
```

Figure 4.11: Normal interface for operations

Another example, observe that in Fig 4.11, the type `IUserRepo` has two operations. `getUserEndpoint` is a function meant to get a user from a database, thus it does not need to store anything. However as it takes $IUserRepo$ as an argument, the function is capable of producing more values than it should. This breaks the Interface segregation principle. So in summary, adherence to the interface segregation principle means that the cardinality of the types is minimized.

4.5.5 Open/Closed principles

Open/Closed principle states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. OCP is advice on how to write modules in such a way that we have backward compatibility and so that if extra functionality is needed, the modifier does not need to look at the class 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 to Functional programming, it can roughly be seen as the same as 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, new cases to datatype should be possible and new functions. OCP exists because modifying code in production might cause regressions. Thus a preferable solution is to extend the previous code instead.

4.6 Summary

This chapter introduced the concepts of functional programming and then from those concepts created a GADT that can be used to construct REST specifications. We have then demonstrated how a value of specification can be used to construct REST compliant server application, I.E. we constructed a higher-order function $make : Specification\ Request\ Response \rightarrow (Request \rightarrow Response)$. The interpreter can be found in Appendix B.1. We also established in this chapter guidelines for evaluating functional programs adherence to SOLID principles, which were originally introduced for Object-oriented programs. Using the library we created, we can construct a functionally correct server and evaluate its software structural quality by comparing it to an imperative version.

Chapter 5

Results

Four interviews were performed and two servers were implemented and analyzed. The source code for the programs can be found in Appendix A.1 for the ReasonML implementation and in Appendix A.2 for the imperative implementation, written in Express. This chapter will present the results of those interviews and also present an analysis of the server's adherence to SOLID principles.

5.1 Evaluating adherence to SOLID

We can through an expert analysis analyze the adherence to SOLID guidelines in Section 3.1 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 consider any design guidelines 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`. The module Endpoint's functions all revolve around the type `route (a)`, which all use book except the handler `delete`, which is a helper function for the function `router`. Thus the solution follows Single Responsibility Principle.

Open/Closed Principle

OCP, as defined in Section 4.5.5, 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 does not apply 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 its 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 its separation, it means that testing the logic of the API is easier.

5.1.1 Imperative solution

Since the solution was developed in an untyped language with no force of structure it makes sense that the SOLID principles will not be followed. However it is presented below:

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 subjects, which is a bit less than the recommended by Norman group of five subjects 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 have graduated students of the Engineering Interaction Technology and Design program at Umeå University. The program 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. We also show once more below the questions that were asked.

Q1 What is your experience with RESTful APIs?

Q2 What is your experience with Express?

Q3 What is your experience with ReasonML?

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

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?

	Person 1	**Person 2**
Q1	Implemented API that should follow REST. I assume its 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

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. Q5 for person 3 the question was omitted as the user could not guess what the code was supposed to do and assumed that it a system for encrypting books, rendering the question useless.

5.3 Summary

In this chapter we have compared the two servers, one created using imperative programming, one using functional and evaluated their's adherence to SOLID principles. We also conducted interviews on four subjects to establish the readability of the two solutions.

Chapter 6

Conclusion

The objective of this study is to evaluate how the functional solution impacts the software quality. Chapter 4 showed how functional programming can be used to enforce protocols and then using that created a library which enables developers to construct correct RESTful APIs. Since the imperative solution does not enforce that a server is necessarily RESTful, it follows that for a user who would typically not create incorrect RESTful APIs the functional solution can enforce a better software functional quality for RESTful APIs. However, as established at the start of Chapter 2 and 1, software quality also encompasses structural quality or maintainability. Section 2.6 introduced the four points of maintainability that needed to be evaluated against the library. These were testability, extendability, readability, and error-proneness. Evaluation is done by reviewing the results gathered in Chapter 5 and tie them together with the four points. These four points are what determine the software structural quality, which this chapter aims to do.

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 had 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 the code. 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 of how long it would take for users to understand it. The study indicates that there are costs in readability to the code for inexperienced users. Arguably these costs seem to be minor and that after a brief introduction the code would be understood. However, more research is needed to confirm that. It can also be argued that some of those costs can be mitigated by adding comments and making the changes in Section 7.1, but more research is needed to prove this.

6.2 Functional programming and SOLID

This study found that functional programming was capable of creating a library that could 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 the Interface Segregation Principle and Dependency Inversion Principle. This aids in reducing immobility, fragility, and viscosity. It was inconclusive as to if it enforces the Liskov substitution principle as further work is needed to create examples where this principle is properly tested. The functional solution also breaks the Open/Closed principle for situations where the user wants to add more details to a specification.

6.3 Summary

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

Testability Software became easier to test compared to an unopinionated solution 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 were made in extendability when using the functional solution, it had a negative impact due to breaking OCP.

Error-proneness The functional solution marginally affects the error-proneness as 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.

In conclusion, this thesis has demonstrated how functional programming can enforce good software functional quality but that it seems that in doing so, software structural quality was negatively impacted in terms of readability and extendability but positively impacted in testability and error-proneness.

Chapter 7

Reflections

From Chapter 6 we found that that the functional solutions had negative impacts in software structural quality. Section 7.1 describes how these results can be improved. This chapter concludes this thesis and give suggestions for future research and how it can be conducted.

7.1 Future work

SOLID principles emphasis extendability and that modifying original code is bad practice. It is questionable if these principles are relevant for strongly typed languages, as the types 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 inherent to the language itself. Some were fixed afterward, the version of the code in Appendix B.1 does not feature the problem with needing to write `type a. Route(a)`. It would be necessary to retry the experiment with five more people to find out if the changes made the code more comprehensible.

Creating the REST library went through a lot of iterations and a lot of work was put into making it more readable. GADTs were surprisingly difficult to grok and the applications were not clear but I hope with this thesis I demonstrated how they can be useful. I hope using GADTs as a way of constructing inputs to outputs proves useful for others.

7.1.1 Clarifying what is URL and what is not

Most users were uncertain what the URLs were with only person 1 correctly assuming that the URI for delete was $/api/books : id$. It might possible to further clarify what is a URI by wrapping it in a function that takes an incomplete route, making it clearer for the user what the URL and what is not. There is also a disconnect since for all other parts of the specification order does not matter while for the URI it does, this may confuse the user.

7.1.2 Extendability for the URIs

There are some issues still with the server not being as extensible 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.

7.1.3 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.

7.1.4 Evaluating effectiveness of SOLID

The effectiveness of SOLID has not been researched and it would be good to establish a correlation between software structural quality, defects and SOLID principles. However, due to the expensive nature of software development, this proves to be difficult as you have to create the same software using different methods for comparison over multiple years which can lead to biases.

7.1.5 Limitations

There is no way for the REST library to enforce that modification and verbs are linked. So if a specification specifies that it only works for GET requests, there is no

way to statically enforce that no mutation is done. There might be a way to statically enforce that handlers with GET requests must use coeffects and that instead, the handler describes what it requires from the database.(ADD_REFERENCE `http://tomasp.net/coeffects/`)

7.2 Improvements

While interviews worked for finding defects in readability, the results could have been improved by doing the interviews in batches. For those wishing to use this method, I recommend trying three or four iterations of interviews and attempt to fix the errors that come up at each set before doing the next set. By improving the library and server after the interviews, errors that are related to syntax and semantics which are not inherent to functional programming could have been eliminated.

It is unclear if SOLID principles are the best principles for maintainable software in functional programming. For instance, it might be worth asking if the expression problem is truly that big of a problem in practice for software developers.

I was surprised to find that the readability was impacted negatively and that despite this most subjects were still capable of extending the code properly without errors.

7.3 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. I hope also that it demonstrates that will it gives benefits in functional quality, structural quality might be affected which can potentially lead to other maintainability problems as the software evolves. My hope is in the future the software industry aims to write software as a specification and then an evaluator for that specification rather than first writing a specification on paper and then implement it as software and manually ensure that the software follows the specification.

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;
```

```
28      Json.Decode.{
29        title: json |> field("title", string),
30        year: json |> field("year", int),
31        author: json |> field("author", string),
32        id: json |> field("id", int),
33      };
34    };
35    let safeJson = myBook =>
36      try (Some(json(myBook))) {
37      | Json.Decode.DecodeError(s) => None
38      };
39    let int = s =>
40      try (Some(int_of_string(s))) {
41      | Failure("int_of_string") => None
42      };
43    let jsonWithKey = Json.Decode.tuple2(Json.Decode.int, json);
44  };
45
46  let deleteFromDatabase = key => {
47    let success = Database.delete(key);
48
49    if (success) {
50      Result.Ok("Deleted");
51    } else {
52      Result.Failed(
53        "Entry does not exist",
54        Status.BadRequest400,
55        MediaType.Plain,
56      );
57    };
58  };
59
60  let replace = (key, book) => {
61    let success = Database.replace(key, Encoders.json(book));
62    if (success) {
63      Result.Ok("Added");
64    } else {
65      Result.Failed("Database failure", Status.Error500, MediaType.
          Plain);
66    };
67  };
68
69  let insert = book => {
70    let success = Database.insert(Encoders.json(book));
71    if (success) {
72      Result.Ok("Added");
```

```
73    } else {
74      Result.Failed("Database failure", Status.Error500, MediaType.
          Plain);
75    };
76  };
77
78  let getFromDatabase = (queryAuthor, queryReleased) => {
79    let cmpAuthor = (author, book) => book.author == author;
80    let cmpReleased = (released, book) => book.year == released;
81    switch (queryAuthor, queryReleased) {
82    | (Some(author), Some(released)) =>
83      Result.Ok(
84        Database.filter(Decoders.safeJson, book =>
85          cmpAuthor(author, book) && cmpReleased(released, book)
86        ),
87      )
88    | (Some(author), None) =>
89      Result.Ok(Database.filter(Decoders.safeJson, cmpAuthor(author
          )))
90    | (None, Some(released)) =>
91      Result.Ok(Database.filter(Decoders.safeJson, cmpReleased(
          released)))
92    | (None, None) => Result.Ok(Database.get(Decoders.safeJson))
93    };
94  };
95
96  module Endpoint = {
97    open Spec.Router;
98
99    let replaceId: type a. route(a) =
100     endpoint(
101       ~handler=replace,
102       ~success=Status.Ok200,
103       ~spec=
104         Method.post
105         >- Path.is("api")
106         >- Path.is("books")
107         >- Path.takeInt
108         >- contenttype([Spec.Accept.json(Decoders.json)]),
109     );
110
111   let post: type a. route(a) =
112     endpoint(
113       ~handler=insert,
114       ~success=Status.Created201,
115       ~spec=
```

```
116           Method.post
117           >- Path.is("api")
118           >- Path.is("books")
119           >- contenttype([Spec.Accept.json(Decoders.json)]),
120       );
121
122     let get: type a. route(a) =
123       endpoint(
124         ~handler=getFromDatabase,
125         ~success=Status.Ok200,
126         ~spec=
127           Method.get
128           >- Path.is("api")
129           >- Path.is("books")
130           >- Path.query(~parameter="author", ~decoder=s => Some(s))
131           >- Path.query(~parameter="released", ~decoder=Decoders.
                  int)
132           >- accept([
133                Spec.Contenttype.json(Encoders.jsonList),
134                Spec.Contenttype.plain(Encoders.plainList),
135              ]),
136       );
137
138     let delete: type a. route(a) =
139       endpoint(
140         ~handler=deleteFromDatabase,
141         ~success=Status.Ok200,
142         ~spec=
143           Method.delete >- Path.is("api") >- Path.is("books") >-
                Path.takeInt,
144       );
145     let router = oneOf([get, post, replaceId, delete]);
146 };
147
148 Spec.listen(3000, Endpoint.router);
```

A.2 NodeJS REST implementation

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

```
7
8  const toString = book => book.author + " " + book.released + " "
       + book.name;
9
10 app.get("api/books", (req, res) => {
11   const released = parseInt(req.query.released);
12   const author = req.query.author;
13   var books;
14   if (released && author) {
15     books = database.getByReleaseAndAuthor(released, author);
16   } else if (released) {
17     books = database.getByRelease(released);
18   } else if (author) {
19     books = database.getByAuthor(released);
20   } else {
21     books = database.get();
22   }
23   switch (req.header) {
24     case "text/plain":
25       res.send(books.map(toString).join(", "));
26       break;
27
28     case "application/json":
29       res.send(books);
30       break;
31
32     default:
33       res.status(405);
34       res.send("Unsupported media type");
35   }
36 });
37
38 app.delete("api/books/:bookId", (req, res) => {
39   const id = parseInt(req.params.bookId);
40
41   var success;
42   if (id) {
43     success = database.remove(id);
44   } else {
45     success = false;
46   }
47
48   if (success) {
49     res.status(200);
50     res.send("Deleted");
51   } else {
```

```
52      res.status(400);
53    }
54  });
55
56  app.post("api/books", (req, res) => {
57    const name = req.body.name;
58    const released = parseInt(req.body.released);
59    const author = req.body.author;
60    const id = parseInt(req.body.id);
61    if (name && released && author) {
62      if (id) {
63        database.update({
64          key: id,
65          book: {
66            author: author,
67            released: released,
68            name: name
69          }
70        });
71      } else {
72        database.add({
73          author: author,
74          released: released,
75          name: name
76        });
77      }
78      res.status(201);
79      res.send("success");
80    } else {
81      res.status(400);
82      res.send("Bad format");
83    }
84  });
85
86  app.listen(3000, function() {
87    console.log("listening on 3000");
88  });
```

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,
```

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

B.0.4 Person 4

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

B.1 ReasonML implementation of Specification

```
1  open Result;
2  //
       ----------------------------------------------------------------------------

3  // SPEC
4  //
5  // Spec creates a pipeline for request to response. If any of the
       computations
6  // it will throw the correct error response.
7  type encoded = string;
8  type response = Response.content(encoded);
9  type responseBuilder('a) = Response.t('a);
10 type server = Request.t => response;
11 type t('input, 'output) =
12   | Top: t(('a, 'a), ('a, 'a))
13   | Exact(string): t(('a, 'b), ('a, 'b))
14   | Custom(string => option('a)): t(('a => 'b, 'c), ('b, 'c))
15   | Query(string, string => option('a))
16     : t((option('a) => 'b, 'c), ('b, 'c))
17   | Slash(t(('a, 'b), ('c, 'd)), t(('c, 'd), ('e, 'f)))
18     : t(('a, 'b), ('e, 'f))
19   | Integer: t((int => 'a, 'b), ('a, 'b))
20   | Path(t(('a, 'b), ('c, 'd))): t(('a, 'b), ('c, 'd))
```

```
21    | Method(HttpMethod.t): t(('a, 'b), ('a, 'b))
22    | Accept(list((MediaType.t, 'b => string))): t(('a, string), ('
        a, 'b))
23    | ContentType(list((MediaType.t, string => option('a))))
24      : t(('a => 'b, 'c), ('b, 'c))
25    | Map(Status.code, 'a, t(('a, string), (Result.t('c), 'c)))
26      : t(Request.t, Result.t(response))
27    | OneOf(list(t(Request.t, Result.t(response)))): t(Request.t,
        response);
28
29  type part('handlerArguments, 'unencodedResponse) = (
30    'handlerArguments,
31    responseBuilder('unencodedResponse),
32    Request.t,
33  );
34
35  let apply = ((arg, res, req): part('a => 'b, 'c), a): part('b, 'c
      ) => (
36    arg(a),
37    res,
38    req,
39  );
40
41  let setReq = (reqChanger, (arg, res, req): part('a, 'b)): part('a
      , 'b) => (
42    arg,
43    res,
44    reqChanger(req),
45  );
46
47  let id: type a. a => a = x => x;
48
49  /**
50  * A spec is a function that takes a request and returns an
      encoded response. The
51  * best way to create these with automatic error handling is by
      composing parts
52  * together and merge them using the success function.
53  */
54  module Accept = {
55    type decoder('a) = string => option('a);
56    /**
57    * Accept a request body of type application/json by providing a
        way to decode
58    * it.
59    */
```

```
60    let json = (decoder: Json.Decode.decoder('a)): (MediaType.t,
         decoder('a)) => (
61      MediaType.Json,
62      a =>
63        Json.parse(a)
64        ->Belt.Option.flatMap(value =>
65            try (Some(decoder(value))) {
66            | Json.Decode.DecodeError(s) => None
67            }
68          ),
69    );
70  };
71
72  module Contenttype = {
73    type encoder('a) = 'a => string;
74    /**
75    * Serialize the body to a text and set content type to text/
         plain.
76    */
77    let plain = (encoder: encoder('a)) => (MediaType.Plain, a =>
         encoder(a));
78
79    /**
80    * Serialize the response body to application/json and set the
         content type to
81    * applicaiton/json.
82    */
83    let json = (jsonEncoder: 'a => Js.Json.t): (MediaType.t,
         encoder('a)) => (
84      MediaType.Json,
85      a => a |> jsonEncoder |> Json.stringify,
86    );
87  };
88
89  /**
90  * Reads accept header of request and sets the encoder to the
       matching content
91  * type. If content type is not found response is set to
       Unsupported Media Type
92  * with status code 415.
93  */
94  let accept:
95    type a handler unencoded.
96      (list((MediaType.t, unencoded => string)), part(handler,
           string)) =>
97      Result.t(part(handler, unencoded)) =
```

```
98    (contentTypes, builder) => {
99      let makeList = x =>
100       Belt.Map.fromArray(
101         Array.of_list(x),
102         ~id=(module Request.MediaComparer),
103       );
104     let (h, res, req) = builder;
105     let makeEncoder = (req: Request.t) =>
106       Result.attempt(
107         ~message="Unsupported Media Type: " ++ MediaType.toString
              (req.accept),
108         ~code=Status.UnsupportedMediaType415,
109         ~contenttype=MediaType.Plain,
110         Belt.Map.get(makeList(contentTypes)),
111         req.accept,
112       );
113     //            --- (encoder(req))--------
114     //           /                          \
115     // (req) =>                             => (handler,encoded(
          response))
116     //           \                          /
117     //            --- response -------------
118
119     let maybeEncoder = makeEncoder(req);
120     Result.map(
121       encoder =>
122         (
123           h,
124           Response.contramap(encoder, res)
125           |> Response.setContentType(req.accept),
126           req,
127         ),
128       maybeEncoder,
129     );
130   };
131
132 /**
133   * Parses a query parameter from URL using a given parser and
        feeds it as an
134   * optional parameter into the handler.
135   * <pre><code>
136   * // Notice that it requires the parameter published!
137   * let books = (published: option(int)): list(book) => ...
138   *
139   * let api = Spec.query("published", Optional.int) |> Spec.
        handler(books)
```

```
140   * </code></pre>
141   */
142  let query =
143      (parameter, parser, builder: part(option('a) => 'handler, '
           response))
144      : part('handler, 'response) => {
145    let (h, res, req) = setReq(Request.parseQueries, builder);
146
147    (h(Request.query(parameter, parser, req)), res, req);
148  };
149
150  /**
151   * Takes a list of content types to decode the body of the
       request.
152   * The decoded body is then feeded as an argument into the
       handler.
153   */
154  let contentType:
155    type a b c.
156      (list((MediaType.t, string => option(a))), part(a => b, c))
           =>
157      Result.t(part(b, c)) =
158    (contentTypes, builder) => {
159      let makeMap = x =>
160        Belt.Map.fromArray(
161          Array.of_list(x),
162          ~id=(module Request.MediaComparer),
163        );
164      let acceptsMap = makeMap(contentTypes);
165      let decodeBody = (req: Request.t) =>
166        Result.attempt(
167          ~message=
168            "Could not parse body with content type: "
169            ++ MediaType.toString(req.contentType),
170          ~code=Status.BadRequest400,
171          ~contenttype=MediaType.Plain,
172          Request.decodeBody(acceptsMap),
173          req,
174        );
175      let (h, res, req) = builder;
176      decodeBody(req) |> Result.map(body => (h(body), res, req));
177    };
178
179  //------------------------------------------------------------------------------

180  // PARSING
```

```
let rec attempt:
  type a b c d. (t((a, b), (c, d)), part(a, b)) => Result.t(part(
     c, d)) =
  (route, state) => {
    let (handler, response, request) = state;
    switch (route) {
    | Top => Ok(state)

    | Exact(str) =>
      parseExact(str, request)
      |> Result.map(newReq => (handler, response, newReq))
    | Integer =>
      parseInt(request)
      |> Result.map(((i, newReq)) => (handler(i), response,
         newReq))
    | Custom(checker) =>
      parseCustom(checker, request)
      |> Result.map(((v, newReq)) => (handler(v), response,
         newReq))

    | Path(thePath) => attempt(thePath, state)

    | Query(param, decoder) =>
      query(param, decoder, state) |> (res => Result.Ok(res))

    | Slash(before, after) => attempt(before, state) >>= attempt(
       after)

    | Method(ofType) =>
      requestUsesMethod(ofType, request)
        ? Result.Ok(state) : Result.invalidParse
    | Accept(l) => accept(l, state)
    | ContentType(l) => contentType(l, state)
    };
  };

let rec build: type a. (t(Request.t, a), Request.t) => a =
  (route, request) =>
    switch (route) {
    | Map(code, subValue, subParser) =>
      attempt(subParser, (subValue, Response.default, request))
      |> Result.map(((value, response, _)) =>
           response
           |> Response.setCode(code)
           |> Response.fromResult
           |> Response.encode(value)
```

```
223           )
224       | OneOf(list) =>
225         switch (list) {
226         | [route, ...rest] =>
227           build(route, request)
228           |> (
229             fun
230             | Ok(response) => response
231             | Failed(_, _, _) => build(OneOf(rest), request)
232           )
233         | [] =>
234           Response.error(
235             ~message="Not found",
236             ~code=Status.NotFound404,
237             ~method=MediaType.Plain,
238           )
239         }
240       };
241
242   // Recall that responseBuilder(Request.t) = Request.t => response
      .
243   let (>-) = (a, b) => Slash(a, b);
244   let endpoint = (~handler, ~success, ~spec) => Map(success,
      handler, spec);
245   let oneOf = routes => List.map(r => r, routes) |> (r => OneOf(r))
      ;
246   let single = route => OneOf([route]);
247   let contenttype = l => ContentType(l);
248   let accept = l => Accept(l);
249
250   let uri = u => Path(u);
251
252   module Path = {
253     let top = Top;
254     let is = (str: string) => Exact(str);
255     let takeInt = Integer;
256     let takeText = Custom(value => Some(value));
257     let takeCustom = (f: string => option('a)) => Custom(f);
258     let query = (~parameter, ~decoder) => Query(parameter, decoder)
        ;
259   };
260
261   module Method = {
262     let get = Method(HttpMethod.GET);
263     let post = Method(HttpMethod.POST);
264     let delete = Method(HttpMethod.DELETE);
```

```
  let put = Method(HttpMethod.PUT);
  let update = Method(HttpMethod.UPDATE);
  let head = Method(HttpMethod.HEAD);
  let option = Method(HttpMethod.OPTION);
  let connect = Method(HttpMethod.CONNECT);
  let trace = Method(HttpMethod.TRACE);
  let patch = Method(HttpMethod.PATCH);
};

let make: t(Request.t, response) => server =
  (specification, request) => {
    let firstDropped =
      String.sub(request.url, 1, String.length(request.url) - 1);
    let requestPrim = {
      ...request,
      url: firstDropped,
      length: request.length - 1,
    };
    build(specification, requestPrim);
  };
```