HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

Bachelor Thesis

OpenAPI 3 Code Binding Generator for Haskell

And its Application to Generate a Library for the Stripe Payment System

Hochschule für Technik Rapperswil

Department of Computer Science

17.02.2020 – 12.06.2020

Authors Joel Fisch
Remo Dörig

Supervisor Prof. Dr. Farhad Mehta
External examiner Tom Sydney Kerckhove
Internal reviewer Mirko Stocker

Abstract

At the moment, there is no up-to-date client library for the online payment provider Stripe available in Haskell. Stripe provides an OpenAPI 3 specification for its API which can be used to generate a client library. Currently, there is no sufficient OpenAPI 3 client code generator for Haskell. The goal of this thesis is to create such a code generator implemented in Haskell and to use it to generate and publish a client library for Stripe. To demonstrate the usage of the generated code, a demo application was implemented using this library. In addition, automated tests were conducted on over 2700 existing OpenAPI 3 specifications. This approach allows easier updates in the future and lower maintenance effort, as it enables maintainers to regenerate the library when the OpenAPI specification is updated. In the future, the generator can be developed further to support more features of the OpenAPI specification and to create libraries for other API providers as well.

Lay Summary

Initial Situation Most applications (mobile apps, desktop applications, web applications) used today communicate with so-called web services (also known as Web APIs). To make it easier to implement an application which talks to a web service, providers of Web APIs write specifications which define in great detail what data should be sent to and can be received from the service.

OpenAPI 3 is a format to write such specifications. It allows to describe an API formally which makes it possible to generate parts of the code needed to communicate with the web service. This could be done manually as well, but it is labour-intensive to update a so-called client library every time the provider adds a new feature to the API. If a code generator is used instead, the only thing which has to be done is to use (the newly published version of) the API specification and transform it into code using the generator.

Haskell is a programming language for which no sufficient generator exists to fulfill this task. A generator would help the Haskell community to keep the client code for APIs such as the online payment provider Stripe up-to-date and to create libraries for other web services as well. Therefore, the goal of this thesis is to create a code generator which can transform an OpenAPI 3 specification into Haskell code.

Approach / Technology The code generator itself is implemented using Haskell as well. This code generator was used to generate a client library for Stripe which can be used to do online payments. To showcase the usage of this library, a small demo application was implemented using this library.

To ensure the correctness of the generated code, several types of automated tests were used. These tests check, for example, if the generated code can be transformed to an executable application.

Results All three parts (code generator, Stripe library and demo application) were successfully developed within this thesis. There are some limitations to the generator which could be addressed in the future to support more of the features provided by OpenAPI 3. To improve the development flow using the generated code, a documentation for the code can be generated which makes it easy to discover the possibilities of the API.

Conclusion The created code generator allows to update the generated Stripe library (and other libraries using the generator) easily and with low maintenance effort. Additionally, it can be used to create new libraries for Web API with an OpenAPI 3 specification. Different members of the Haskell community either plan to use the library to implement commercial software or plan to join forces to create an even better version of the code generator.

Management Summary

Initial Situation Most applications used today communicate with web services. These web services often use a formally defined format called OpenAPI 3 to specify the features of the service which makes the implementation of client applications easier and cheaper.

An OpenAPI 3 specification can be transformed into client code using a code generator to prevent repetitive development effort. For many languages there are such generators, but for the programming language Haskell there is no sufficient generator, resulting in high maintenance costs for client libraries and out-dated versions as it is the case for the community-developed library for the online payment provider Stripe.

The goal of this thesis is to create a code generator which can be used to transform an OpenAPI 3 specification into Haskell code and generate an up-to-date version of a Stripe client library.

Approach / Technology The code generator itself is implemented using Haskell as well. This code generator was used to generate a client library for Stripe which can be used to do online payments. To showcase the usage of this library, a small demo application was implemented using this library. To ensure the correctness of the generated code, several types of automated tests were used on around 2700 specifications.

Results All three parts (code generator, Stripe library and demo application) were successfully developed within this thesis. There are some limitations to the generator which could be addressed in the future to support more of the features provided by OpenAPI 3. To improve the development flow using the generated code, a documentation for the code can be generated which makes it easy to discover the possibilities of the API.

Conclusion The created code generator allows to update the generated Stripe library (and other libraries using the generator) easily and with low maintenance effort.

Acknowledgements

We thank the following people sincerely for their support:

- **Prof. Dr. Farhad Mehta** for his constant support and feedback as supervisor, his time answering our questions and interesting conversations, whether regarding the bachelor thesis or about something else.
- **Tom Sydney Kerckhove** for sharing his insights regarding Haskell and his constant interest in the project.
- **AnneMarie O'Neill** for correcting the English parts of the documentation.
- **Claudia Fisch** for correcting the German parts of the documentation.

Contents

Glossary

Apache Apache as referenced in this thesis is a web server. See https://httpd.apache.org/. 62, 113

ASCII The American Standard Code for Information Interchange is a character encoding using seven bits. 86

AST An abstract syntax tree (AST) is the representation of a program as data (tree structure) in contrast to a textual representation. 44, 45, 46, 47, 119

Cabal Cabal is a system for building and packaging Haskell libraries and programs. See https://www.haskell.org/cabal/. 110, 112

GHC The Glasgow Haskell Compiler is the main compiler for Haskell. 4, 35, 44, 101

Hackage Hackage is a package repository for Haskell, used by tools like Stack and Cabal. See http://hackage.haskell.org/. 63, 81, 109

Haddock A tool for automatically generating documentation from annotated Haskell source code. https://www.haskell.org/haddock/ 26, 44, 68, 78, 101

language extension A language extension of Haskell extends the language with some functionality. Language extension can be activated on a file level. See https://wiki.haskell.org/Language_extensions. xiv, 8, 35, 46, 73, 134

Monad Monads in Haskell can be thought of as composable computation descriptions. See https://wiki.haskell.org/Monad for more information. 11, 59, 118

Nginx Nginx is a web server. See https://www.nginx.com/. 62

property-based testing Tests that focus on the *properties* of functions. These *properties* are automatically tested with sophisticated random inputs. 5, 77, 79

SEPA The Single Euro Payments Area is a payment-integration initiative of the European Union for simplification of bank transfers denominated in euro. See https://www.sepa.ch/en/home.html. 62

Stack Stack is a cross-platform program for developing Haskell projects. It is aimed at Haskellers both new and experienced. Often used instead of cabal. See https://docs.haskellstack.org/en/stable/README/. 85, 106

Acronyms

ADT Algebraic data type. 44, 59

API Application programming interface. 2, 24, 92, 115

CD Continuous Deployment. 73, 75

CI Continuous Integration. 63, 73, 75

CLI Command Line Interface. 58, 65, 85, 88, 102, 119

ERP Enterprise resource planning. 2

HSR Hochschule für Technik Rapperswil. 113

HTTP HyperText Transfer Protocol. 2, 20, 30, 48, 49, 54, 55, 59, 61, 94, 103, 113, 121, 145

HTTPS HyperText Transfer Protocol Secure. 30, 49, 61

JSON JavaScript Object Notation. 2, 9, 14, 30, 33, 35, 52, 61, 62, 66, 79, 85, 86

MIT Massachusetts Institute of Technology. 89

SPA Single-page application. 62

UML Unified Modeling Language. 21

XML Extensible Markup Language. 14, 25, 30, 66

YAML YAML Ain't Markup Language. 64, 85, 86, 88, 106

List of Figures

List of Tables

Listings

Part I

Technical Report

Chapter 1

Introduction

1.1 Initial Situation

Most larger IT systems have a need to communicate with other systems. For example if a customer makes an order in an online shop, then the online shop may send the order to a ERP-system or if a button on a mouse is pressed, a signal to the computer is sent. These communications happen mostly over different interfaces, in case of (web-)applications these are called API. For systems working within the web it is common to have an API using HTTP and to send information in the form of JSON. How an API works, where it receives data, where it sends data, which form the data has, etc. can be formally specified. Many companies specify their APIs with OpenAPI, for example Stripe.

With an OpenAPI specification an API can be described in a standardized, machine readable way. As a result, it is possible to generate code in a desired programming language which can use the API. Code binding generators exist for a number of languages [1], but there is currently no suitable code generator for Haskell. See section 1.4 Research / Existing Work for the reasons why they are not suitable for this project (Code does not compile, no support for OpenAPI version 3.0).

The current Haskell library for Stripe [2] is hardcoded to an old version of the API [3]. This library is coded manually, which makes it labor intensive to update. A Haskell library which is automatically generated and supports the most recent Stripe API version is desired. And

[1] https://github.com/OAI/OpenAPI-Specification/blob/master/IMPLEMENTATIONS.md
[2] https://github.com/dmjio/stripe
[3] https://github.com/dmjio/stripe/blame/e00910f59b065bc68335c3f91d956ec0c662b0b4/README.md

therefore a Haskell code generator is desired. Stripe provides an OpenAPI 3.0 specification for its API at https://github.com/stripe/openapi.

The development of an OpenAPI code binding generator for Haskell could be used to generate an easy to maintain Haskell library for Stripe and could also have much broader use within the Haskell community.

The full original task description can be found in the appendix **??**.

1.1.1 OpenAPI 3.0

The OpenAPI Specification (originally known as the Swagger Specification) is a specification for machine-readable interface files for describing, producing, consuming, and visualizing web services. It is claimed to be a broadly adopted industry standard for describing modern APIs. The bachelor thesis works with the current OpenAPI version, which is 3.0.2, released on 08.10.2018.

1.1.2 Code Generators

Code generators can be used to generate code in a programming language from an input (typically a file / specification). Some code generators are used to create a basis of a project, which then can be manually extended. For this project the generated code is intended to be a self contained package, so that it can be updated when the specification is updated. An API defined by a OpenAPI specification has a client side and a server side. To work with the existing Stripe API, only a client side code generator is needed.

1.1.3 Haskell

Haskell is a general-purpose, statically typed, purely functional programming language with type inference and lazy evaluation. Developed to be suitable for teaching, research and industrial application, Haskell has pioneered a number of advanced programming language features, especially in the area of type systems. Haskell is used in a number of applications where efficiency and reliability are valued, particularly in the financial sector. The bachelor thesis works with the current version of the Haskell development environment GHC, which is 8.8.2, released on 16.01.2020.

1.1.4 Stripe

Stripe is a company that provides technical, fraud prevention, and banking infrastructure required to operate online payment systems. Stripe provides an API that web developers can use to integrate payment processing into their websites and mobile applications. Stripe provides an OpenAPI 3.0 specification for this API at https://github.com/stripe/openapi.

1.2 Goals

The original goal description can be found in the appendix **??**. The focus of this thesis is to create a working solution and not only a feasibility study.

1.2.1 OpenAPI Code Binding Generator

The primary goal of the project is to create an OpenAPI code binding generator for Haskell written in Haskell. With this tool, code bindings can be generated to interact with an OpenAPI specified API. The primary goal of the code binding generator is not to generate code for an API server, but to generate code to talk to an API server.

1.2.2 Haskell Library for Stripe API

With the code binding generator, a library for the Stripe API should be implemented. It should be possible to handle online payments with the library. The library should be easy to update if the Stripe OpenAPI specification is changed.

1.2.3 Demo Application

To prove that the library works, a simple working demo application that uses the produced Stripe API library for simple online payments should be implemented.

1.2.4 Requirements for the Implementation

Special care must be taken to take advantage of the advanced features of Haskell's type system to enforce data consistency at compile time, as well as the current state of the art in Haskell development (e.g. property-based testing). At the same time, the resulting tools must be understandable, usable and maintainable by a wide community of Haskell developers.

While developing the *OpenAPI code binding generator*, priority should be given to parts of the OpenAPI specification that are necessary to generate the code bindings required for the

Haskell Library for Stripe API. This is the minimum subset of theOpenAPI specification that must be supported [4].

All product documentation must be in English, and in a form that is appropriate for continued development, publicly or otherwise. Project documentation that is not relevant to the continued development of the project may be in German.

[4] Note that the Stripe OpenAPI specification also uses Stripe specific extensions to the OpenAPI specification.

1.3 Requirements

For this project three different sets of requirements exist for the code generator, the generated code and the Stripe library.

- Appendix A Requirements Specification for Code Generator
- Appendix B Requirements Specification for Generated Code
- Appendix C Requirements Specification for Stripe Library

1.4 Research / Existing Work

Previous work has been carried out in the field of OpenAPI, Stripe and Haskell. This chapter looks at existing work and examines if something useful can be learned.

1.4.1 OpenAPI Tools Code Generators

OpenAPI Tools [5] already have an OpenAPI version 3.0 compatible Haskell code generator [6,7] [16].

1.4.1.1 Stripe OpenAPI file

OpenAPI Tools Code Generators can successfully build Stripe API clients and servers if the OpenAPI validation is disabled, but for neither does the generated code compile. The server code fails to generate valid names for *anyOf* expressions and the client code does not use type names uniquely [8]. The server uses Servant [9]. Servant allows to define the API as a type and looks very promising for this project as well. The client uses *http-client* [10] and creates a custom function for every operation. Since this project is about building a client library, the generated client is of more interest.

The client represents simple data structures as records with the lens-library [13] and generates functions to create a record with every non *Maybe* value. Separate *anyOf* types are used for the expandable fields, but the types are never declared. The client uses many language extensions Listing I.4. QuickCheck [17] tests are generated to test the json serialization and deserialization.

1.4.1.2 Complex Example OpenAPI File

To explore how the generator generates complex types (*oneOf*, *anyOf*, *allOf*) and value constraints (*minLength*, *maxLength*), a custom OpenAPI file was created. The file is in the repos-

[5] https://github.com/OpenAPITools
[6] https://openapi-generator.tech/docs/generators/haskell-http-client
[7] https://openapi-generator.tech/docs/generators/haskell
[8] The type names also look wrongly generated but are valid.
[9] https://docs.servant.dev/en/stable/index.html

itory.

A *oneOf* type gets represented as an *Aeson* [10] value, which means it is not safely typed.

An *anyOf* type gets represented the same as a *oneOf* type, which means it is not safely typed.

An *allOf* type generates a new type with the fields of all the subtypes.

A *not* type gets represented the same as a *oneOf* type, which means it is not safely typed.

minLength*, *maxLength and other constraints are ignored.

date* and *date-time are represented as custom *newtype* for JSON serialization and deserialization which wrap *Data.Time* types.

nullable types are represented with the help of a *Maybe* type.

enum are represented as algebraic data types with custom serialization and deserialization functions.

1.4.2 servant-swagger

servant-swagger [11] can generate a Swagger specification (aka. OpenAPI version 2.0) from Servant API definitions. An analysis of this tool is not very useful for this project because this project generates code from a specification and not the other way around.

[10] https://hackage.haskell.org/package/aeson
[11] https://hackage.haskell.org/package/servant-swagger

1.4.3 Stripe API Coverage for Haskell by dmjio

A Stripe API library for Haskell exists[12], but it does not support current versions [13]. It does not use auto generated code.

The library uses *http-client* [10] and *http-streams* [14] for communication and *hspec* [15] for testing. For expandable types from an *expand* request, a custom expandable type family is used. [16] Other more complex types do not exist. JSON serialization and deserialization is written manually. Some helper functions are written which use primitive types for ease of use.

[12] https://github.com/dmjio/stripe
[13] https://github.com/dmjio/stripe/blame/e00910f59b065bc68335c3f91d956ec0c662b0b4/README.md
[14] https://hackage.haskell.org/package/http-streams
[15] https://hackage.haskell.org/package/hspec
[16] https://github.com/dmjio/stripe/blob/5dfc9b2a54f42a2e271f2d1535414016edd1c2a9/stripe-core/src/Web/Stripe/Types.hs

1.5 Involved People

Apart from the students conducting this bachelor thesis, there were four people involved: the supervisor Prof. Dr. Farhad Mehta, the external examiner Tom Sydney Kerckhove, the internal review Mirko Stocker and Markus Schirp.

Feedback from Prof. Dr. Farhad Mehta is documented in the meeting minutes and consists mostly of project related advice (regarding planning and the process). Tom Sydney Kerckhove gave inputs in the form of several code reviews and conversations via e-mail. It is denoted where his inputs directly led to changes. Everything not referenced anywhere else is listed below.

- Do not use *MonadThrow* and *MonadCatch*.
- Use *WriterT* instead of *StateT* for logs.
- Directly use a *Reader* monad instead of a *ReaderT* Monad transformer.
- Use *Text* instead of *String* in the generated code.
- Request to add an option to transform generated names to CamelCase.

Almost at the end of the project, Markus Schirp reached out to Tom Sydney Kerckhove and because he was developing a code generator for OpenAPI 3 as well. Several e-mail and Slack messages were exchanged. It is mentioned, whenever this material is used.

Chapter 2

Problem Analysis

This chapter describes the different problem domains present in this thesis and analyzes them.

2.1 Code Generation

To master the code generation domain, four different domains have to be discerned. The *Code-Generator* takes an *OpenAPI-Specification* and generates *Generated-Code*. The OpenAPI specification formulates its models in an *OpenAPI-Schema*.

The *OpenAPI-Specification* and *OpenAPI-Schema* are illustrations of existing domains for better understanding. The *Code-Generator* and *Generated-Code* are illustrations for solutions as part of this project.

In the following sections these domain models are shown and the important parts explained.

2.1.1 Code Generator Workflow

This diagram in Figure 2.1 Workflow Code Generation gives an overview of the *Code Generator* and how it will be used in a typical workflow.

From an *OpenAPI-Specification* including *Schemas* the *Generated Code* is generated. The *Schemas* get converted to *Models*. The *Generated Code* can be used by other Code (in the case of this project a Stripe library).

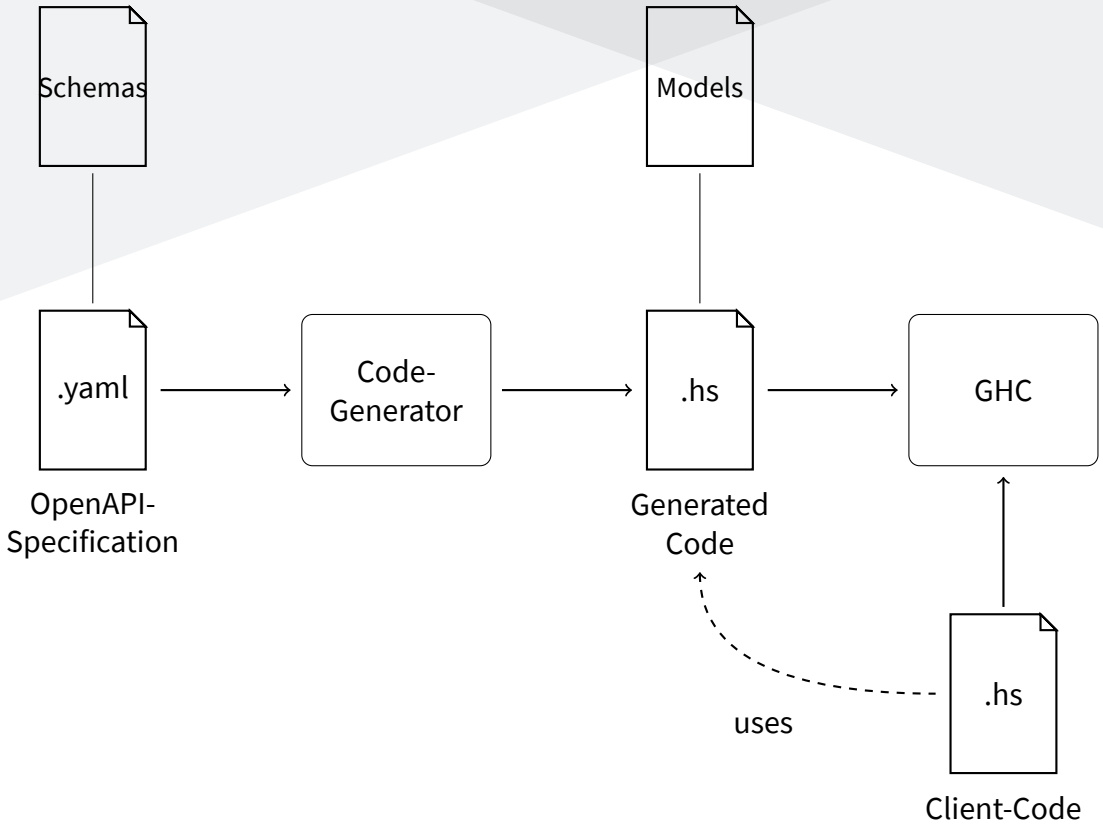

Figure 2.1: Workflow Code Generation

2.1.2 OpenAPI Schema

The information of this section is from the OpenAPI specification [14] and the JSON Schema [12].

The OpenAPI specification uses a modified version of the JSON Schema, which is used to validate and specify JSON and XML formats. For this project it is important how code can be generated, which can hold models conforming the specification. The Figure 2.2 Domain model of the OpenAPI schema draws the domain in an UML diagram.

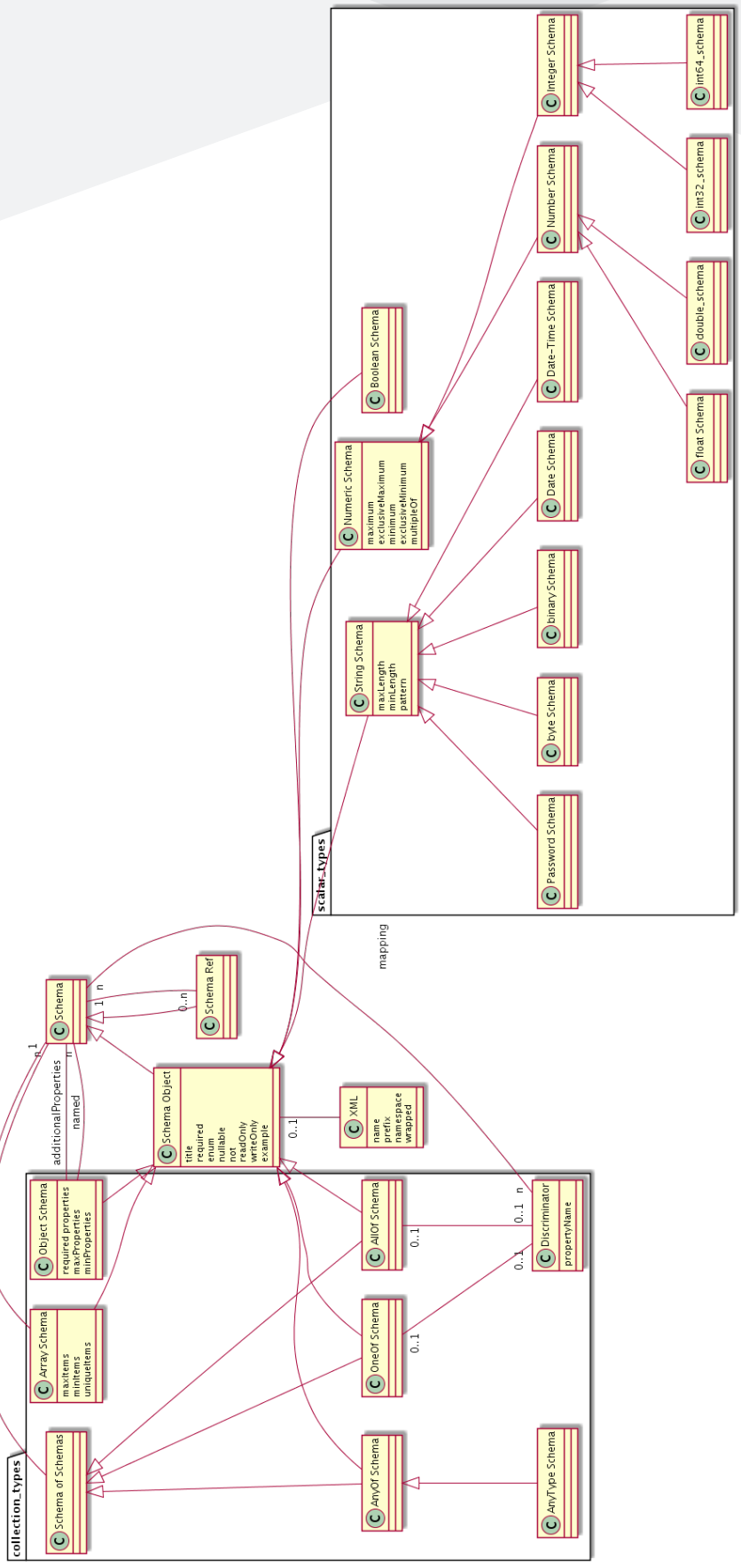

Figure 2.2: Domain model of the OpenAPI schema

2.1.2.1 Reference Types

A schema can either be a schema or a reference to another schema. These references can be circular.

2.1.2.2 Schema Object

All other types inherit *Schema Object* which defines some meta information.

writeOnly / readOnly With these properties a schema may define that it is only used when sending or retrieving data from a service. This essentially splits up the parent schema into two different models.

2.1.2.3 Scalar Types

There are a number of primitive types which should be easily expressible in their corresponding Haskell primitive types. Contrary to the standard JSON schema one can specify if an integer should be int32 oder int64, the same goes for the floating value types.

2.1.2.4 Array Schema

Defines a collection with items always corresponding to one single schema. If an array can have two different types, a *OneOf Schema* can be used.

2.1.2.5 Object Schema

This defines an object and is not to be mistaken with the *Schema Object*. An object may hold certain properties corresponding to a schema. Additional properties which cannot be named directly all correspond to the same schema.

2.1.2.6 *Inheritance / Polymorphism*

Further information:

https://swagger.io/docs/specification/data-models/inheritance-and-polymorphism/

AllOf Schema This schema can be used for inheritance / model composition. The schema incorporates all properties from the included schemas.

OneOf Schema This schema defines that a value must correspond to one schema of a collection of schemas, but it cannot correspond to more than one schema. The type *Any* can be represented as a *OneOf Schema*.

AnyOf Schema This schema is a mix of the *AllOf Schema* and the *OneOf Schema*. The schema must at least match one of the subschemas, but additional properties from other subschemas may be included.

Discriminator To differentiate better between subschemas, a discriminator property and a mapping from values to subschemas may be defined.

2.1.3 OpenAPI Specification

This model defines the OpenAPI specification [14] as is. This model will be transformed into the *Generated-Code* model. The *Schema* corresponds to the the model of subsection 2.1.2.

The Figure J.1 in the appendix refers to the complete specification. This includes information which is targeted for humans and not tools, such as examples and documentation and some meta information like server. This information may be useful to document generate code.

The Figure J.2 in the appendix refers to the complete specification without the reference types. Reference types can be either a concrete entity or a reference to this entity. For the purpose of modelling, it mostly distracts from more important information, but it should not be dismissed. References can be circular.

The Figure 2.3 refers to the specification in regards to the operations. This model is most useful to understand the problem domain for the generation of the code.

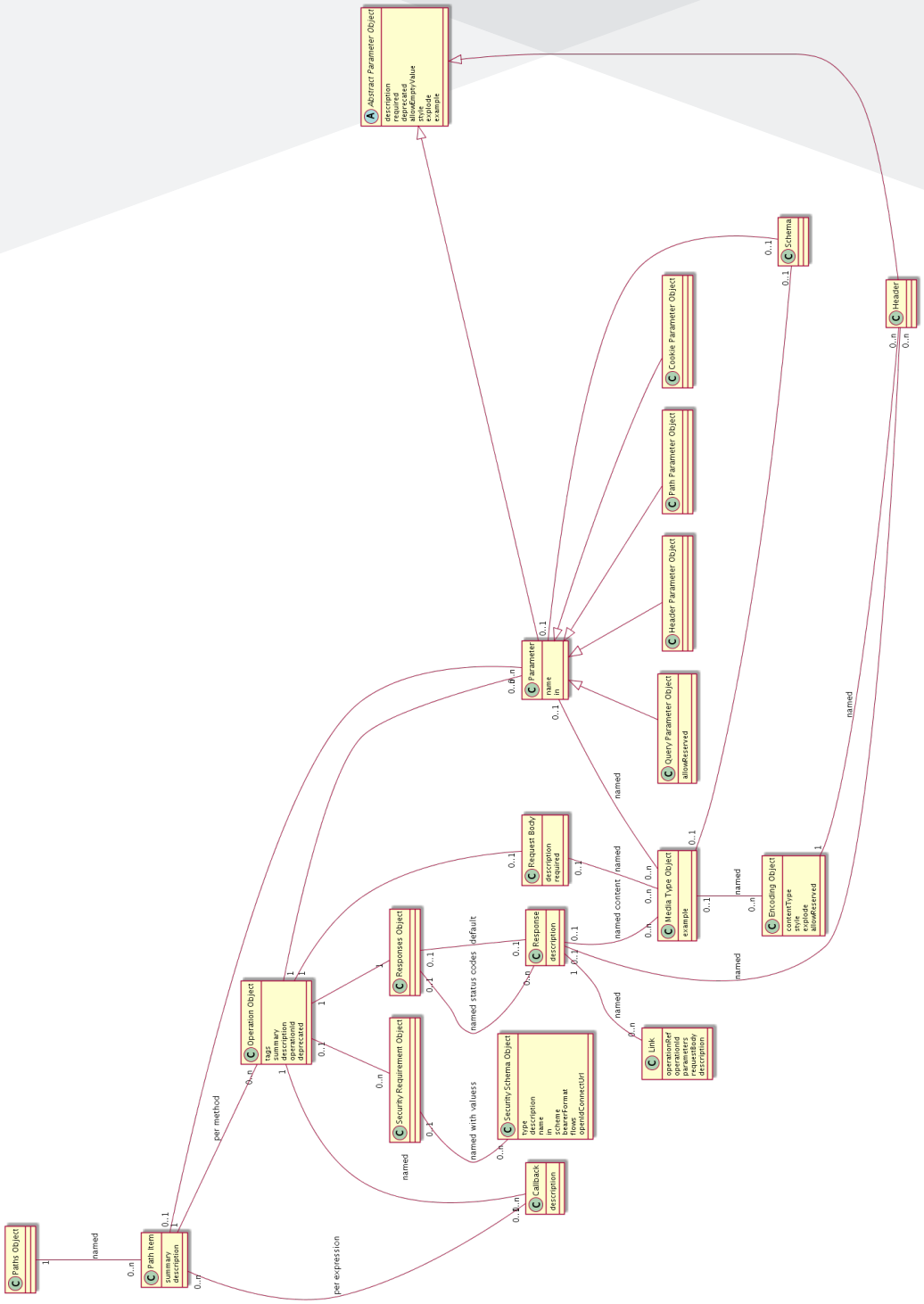

Figure 2.3: Domain model of the OpenAPI Specification in regard to the operations

2.1.3.1 Operation

An operation entity encapsulates an action (GET, POST, PUT, etc…) to a path. It has parameters in different places and a request body depending on the media type. It may produce different results depending on the HTTP status code and media type. Request body, response body, parameters and header parameters can correspond to a schema.

2.1.3.2 Parameters

Parameters to an operation can be in the query, the path, in a cookie or in a header. Parameters can refer to a schema. Parameters can have different styles of serialization and deserialization.

2.1.3.3 Link

Using links, one can describe how various values returned by one operation can be used as input for other operations. Runtime expressions variables from the request and response can be referenced.

2.1.3.4 Security

The OpenAPI specification allows different security authentication methods[1] specified for different operations.

Basic Authentication is the normal Basic Authentication with base64 encoding. Uses the standard *Authorization* header.

Bearer Authentication Uses the standard *Authorization* header. The format of the token can be described, but in the scope of the OpenAPI specification is only used for documentation.

[1] https://swagger.io/docs/specification/authentication/

API Keys API keys are parameters which will be sent on every request requiring the security schema. Similar normal parameters, they can appear in different locations.

OAuth and OpenID For OAuth and OpenID one can specify how to get the credentials. Some forms are entirely out of scope for this project since they require redirection of the user. Other forms could be implemented optionally, but it is possible to use the obtained credentials with the standard parameterization of the operations.

2.1.4 Generated Code

The model in Figure 2.4 should give an overview for how the generated code looks like. It is not a UML specification and should be looked at informally. This does not represent the model in which the generated code will be used. The model in Figure J.3 represents a more correct solution. It was disregarded in favor of a simpler solution.

2.1.4.1 Operation

An operation is an action to a certain predefined path. Certain types of parameters/request bodies and results are associated with an operation.

2.1.4.2 Domain Models

The domain models correspond to the schemas of the OpenAPI Specification. In the more correct solution Figure J.3 the models are split up between receiving and sending data because the same schema may have different properties depending on whether it is sent or received. Field research showed, that splitting up of models is rarely used. Those models have to be parsed and serialized.

2.1.4.3 Call Result

A call result is either a domain result or a custom error for example when the server is not reachable. A call result may have a result body corresponding to a model.

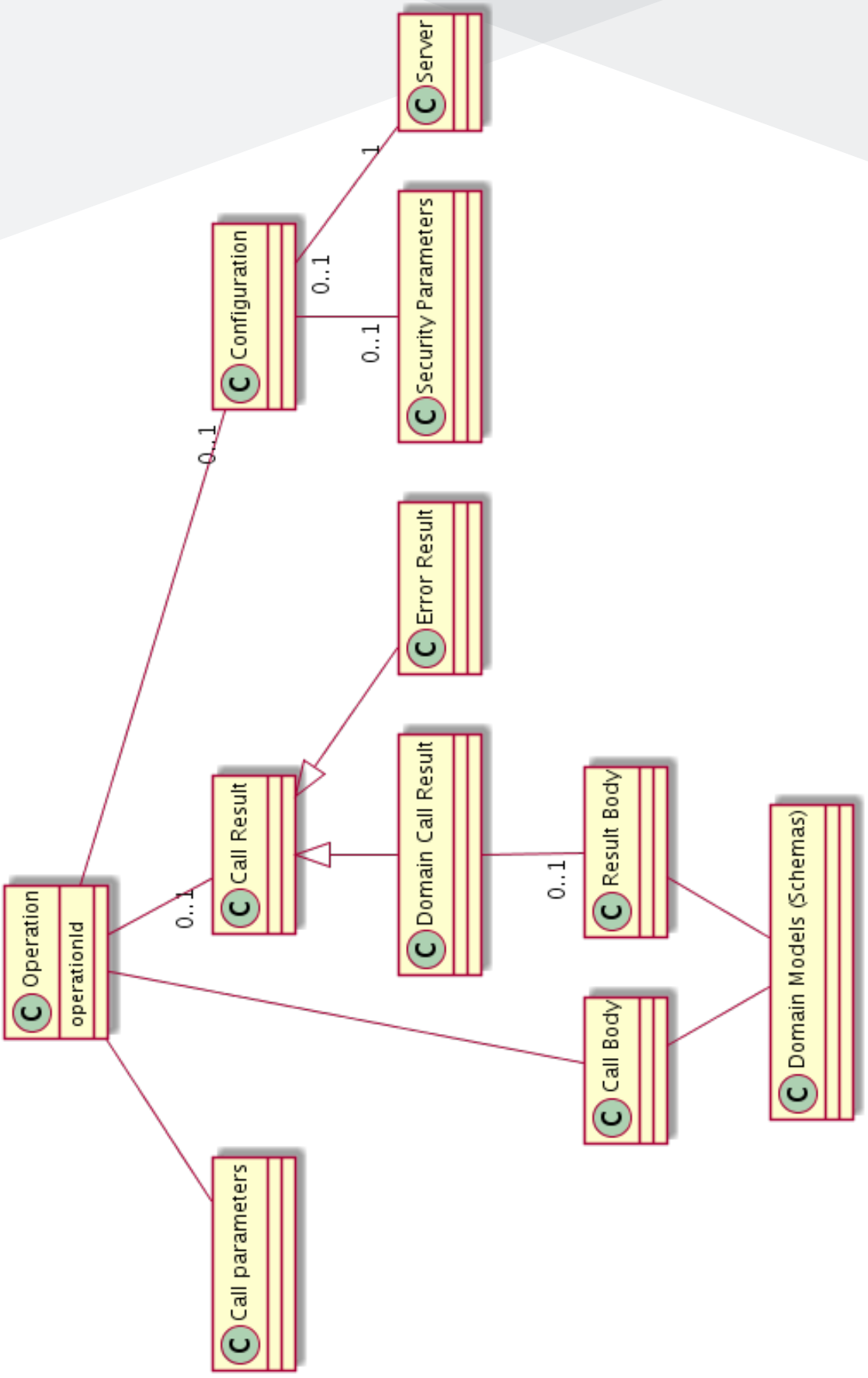

Figure 2.4: Informal domain model of the generated code

2.2 Stripe API

See Stripe for an introduction to what Stripe is. With around 58 thousand lines, the Stripe API OpenAPI specification is one of the largest found during research.

2.2.1 OpenAPI Features

This chapter describes some difficult OpenAPI features and if Stripe uses them [21] [20].

2.2.1.1 Security

Both basic authentication and bearer authentication can be used. They can be used for all operations. Only basic authentication is referenced in the documentation [2].

More complex security schemas like OAuth and OpenID can be ignored for the Stripe library.

2.2.1.2 Vendor Extensions

Stripe uses some vendor extensions [21].

***x-resourceId* and fixtures** This extension can be ignored as it is only used in conjunction with test fixtures which are not relevant for generating client code.

x-polymorphicResources This extension is described on the official GitHub page of the OpenAPI specification[3], but it is only used in the specs using version 2.0. Therefore, this extension can be ignored.

x-expandableFields* and *x-expansionResources Many objects that hold IDs of other objects can automatically be expanded. *x-expandableFields* defines which fields can be ex-

[2] https://stripe.com/docs/api/authentication
[3] https://github.com/stripe/openapi

panded. *x-expansionResources* defines to which resources an id can be expanded. See https://stripe.com/docs/api/expanding_objects for a detailed explanation.

It is possible to implement the API without the help of these two extensions since the data structures can be completely and correctly represented without them. They are useful to validate the *expand* property and to document its semantics. They describe which *expand* values expand which *id* properties to their corresponding resources. Without them it is not clear which resources will be expanded.

2.2.1.3 Links

Links are not used and can be ignored.

2.2.1.4 Callbacks

Callbacks are not used and can be ignored.

2.2.1.5 Expressions

Expressions are not used and can be ignored.

2.2.1.6 Polymorphism

allOf is not used and can be ignored.

oneOf is used in conjunction with *x-expansionResources*, otherwise it is not used.

anyOf is often used in conjunction with *x-expansionResources* to either describe a resource or an id of a resource but not solely. It is also used to give additional descriptions to referenced resources.

2.2.1.7 XML

XML is not used and can be ignored.

2.2.1.8 Schema Restrictions

The OpenAPI specification allows to define some restrictions on schemas which are not or not practical to represent as types at compile time. It is necessary to define how those restrictions are handled since the Haskell type system cannot support the programmer in those cases without heavy expenses at usability.

The enforcement of the constraints has to be done on the server because clients can violate them anyway. Therefore, it is generally not required to enforce them on the client. The reason to process them anyway would be to deliver faster feedback to the user of the client library. Since this does not help directly achieve the goals of this project, the following restrictions from Schema Object [4] are ignored within this project with the perspective that they could be supported in the future. They are not used within the Stripe API definition.

- multipleOf
- maximum
- exclusiveMaximum
- minimum
- exclusiveMinimum
- minLength
- maxItems
- minItems
- uniqueItems
- maxProperties

[4] http://spec.openapis.org/oas/v3.0.3#schema-object

- minProperties
- not
- readOnly
- writeOnly
- xml

The restrictions used in the Stripe API and not needed for schema generation are:

- maxLength
- pattern

The additional value generated by validating those restrictions on the client side (especially in responses) can be neglected and therefore will only be taken into account if time allows. At least the two restrictions used by the Stripe API should be reflected within the Haddock documentation.

2.2.2 Demo Use Case

For the purpose of demonstration, a demo online payment must be handled.

There are two easy ways to pay online via Stripe, either using the *Payments*[5] or using the *Checkout*[6] APIs[7]. With *Payments* an iframe is displayed and the customer does not leave the website. With *Checkout* the customer is redirected to the website of Stripe and redirected back again. *Payments* use *PaymentIntents*[8], *Checkout* uses *Checkout-Sessions*[9]. With this the customer completes its purchase via Stripe and the website is notified if it was successful. To securely verify that the payment is complete the status of a *PaymentIntent* can be retrieved. Webhooks[10] can be used in both cases to be independent of browser behavior.

[5] https://stripe.com/docs/payments
[6] https://stripe.com/docs/payments/checkout
[7] Other ways exist, but are out of scope for the demo use case
[8] https://stripe.com/docs/api/payment_intents
[9] https://stripe.com/docs/api/checkout/sessions/create
[10] https://stripe.com/docs/webhooks

The Figure 2.5 shows from where to where calls are made and how an online payment proceeds with Stripe.

To verify broader usage *Payments* and *Checkout* will be used. For the sake of simplicity, only *Checkout* API will be used with a server side verification and both without webhooks.

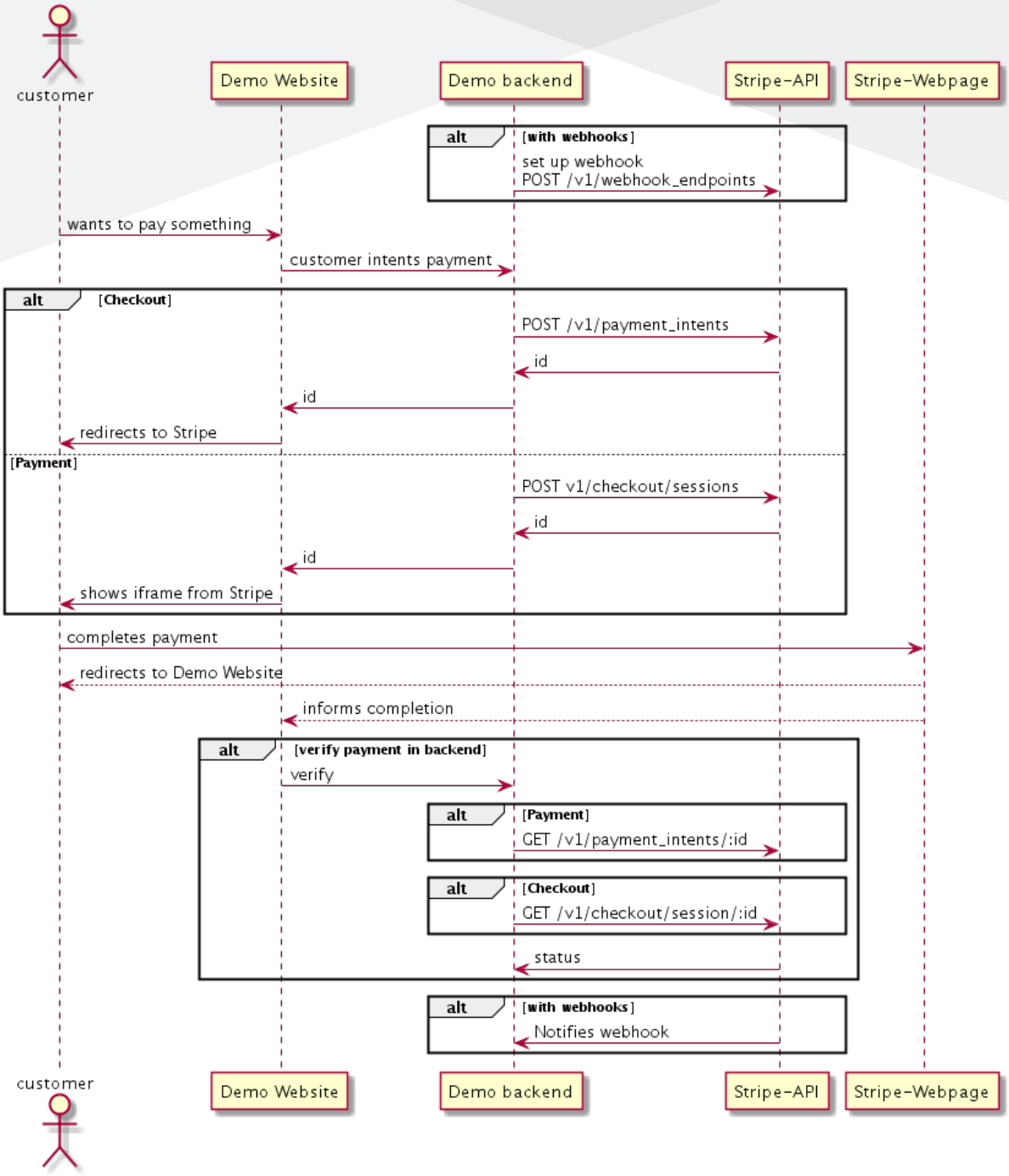

Figure 2.5: Sequence diagram of an online purchase with Stripe

Chapter 3

Solution Design

In this chapter, different areas needed for a working solution satisfying the requirements are examined and solutions are worked out. Where needed, multiple options are weight up against each other, and a decision is documented.

3.1 OpenAPI 3

This chapter evaluates the required Haskell features for the OpenAPI specification. This analysis stems mainly from the analysis in subsection 1.1.1 OpenAPI 3.0 as also from section 2.1 Code Generation, especially chapters 2.1.2 and 2.1.3. The feature set of a generator from *openapi-generato.tech* [15] was also used.

3.1.1 HTTP Calls

Since OpenAPI specification is a specification on top of HTTP, the code generators need to be able to execute and understand HTTP calls. Because the generator will be used for client applications only, outgoing calls need to be supported. There are numerous HTTP libraries for Haskell, see section 3.3 HTTP Library for more information.

1. All HTTP methods and custom methods need to be supported.
2. HTTPS needs to be supported.

3.1.2 Data Formats

As default data is sent as JSON but can also be sent as XML. Extensions for OpenAPI specification can also allow other formats, for example *Protocol Buffers*[1]. Stripe also uses *application/x-www-form-urlencoded* and *multipart/form-data* for sending data, however *multipart/form-data* is used seldom.

3.1.3 Data Models

OpenAPI data models are defined by schemas. See 2.1.2 for their domain model. These schemas should be represented in an easy to use type safe way.

For the following sections there are code examples after the text when applicable. Because of their size, they are mostly on the next page respectively. The code examples are not gen-

[1] Protocol Buffers and specification language developed by Google: https://developers.google.com/protocol-buffers

erated, they put emphasis on the logic of implementation and understandability.

3.1.3.1 References

References will be represented as a reference to a type, only concrete schemas will create new types. This means that the name of a type must be discoverable from the reference. For *anyOf*, *oneOf*, *allOf* references will have to be resolved. The integrity of references will not be checked.

3.1.3.2 Naming and Ordering

For the created Haskell types meaningful names are needed which are valid Haskell type-names, type-constructors or record-field-names. An OpenAPI type can have a *title* attribute which is not required to be unique. Another variant is to use the path/component in which the type was declared. Different existing code generator [15] provide the user with the possibility of configuring naming and ordering information.

3.1.3.3 Scalar Types

Mapping the scalar OpenAPI types to Haskell types is relatively straightforward, it must be considered that per default numbers do not have a fixed size and that strings are unicode strings. JSON encoding and decoding should also be straightforward. Examples for scalar types can be found in the appendix I.1.

date* and *date-time are strings but they represent ”time” types. The time [2] library seems to be the canonical library for time management [23] and will therefore be used.

[2] http://hackage.haskell.org/package/time

3.1.3.4 Arrays

OpenAPI arrays can be mapped to Haskell lists and because they are polymorphic there is no need to differentiate between lists of different items.

```
# ["hello","world"]
type: array
items:
  type: string
```

Listing 3.1: OpenAPI example array

```
{-# LANGUAGE OverloadedStrings #-}

module ExampleArray where

import qualified Data.Text as T

example :: [T.Text]
example = ["hello", "world"]
```

Listing 3.2: OpenAPI Haskell example array

3.1.3.5 Objects

OpenAPI Objects can be mapped to Haskell records.

Duplicate record-fields are not possible in the same module, since a field generates a function in the namespace of the module. This is crucial for auto generated code. Each field has to be prefixed with the record name to avoid name collision.

Code for JSON encoders and decoders has to be generated. Encoders and decoders for other formats can rely on JSON encoders and decoders. They could be autogenerated with either *Template Haskell* or *deriving Generics*, but the penalty in compilation time is too high.

For this project using simple records is the least complex and most manageable variant.

```yaml
type: object
properties:
  id:
    type: integer
  trustworthy:
    type: boolean
    nullable: true
```

Listing 3.3: OpenAPI example object

```haskell
{-# LANGUAGE OverloadedStrings #-}

module ExampleObject where

import Data.Aeson

data Example
  = Example
      { exampleId :: Maybe Integer,
        exampleTrustworthy :: Bool
      }
  deriving (Show, Eq)

instance ToJSON Example where
  toJSON obj =
    object
      [ "id" .= exampleId obj,
        "trustworthy" .= exampleTrustworthy obj
      ]

instance FromJSON Example where
  parseJSON =
    withObject
      "Example"
      (\obj -> (Example <$> obj .:? "id") <*> obj .: "trustworthy")
```

Listing 3.4: OpenAPI Haskell example object

Disregarded Plans Throughout development the object implementation plan has undergone drastic changes.

Since GHC version 8.0, there is a language extension *DuplicateRecordFields* [6] to solve the problem of duplicate record fields. Prefixing all fields has the disadvantage that it is verbose. But even with *DuplicateRecordFields*, the compiler often complains about ambiguous usage of the record access functions and the types have to be made explicit, either with a separate helper function or by annotating the object with its type every time it is used. During development *DuplicateRecordFields* has been used for the models of the generated code. The overhead for generating helper functions compared to the verbosity of prefixed fields turned out to be too large.

Deriving JSON encoders and decoders with prefixed data structures is possible. The function *fieldLabelModifier*[3] can for example be used with the *aeson* package. During development the compilation effort was so high, that the computers [4] used for development could not finish the compilation. To decrease compilation time, code for JSON encoders and decoders has to be generated.

Another problem is that nested updates use a verbose syntax. The Haskell community has already created libraries to work around this, most notably *lens* [13]. The *lens* library can automatically generate lenses for both prefixed (makeFields) and not prefixed [5] (makeFieldsNoPrefix) version [6]. Because of feedback from Tom Sydney Kerckhove [3] and the compilation effort necessary for *lens*, *lens* got disregarded.

For this project using the *lens* library in conjunction with the Haskell language extension *DuplicateRecordFields* seemed the least verbose and complex variant in the beginning but prefixed field names were much more practical in the end due to the reasons mentioned above.

[3] https://hackage.haskell.org/package/aeson-1.4.6.0/docs/Data-Aeson.html
[4] The computers have at least 24gb of RAM
[5] The ”not” prefixed variant still needs an underscore as a prefix
[6] https://hackage.haskell.org/package/lens-4.16/docs/Control-Lens-TH.html

```haskell
{-# LANGUAGE DuplicateRecordFields #-}

module ExampleObject where

data ExampleObject
  = ExampleObject
      { id :: Integer,
        trustworthy :: Maybe Bool
      }
  deriving (Show, Eq, Ord)

exampleObj :: ExampleObject
exampleObj = ExampleObject 7 (Just True)
```

Listing 3.5: OpenAPI Haskell example object (disregarded plan)

3.1.3.6 oneOf

The *oneOf* OpenAPI type can be mapped to algebraic Haskell data types. OpenAPI subtypes are always uniquely identifiable, it is not possible that a data structure can be of more than one subtype. If discriminators and a mapping is present, they can be used to name the type constructors.

```
oneOf:
  - type: object
    properties:
      bark:
        type: boolean
      age:
        type: integer
  - type: object
    properties:
      length:
        type: integer
      height:
        type: integer
```

Listing 3.6: OpenAPI example oneOf

```haskell
{-# LANGUAGE DuplicateRecordFields #-}

module ExampleObject where

data ExampleObjectOne
  = ExampleObjectOne
      { bark :: Bool,
        age :: Integer
      }
  deriving (Show, Eq, Ord)

data ExampleObjectTwo
  = ExampleObjectTwo
      { length :: Integer,
        height :: Integer
      }
  deriving (Show, Eq, Ord)

data ExampleOneOf = One ExampleObjectOne | Two ExampleObjectTwo
  deriving (Show, Eq, Ord)

exampleObj :: ExampleOneOf
exampleObj = One (ExampleObjectOne True 7)
```

Listing 3.7: OpenAPI Haskell example oneOf

3.1.3.7 allOf

The *allOf* OpenAPI type can represent inheritance, a feature Haskell does not have. Every new *allOf* type will be a new Haskell type without a connection to the other Haskell types. It is important that the merging of the OpenAPI subtypes to one OpenAPI type happens at the level of the OpenAPI schema because it is also possible to merge two scalar schemas. After the merging, no information about the *allOf* type exists and can be treated as a normal type.

```
type: object
  properties:
    bark:
      type: boolean
    age:
      type: integer
allOf:
  - $ref: '/components/....'
  - type: object
    properties:
      length:
        type: integer
      height:
        type: integer
```

Listing 3.8: OpenAPI example allOf

```haskell
{-# LANGUAGE DuplicateRecordFields #-}

module ExampleObject where

data ExampleObjectOne
  = ExampleObjectOne
      { bark :: Bool,
        age :: Integer
      }
  deriving (Show, Eq, Ord)

data ExampleAllOf
  = ExampleAllOf
      { bark :: Bool,
        age :: Integer,
        length :: Integer,
        height :: Integer
      }
  deriving (Show, Eq, Ord)

exampleObj :: ExampleAllOf
exampleObj = ExampleAllOf True 7 6 5
```

Listing 3.9: OpenAPI Haskell example allOf

3.1.3.8 anyOf

The *anyOf* OpenAPI type is a mixture of *oneOf* and *allOf*. Its OpenAPI subtypes are not necessarily all of the same OpenAPI type, for example Stripe uses *anyOf* to represent either an id to a resource or the resource itself. To represent an *anyOf*, it needs to be converted to a *oneOf* type with all subtypes being *allOf* types with all other fields nullable. Only subtypes of the same primitive type (objects with objects, ints with ints) can be merged.

```
type: object
  properties:
    bark:
      type: boolean
    age:
      type: integer
anyOf:
  - type: integer
  - $ref: '/components/....'
  - type: object
    properties:
      length:
        type: integer
      height:
        type: integer
```

Listing 3.10: OpenAPI example anyOf

```haskell
{-# LANGUAGE DuplicateRecordFields #-}

module ExampleObject where

data ExampleObjectOne
  = ExampleObjectOne
      { bark :: Bool,
        age :: Integer,
        length :: Maybe Integer,
        height :: Maybe Integer
      }
  deriving (Show, Eq, Ord)

data ExampleObjectTwo
  = ExampleObjectTwo
      { length :: Integer,
        height :: Integer,
        bark :: Maybe Bool,
        age :: Maybe Integer
      }
  deriving (Show, Eq, Ord)

data ExampleAnyOf
  = I Integer
  | One ExampleObjectOne
  | Two ExampleObjectTwo

exampleObj :: ExampleAnyOf
exampleObj = One (ExampleObjectOne True 7 (Just 6) Nothing)
```

Listing 3.11: OpenAPI Haskell example anyOf

3.1.3.9 anyType

The special OpenAPI type *anyType* can represent any value and is defined as a special case of the *anyOf* type [7]. This would be possible but a bit unpractical so it is easier to work with JSON data types directly.

[7] https://swagger.io/docs/specification/data-models/data-types/#any

3.2 Code Generation

One of the main goals of this project is to generate Haskell code. Therefore, it is important to investigate the different options of code generation in this language and its corresponding eco system. This section describes the evaluated features and libraries and explains the reason for the final choice.

3.2.1 Criteria

The code generation library / method should be evaluated using the following criteria:

- Comments (Haddock): It must be possible to generate Haddock comments as this is the main way of documenting the generated code and is necessary for the use of the generated code.
- Correct code generation (parsable and type-safe): The generated code should be compilable by GHC (version 8.8). For this purpose there are several levels of support a library can give, reaching from generating parsable code up to type-safe code with the intended types.
- Future support: The library should be actively maintained to increase the probability of long-term support for a continued development of the code generator.
- Options for code generation: Typically an AST is represented in Haskell using algebraic data types. These can be constructed with the typical data constructor invocation (as one would do with any ADT). Some libraries offer helper functions to allow easier construction of more complex ASTs. But a Haskell AST can also be represented using actual Haskell code. This criteria evaluates if it is possible to construct the whole or parts of an AST using Haskell code literals (regardless if these are inlined or loaded from external source files).
- Usability: This includes how easy it is to use and especially to get started.

The speed of the code generation is a criteria which is specified in the requirements specification. Since the goal in this area is not very difficult to reach and not crucial for the useful-

ness of the generator, it is not covered here as it would be very time-consuming to check the speed of all of the libraries.

3.2.2 Candidates

The following options have been investigated in order to decide which one should be used for this project.

Plain text manipulation This option is the simplest method and allows the generation of arbitrary strings. Therefore, the generation of Haddock comments is easy but there is no extra support to it. There is absolutely no support in the generation of correct code with this method but this option will be supported as long as Haskell exists. Speed should be of no concern as it is simple *Text* manipulation.

Haskell Source Extensions *Haskell Source Extensions* [7] is a standalone parser for Haskell which can be used to parse and manipulate Haskell code as well as pretty-print it. The main focus does not lie on code generation but nevertheless, it could be used in such a way. It is currently only maintained and not developed any further.

The generation of comments could be be done using *haskell-src-exts-sc* [9]. It makes comment generation easier with convenience functions such as *preComment*. Unfortunately, it was not possible to build a compiling application using this module and neither did the provided example compile. Therefore, it is questionable how well maintained this package is.

With this method it is possible to guarantee the generation of parsable code but it is not ensured that it will type check. The extension *haskell-generate* [8] claims to be able to ensure this, but this library is not well maintained (the last commit was in 2016).

Since it is a parser at the same time, existing source code could be parsed to an AST and used for code generation. A wild mix of both creating an AST and parsing is likely to be inconvenient but possible.

ghc-source-gen *ghc-source-gen* [5] is a library which is designed to generate whole Haskell files or code fragments. Under the hood, it uses the AST of the GHC library API to represent the code which should be generated.

It does not seem to support the generation of comments. Since it is only possible to generate a string of the AST, the comment generation would have to be performed the same way as plain text manipulation.

The generated code should be parsable but will not necessarily type check. With this method, it is not possible to create code from existing code.

Regarding maintenance it can be said that the example is working and the code base has been updated multiple times this year.

Template Haskell *Template Haskell* [22] is a language extension which can be used to generate Haskell code at compile time. It can be used for code generation too because the AST can be pretty-printed instead of being used in the program (typically with splicing). The pretty-printing is done via a special *Doc* type which can be used to combine multiple snippets and generate items like module headers and comments.

One interesting feature of *Template Haskell* is the use of quotation brackets which can be used to create an AST out of Haskell code. This allows mixing regular Haskell code with AST creation.

With regular *Template Haskell* it is ensured that parsable code will be generated. With the help of typed *Template Haskell* expressions it is possible to support type-safety within expressions. This does not apply to declarations.

It is closely linked to GHC as it is a GHC extension and is widely used (e. g. in the *lens* package [13]). Therefore, it can be assumed that it will be supported well enough in the future. Furthermore, there are numerous tutorials and guides available.

Other options *ghc-exactprint* and *ghc-lib* were studied but discarded. *ghc-exactprint* did not look promising because the main focus is on refactoring existing source code and there are no helpful examples for code generation. *ghc-lib* could probably be used to generate

source code but includes way too many other modules and to little information about code generation. In this case, *ghc-source-gen* would be a better option since it is based on the AST of GHC.

Overview The following table gives a short overview resulting of the evaluation presented above. The respective ratings should not be used without the corresponding evaluation.

Criteria	Comments	Correctness	Support	Options	Usability
Plain text	0	-	+	+	+
haskell-src-exts	+	0	0	0/+	0
ghc-source-gen	-	0	+	0	+
Template Haskell	0	+	+	+	+

Table 3.1: Code generation method comparison

3.2.3 Decision

For this project, *Template Haskell* will be used. In comparison to *Haskell Source Extensions* with *haskell-generate* and *haskell-src-exts-sc*, it is very likely to be actively maintained in the future and to support new Haskell features.

ghc-source-gen would probably be the second best option. But *Template Haskell* has a major advantage when it comes to generating code at compile time and using regular Haskell code as part of the generation (with quotation brackets) as well as typed expressions to ensure type safety. The former feature would enable to relatively easily create a QuasiQuoter which would enable the transformation of a inlined OpenAPI specification into Haskell code at compile time.

As a first step, *Template Haskell* will be used to generate source code files. But as a second step, the option with the QuasiQuoter could be explored and implemented if desired.

In the prototype, *Template Haskell* could easily satisfy the speed requirement. Therefore, this is not likely going to be a problem.

3.3 HTTP Library

The handling of HTTP calls affects the code generation greatly. Depending on the HTTP library and the level of abstraction the generated code offers, the usage of the generated code is affected too. This section describes an evaluation of different approaches. The evaluation was conducted in the first week of April 2020.

3.3.1 Criteria

The handling of HTTP calls should be compared in the following areas:

- **Ease of Use:**
  - How easy is the library to use?
  - Is it easy enough so that it can leak outside the generated code and directly be used by the users of the library?
  - How can the usage be documented?
- **Code Generation:** Can code for the library be generated?
- **Feature richness:** How many features are provided by the library and how many have to be implement yourself?
- **Type safety:** How much is the type system of Haskell used for type safety?
- **Support / Future:** The library should be actively maintained to increase the probability of long-term support for a continued development of the code generator.
- **Future possibilities:** Does the approach give additional possibilities for future use?

The following options have been investigated in order to decide which one should be used for this project.

3.3.2 Traditional HTTP libraries

The following candidates are traditional HTTP libraries in the sense that they abstract the use of HTTP away. The *Code Generator* would generate functions which use one of these candidates. Users of the generated code would only notice on the outskirts of the generated code which library is used under the hood and the abstraction would not leak outside. Since the users of the generated code see only the generated code, the documentation can be fully controlled.

3.3.2.1 http-client

"An HTTP client engine, intended as a base layer for more user-friendly packages" [10].

```
1  import Network.HTTP.Client
2  import Network.HTTP.Types.Status (statusCode)
3
4  main :: IO ()
5  main = do
6    manager <- newManager defaultManagerSettings
7    request <- parseRequest "http://httpbin.org/get"
8    response <- httpLbs request manager
9    putStrLn $
10     "The status code was: "
11       ++ show (statusCode $ responseStatus response)
12   print $ responseBody response
```

Listing 3.12: http-client example from http-client itself

This is a relatively small library, and some other candidates like *http-conduit* depend on it. For HTTPS *http-client-tls* can be used.

The code generation should be straightforward. It does not have many features and helpers. It does not provide any advanced type safety systems, a request body is a text.

The library is actively maintained [8], and since it is the base for many other libraries, no concern about its future maintenance is raised.

Decision This library seems too low level for the usage in this project. There are better alternatives with more features.

3.3.2.2 wreq

"A web client library that is designed for ease of use" [25].

[8] Last commit was in March 2020.

```haskell
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE ScopedTypeVariables #-}

module WreqExample where

import Control.Lens ((&), (.~), (^.), (^?))
import Data.Aeson (FromJSON)
import Data.Aeson.Lens (key)
import Data.Map (Map)
import Data.Text (Text)
import GHC.Generics (Generic)
import Network.Wreq

data GetBody
  = GetBody
      { args :: Map Text Text,
        url :: Text
      }
  deriving (Show, Generic)

instance FromJSON GetBody

basicAsJSON :: IO ()
basicAsJSON = do
  let opts = defaults & param "foo" .~ ["bar"]
  r <- asJSON =<< getWith opts "http://httpbin.org/get"
  putStrLn $ "args: " ++ show (args (r ^. responseBody))

lensAeson :: IO ()
lensAeson = do
  let opts = defaults & param "foo" .~ ["bar"]
  v <- asValue =<< getWith opts "http://httpbin.org/get"
  print $ v ^? responseBody . key "args" . key "foo"
```

Listing 3.13: http-client example from http-client itself

wreq builds on top of *http-client* and is easier to use. It integrates well with *lens* [13]. The ease of use comes from its simplicity.

It has some helpful features like OAuth bearer authentication build in, connection keep-alive, handling of JSON deserialization against a schema and tls support.

It uses the type system to deserialize JSON and for error handling.

It is not very actively maintained [9] but it has its own website with tutorials and examples [10] and it has over 350 stars on github.com. Most blog entries about *wreq* are around 5 years old.

Decision *wreq* is very easy to use. It would be the more suitable solution for this project than *http-conduit* if it were more often used and maintained. If the features of *wreq* were not enough, the switch to another library would be costly.

3.3.2.3 http-conduit

"HTTP client package with conduit interface and HTTPS support." [11]

[9] Last commit was on Jan 25, 2019.
[10] http://www.serpentine.com/wreq/

```haskell
{-# LANGUAGE OverloadedStrings #-}

import Data.Aeson (Value)
import Network.HTTP.Simple

main :: IO ()
main = do
  response <- httpJSON "http://httpbin.org/get"
  putStrLn $
    "The status code was: "
      ++ show (getResponseStatusCode response)
  print $ getResponseHeader "Content-Type" response
  print (getResponseBody response :: Value)
```

Listing 3.14: http-conduit example with Network.HTTP.Simple

```haskell
import Data.Aeson.Parser (json)
import Data.Conduit ((.|), runConduit)
import Data.Conduit.Attoparsec (sinkParser)
import Network.HTTP.Client
import Network.HTTP.Client.Conduit (bodyReaderSource)
import Network.HTTP.Client.TLS (tlsManagerSettings)
import Network.HTTP.Types.Status (statusCode)

main :: IO ()
main = do
  manager <- newManager tlsManagerSettings
  request <- parseRequest "http://httpbin.org/get"
  withResponse request manager $ \response -> do
    putStrLn $
      "The status code was: "
        ++ show (statusCode $ responseStatus response)
    value <-
      runConduit
        ( bodyReaderSource (responseBody response)
            .| sinkParser json
        )
    print value
```

Listing 3.15: http-conduit example with streaming interface

http-conduit builds on top of *http-client* and uses *conduit* [11]. It provides a simple and a streaming interface. The simple interface is simple to use as its name suggests. The streaming interface is a bit more complicated to use. *http-conduit* claims to be efficient. *http-client* provides *http-conduit* as an example of a simpler library. The code generation should be straightforward.

It is very feature rich in the processing of streams. For normal HTTP calls it does not provide many higher level features like OAuth authentication.

[11] https://hackage.haskell.org/package/conduit

It is relatively actively maintained [12] and is maintained together with *http-client*.

Decision The streaming interface adds unnecessary complications for this project and will therefore not be used. The simple interface is easier to use than raw *http-client* but has all its possibilities. The switch from the simple interface to *http-client* would not be that hard if necessary. The future of the simple interface is predictably stable. Because of this reason *http-conduit* with the simple interface is the best traditional http-library for this project.

3.3.3 Servant

"Servant is a set of Haskell libraries for writing type-safe web applications but also deriving clients (in Haskell and other languages) or generating documentation for them, and more" [18].

Compared to the *traditional HTTP libraries* servant does more than just abstract HTTP. It abstracts whole API endpoints, meaning that parameters, body types and response types are predefined. The *Code Generator* would define an API as a type, *servant* would generate methods for this type. This gives away some control but adds many possibilities. In this category, *servant* is the only real possibility discovered.

[12] Last commit was in November 2020.

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE TypeOperators #-}

module ServantExample where

import Data.Aeson
import Data.Proxy
import GHC.Generics
import Network.HTTP.Client (defaultManagerSettings, newManager)
import Servant.API
import Servant.Client

data Position
  = Position
      { xCoord :: Int,
        yCoord :: Int
      }
  deriving (Show, Generic)

instance FromJSON Position

newtype HelloMessage = HelloMessage {msg :: String}
  deriving (Show, Generic)

instance FromJSON HelloMessage

newtype Email = Email String

newtype ClientInfo = ClientInfo String

type API =
  "position" :> Capture "x" Int :> Capture "y" Int :> Get '[JSON] Position
    :<|> "hello" :> QueryParam "name" String :> Get '[JSON] HelloMessage
    :<|> "marketing" :> ReqBody '[JSON] ClientInfo :> Post '[JSON] Email
```

Listing 3.16: Servant Example

A full example can be found at section I.2.

To gain insights about the development with Servant, the development server 4.1 is developed with it. Servant has an initial learning curve and can not simply be used by everyone without prior knowledge. After a short learning phase and when the API is defined the usage is simple. It is clear enough to use that it would be possible to let the users of the generated code directly use the *servant* API. It would also be possible to create an interface and use *servant* only internally.

Code generation could be hard because *servant* uses a lot of type-"magic". The type system of Haskell is extensively used to provide good type safety.

If a *servant* API is defined it could also be a possible code stub for a server.

servant is actively maintained [13] and has its own website [14]. Many blog posts which were written this or last year can be found. The usage of *servant* in the community seems good enough for the library to have a stable future.

3.3.4 Decision

From the traditional category *http-conduit* with the simple interface is the best decision for this project because of the reasons mentioned in the section above. The decision now is between *http-conduit* and *servant*. These are two fundamentally different approaches. With *servant* the future possibilities would be better but there is an overhead in writing and using servant code. For the success of this project so that many people can use type safe OpenAPI libraries in Haskell *http-conduit* is the better approach and therefore will be used.

[13] Last commit was in March 2020.
[14] https://www.servant.dev/

3.4 Error Handling

This section explains how errors are handled within the code generator and the generated code.

3.4.1 Code Generator

If the code generator cannot generate code for a part of the specification (either because of unsupported features or misconfigurations), it should still generate code for the other parts. In order to be able to use the generated code and to develop the code generator, it is necessary to know which parts of the specification could not be processed. This leads to two goals for the error handling:

- **Traceability** A.3.4: Users are able to identify which parts of the OpenAPI specification caused an error or warning and why.
- **Error tolerance** A.3.1: The generator is able to generate some code even if some parts of the specification contain errors. In some cases, this is not possible to achieve reasonably.

3.4.1.1 Command Line Arguments

If non-existing arguments are used or no specification is passed, the user is informed accordingly. A help page is provided by the CLI to ensure all available options can be discovered.

3.4.1.2 Parsing

The process of parsing the OpenAPI specification into a Haskell data model sets the minimal requirement regarding to the strictness of the specification. The parsing allows some misconfigurations (i.e. violating the OpenAPI specification) to slip through, either because it is not easily detectable (e. g. does a reference exist?) or because it is not necessary to enforce these constraints.

The parser used in this project already produces reasonable messages including the location

of the failure and therefore those messages are used.

3.4.1.3 Generating

While generating the code, some constraints (such as enforcing OpenAPI compliance on the whole specification) can be loosened in favor of generating more useful results (e. g. generate some of the functions instead of just failing) for reliability (A.3.1). The user still has to be notified if some parts were left out or have been transformed in a way which cannot be assumed. This is achieved using a *WriterT* Monad transformer in conjunction with a *Reader* Monad which store the currently processed path and produced messages for traceability (A.3.4). If a function is not able to process the information to a reasonable output, ADT like *Either* and *Maybe* are used to indicate the failure. Functions on a higher level can then in turn decide if this is an error they can recover from or not (and probably log an according message either way).

3.4.2 Generated Code

The error handling within the generated code is done using ADT to ensure a clear control flow. The generated functions never throw an exception and catch exceptions thrown by used packages (like the HTTP client). To prevent loss of information, an appropriate ADT like *Either* is used.

Chapter 4

Results

This chapter contains the results worked out in this thesis in order to solve the problems presented by the task description.

4.1 Demo Server

For the purpose of properly demonstrating the code generator and the Stripe API, a demo server is used. The demo use case is documented at 2.2.2. An deployment diagram can be found at Figure F.1.

4.1.1 Deployment

The demo server is dockerized to the GitLab registry at *registry.gitlab.com/hsr-ba-openapi-3/hsr-ba-openapi-3:latest*. A Watchtower [1] is running on the demo server and automatically updates a demo server container via a *docker-compose.yml* file. The demo server uses certs from *Let's Encrypt* [2] for HTTPS communication. The deployment is verified with a test HTTP call to the server, which checks the git revision hash.

4.1.2 Endpoints

/version returns JSON with information about the git revision. This endpoint is used to verify that the correct version is deployed.

/time returns JSON with the current time and the start up time. This endpoint is used as a proof of concept that IO operations can be done.

/inventory returns JSON with a result to the petstore demo server. This endpoint is used to verify that the generated code from the petstore can be used.

/paymentIntent returns *plaintext* with debug information for a paymentIntent from Stripe.

/newPaymentIntentSecret returns JSON with a new secret that can be used for normal payments.

[1] https://github.com/containrrr/watchtower
[2] https://letsencrypt.org/

/newPaymentIntentSepaSecret returns JSON with a new secret that can be used for SEPA payments.

/getCheckoutSessionId returns JSON with a new session ID that can be used for checkouts.

/showSuccess Needs a *sessionid* as a query parameter. Returns JSON with information about the payment after a checkout. It is used to verify that the success from a checkpoint succeeded without relying on browser behavior.

Everything else Serves static files from a directory. */index.html* displays some debug information. *payments/index.html* displays the demo page used in the demonstration. It is the entry point for a normal payment, SEPA payment and a checkout.

4.1.3 Architecture

The demo server uses *servant* [18] and because of that *warp-tls* [3]. *warp-tls* is used directly without any Apache or Nginx server in front of it. Servant was chosen as it is a potential candidate to be used in the generated code. However the development with *servant* highlighted that it is too complex for the purpose of this project, more information can be found at section 3.3 HTTP Library and subsection 3.3.3 Servant. The web page is a Single-page application (SPA) statically delivered without any server side rendering.

[3] https://hackage.haskell.org/package/warp-tls

4.2 Publication

Instructions on how the published code can be used are available at Appendix G Installation Instructions. The Stripe library and code generator are published on GitHub and Hackage. GitHub was chosen because of its popularity with other Haskell libraries.

Code Generator on GitHub

https://github.com/Haskell-OpenAPI-Code-Generator/Haskell-OpenAPI-Client-Code-Generator

Code Generator on Hackage

https://hackage.haskell.org/package/openapi3-code-generator

Stripe Library on GitHub

https://github.com/Haskell-OpenAPI-Code-Generator/Stripe-Haskell-Library

Stripe Library on Hackage

https://hackage.haskell.org/package/stripeapi

4.2.1 CI/CD

4.2.1.1 Code Generator

All CI jobs from the private GitLab repository related to the code generator alone and linting were ported to *CircleCI* [4]. The deployment to Hackage is done manually when necessary.

4.2.1.2 Stripe Library

As the generator is always tested with the Stripe OpenAPI definition as well, the generated code should always compile. But to ensure the code checked in into the repository does compile with the provided example, the Stripe library repository has a pipeline as well [5]. The repository and Hackage are updated when necessary.

[4] https://app.circleci.com/pipelines/github/Haskell-OpenAPI-Code-Generator/Haskell-OpenAPI-Client-Code-Generator

[5] https://app.circleci.com/pipelines/github/Haskell-OpenAPI-Code-Generator/Stripe-Haskell-Library

4.3 Code Metrics

This chapter gives an overview of how much code was written and generated in this thesis. To get an architectural overview, see Appendix F Architecture Overview.

All line counts are without empty lines.

4.3.1 Code Generator

The code of the code generator is written by hand. It consists of around ~3100 lines (~3500 including comments) which are tested by roughly ~1500 lines of test code (consisting of ~500 lines unit tests and ~1000 lines system tests). The productive code is split into 19 modules which contain about ~175 top-level functions.

4.3.1.1 Code Coverage

The tests cover about ~85% of the expressions. The coverage is calculated with a combination of the unit/property tests and the system tests. Additionally, a coverage overlay [2] is used to exclude *Show* and *Eq* instances as well as the Common module which is never evaluated at all while generating code.

4.3.1.2 Extended Compile Tests

To measure how well suited the generator is for generating code from a wide variety of OpenAPI specifications, around ~2700 specifications have been collected. They were used as input to the generator and the resulting code was compiled. Around ~0.7% of them failed during the code generation and ~3.5% during compilation. The main reason for generation failure is that some specifications do not quote the strings true and false and therefore are interpreted as booleans instead of strings [6]. The two reasons for compilation failure are name conflicts (~80%) and not supported reference schemes (~20%).

[6] This is an issue in the specification files as they do not correctly encode their values as strings. The used YAML parser *Aeson* strictly interprets true and false as booleans and does not convert them silently to strings if needed.

4.3.2 Generated Code

As it is possible to generate code from an arbitrary number of specifications, not all generated code can be analyzed here. Exemplary, the Stripe library is used to demonstrate what code results of an OpenAPI specification.

The Stripe specifications has a little less than ~59 000 lines. For this specification, around ~110 000 lines of code are generated (~145 000 lines including comments, ~91 000 excluding imports, language pragmas and module definitions). This code is split into 554 modules which contain about ~1400 functions.

4.3.3 CLI Options

The output of the generated code can be configured using the CLI options. There are 14 options (excluding the help option) which can be discovered using the `--help` option.

4.4 Limitations

This section describes the known limitations of the code generator, they are in no particular order.

4.4.1 Parameter Limitations

Parameters are only supported *in query* and *path* and only scalar types are serialized.

4.4.2 XML and other Transport Data Protocols

JSON is supported for both sending and receiving data. *application/x-www-form-urlencoded* is supported for sending data, but not receiving data. All other transport data protocols such as XML are not supported [7].

4.4.3 Links and Callbacks

Links [8] and *callbacks* [9] are currently not supported in the code generation. They are not implemented because Stripe API does not use them and they would add significant development and maintenance effort compared to the benefits their implementation would offer. The *runtime-expressions* [10] parsing of *links* would have added too much development effort for the scope of this project, but for a future use case they could be beneficial. *Callbacks* should be integrated in an already existing environment and therefore an implementation from the code generator would most likely not be usable. Models for *callbacks* are generated if they are in *components.schemas*.

[7] Note: OpenAPI 3 does not specify a limited set of supported media types and it is therefore impossible to fully implement every possible media type.
[8] https://swagger.io/docs/specification/links/
[9] https://swagger.io/docs/specification/callbacks/
[10] https://swagger.io/docs/specification/links/#runtime-expressions

4.4.4 Only local References

Only references to *components* parts are supported. It is possible to reference other files from an OpenAPI specification. From our test-set of 2710 specifications only 18 specifications used references which are not supported.

- 2 have references directly under *components.schemas*.
- 1 has references which reference deeper nested elements.
- 15 have references to external specification files.

4.4.5 Object Constructors and Default Values

Currently, every field including fields which are optional or have a default value have to be specified to create an object.

4.4.6 Length of Names is limited by the File System

The names used for modules (model and operation names) from the specification are not limited by OpenAPI and therefore can be very long, even exceeding the file systems limit.

4.4.7 Circular References inside the Specification

Not all kinds of circular references inside the schemas can be solved. For example an *allOf* which references itself. From our test-set of 2710 specifications only 1 specification has such a circular reference.

4.4.8 Naming Conflicts

Depending on the naming used in the specification, it is not possible for the code generator to generate meaningful differentiable names. For example, if two names differentiate each other only by the casing of the first letter. From our test-set of 2710 specifications 79 resulted in naming conflicts.

4.4.9 Other Limitations

There are some other limitations which are mostly constraints which are not easily enforceable using the type system. They are reflected in the automatically generated Haddock documentation.

- **additionalProperties** [11] is not supported.
- **not** [12] is not supported.
- **writeOnly** and **readOnly** (2.1.2.2) are not supported.
- **multipleOf** is not supported
- **maximum** is not supported
- **exclusiveMaximum** is not supported
- **minimum** is not supported
- **exclusiveMinimum** is not supported
- **minLength** is not supported
- **maxItems** is not supported
- **minItems** is not supported
- **uniqueItems** is not supported
- **maxProperties** is not supported
- **minProperties** is not supported
- **xml** is not supported

[11] https://swagger.io/docs/specification/data-models/dictionaries/
[12] https://swagger.io/docs/specification/data-models/oneof-anyof-allof-not/#not

Chapter 5

Conclusion

In this chapter the achieved results are compared to other solutions, discussed and future possibilities are introduced.

5.1 Comparison to other Work

An overview of existing work can be found at section 1.4 Research / Existing Work. The most important difference to other published products is that compilable code can be generated from the Stripe OpenAPI specification.

5.1.1 Markus Schirp's OpenAPI Code Binding Generator

During the development it was possible to communicate with Markus Schirp [4] (1.5) about his yet not fully published code generator. A parallel evolution between the two projects was discovered where both projects had the same problems and similar solutions for most problems. Here are the differences:

Stages His code generator works in three stages.

1. Modify the OpenAPI specification itself
2. Generate code
3. Modify the generated code

The code generator from this project uses only the second stage. The first and third stages are done on a per specification basis and are not fully applicable to other specifications.

AST Another difference is that this code generator uses Template Haskell not directly. *semantic* [1] as an AST is/was in evaluation, it would make it possible to generate code for other languages as well.

Servant One of the biggest differences is that *servant* is used. During the thesis *servant* was considered but disregarded because of reasons mentioned in subsection 3.3.3 Servant. Markus Schirp is not entirely satisfied with *servant* because of mostly the same reasons why it was disregarded in this project and would probably use *http-client* directly like in this project.

[1] https://github.com/github/semantic

Cyclic Dependencies Markus Schirp's code generator uses *.hs-boot* [2] files to solve cyclic dependencies, which allows the compile time to be roughly halved. Memory consumption was not measured but is probably less than half. This method solves a problem of this project and is considered for future adoption in Resolve Cyclic Dependencies.

Conclusion Markus Schirp's code generator is very interesting for the future scope of this project, and a possible cooperation could be beneficial. More information about a cooperation is in subsection 5.4.7 Cooperation with Markus Schirp and Future Maintainance.

5.1.2 OpenAPI Tools for Haskell

The most direct comparable product is OpenAPI Tools [3], which was updated since the tests from 1.4 to version 4.3.1 (commit 003165c). The code it produces for the Stripe API specification does still not compile because of invalid names (*AnyOf<accountBusinessProfile&gt*) but a comparison can still be made.

Modules *OpenAPI Tools* generate only two modules (*Types* and *API*). The code generator from this project generates over 500 modules but reexports them to one single module. The *Types* module is around $10\,000$ lines long, while the combined modules for types from this projects generator are at around $40\,000$ lines.

JSON Both products use *Aeson* [4]. To reduce compilation effort most *toJSON* and *fromJSON* instances are generated, OpenAPI Tools use *Generic* for serialization.

Types Complex types like *enum*, *anyOf*, *oneOf* or *allOf* are not supported by *OpenAPI Tools* for Haskell (they are the reason the code does not compile). Floatingpoint type and types for *date* or *date-time* are not configurable.

[2] https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/separate_compilation.html#how-to-compile-mutually-recursive-modules
[3] https://github.com/OpenAPITools
[4] https://hackage.haskell.org/package/aeson

Operations / HTTP Calls *OpenAPI Tools* uses Servant like *Markus Schirp* code generator.

Conclusion *OpenAPI Tools* does not mainly generate Haskell code. The code generator from this project is better suited for Haskell code generation in nearly all aspects.

5.2 Lessons Learned

5.2.1 Learnings from Failure

5.2.1.1 Compilation Time

For the compilation time and especially the compilation time of the generated code not enough attention was paid. In the requirements specification compilation time of the generated code is not covered at all. We expected it to have an impact on our CI/CD pipelines and made sure that a short feedback loop is possible locally (D.2.4).

We did not expect:

- The compilation time increases exponentially with module size.
- The compilation to run out of memory on a modern computer.
- The compilation time of our very simple demo server to take up to 45 minutes.
- The way code is written even if it results in the same results can make a huge difference in compilation time. For example *toJSON* manually written instead of using *Generic*.

Our lessons learned are:

- Compilation effort needs to be considered from the start when dealing with such an amount of code
- Compilation effort needs to be addressed in the requirements specification
- Compilation time is not just different between programming languages, the actions to reduce the effort are vastly different

5.2.1.2 DuplicateRecordFields and Servant

The Haskell language extension *DuplicateRecordFields* seemed like a good idea to make the code less verbose. The reasons why this extension was not useful for our project are described at 3.1.3.5 Disregarded Plans. We further think that we would not use this extension

in other projects. We used *DuplicateRecordFields* in the code generator as a test which led to a fast disregarding of our original plan, but our code generator uses some boilerplate code around the extensions limitation.

Servant was also a technological choice which we thought about using for the generated code. Like *DuplicateRecordFields* we adapted it very early on as it was used in the demo application. The development effort and the complexity in its usage were the key points why we disregarded *Servant*. More information can be found at subsection 3.3.3 Servant. We think that *Servant* is a very useful technology but that it was not best suited for our case.

This project solidified our understanding that:

- Adapting technology early on gives good real world experience and the cost to disregard a technology is much smaller early on.
- Simpler solutions - even if they are less powerful and more verbose - are often better suited because they are easier to reason about.

5.2.1.3 Use Records instead of Tuples

We worked a lot with tuples especially in return types. They are very simple to introduce, especially when additional information needs to be returned from an existing function, and they provide many typeclass instances out of the box. Tuples are problematic to reason about because they and their content do not have names. A *fst* or *snd* provides very little information to the reader compared to the name of a field of a record, the same is true for type declarations.

Every time a tuple is used across functions it needs to be reconsidered to use a record instead.

5.2.1.4 Property Tests

We tried to use property tests in the beginning of the project and realized that it is very hard to come up with properties for code generation. This led to fewer property tests at the end of the project. Still, there were areas where property tests added real value to the project.

One example is the generation of the identifiers were the property tests discovered problems with some unicode characters which are lowercase but cannot be converted to uppercase.

We realized that:

- There are probably domains better suited to property testing than code generation. This is related to the “one-way” nature of generation as properties typically arise from reverse functions.
- There are still areas where property testing, even in a domain not best suited, can be very helpful and they should therefore not be forgotten as a valuable tool.

5.2.2 Learnings from Success

5.2.2.1 Golden Tests

Golden tests [5] were introduced to the project relatively late upon the recommendations of Tom Sydney Kerckhove [3]. They have been very helpful. Merge requests could be understood far better with the changes in the resulting generated code directly visible.

5.2.2.2 Fast Feedback Loop

A custom-made fast feedback loop was essential, especially because the compilation times turned out to be very long. To have a custom-made specification which could be extended alongside newly developed features helped not only for a faster feedback loop, but it was also simpler to reason about new features.

5.2.2.3 Early and strong CI / CD Pipeline

The introduction of the pipeline jobs was not that hard, the complex part were fine adjustments for faster execution times. From our earlier project we already had a good pipeline for our documentation. During development the CI / CD jobs evolved alongside the code gen-

[5] https://softwareengineering.stackexchange.com/questions/358786/what-is-golden-files

erator and the demo project and made its changes more visible. The pipeline has prevented numerous bugs.

System Tests Probably the most valuable thing in our pipelines were the system tests. The code generator can very easily be tested end-to-end as it is a command-line tool and not a classic user interface. These tests led to a very high confidence that the code is actually correct and runs.

Another valuable thing was to divide them into three levels. This led to little effort in adapting the tests and still provided much information about the state of the generator (for example the API had no breaking changes and actual HTTP calls can be made).

5.3 Result Discussion

The results worked out in this thesis satisfy the goals of the task description. As explained in the section 4.4 Limitations, there are some limitations in the current version of the generator which could be addressed in the future. This is mainly due to the limitation of resources (especially time) within this thesis and cannot be directly connected to bad planning or major issues during implementation.

The biggest delay was introduced by the issues regarding compilation time (see 5.2.1.1). These issues could not be resolved completely, but their impact could be limited in a way that compilation is possible in a reasonable time.

Therefore, the project team views the results as satisfying and the project as successful.

5.3.1 Detailed Goals Resolution

The main goals of this project are:

- **OpenAPI code binding generator:** To implement an OpenAPI code binding generator for Haskell in Haskell. This goal is fulfilled.
- **Haskell Library for Stripe API:** To use this generator to generate code bindings that can be used to implement a Haskell library for the Stripe API. This goal is fulfilled.
- **Demo Application:** To implement a simple working demo application that uses the Haskell Stripe API library for online payments to demonstrate the use of the developed tools. This goal is fulfilled.

Special care must be taken to take advantage of the advanced features of Haskell's type system to enforce data consistency at compile time, as well as the current state of the art in Haskell development (e.g. property-based testing). This goal is fulfilled. Data inconsistency is prevented at compile time. Property-based testing is used albeit not that often. Advanced system tests are used additionally.

At the same time, the resulting tools must be understandable, usable and maintainable by a wide community of Haskell developers. This goal is fulfilled. The usage of the generated

code needs no advanced Haskell knowledge. The code generator and the generated code is documented with Haddock. An installation guide and examples how to use are online available (4.2).

While developing the "OpenAPI code binding generatoe", priority should be given to parts of the OpenAPI specification that are required to generate the code bindings required for the "Haskell Library for Stripe API". This is the minimum subset of the OpenAPI specification that must be supported. This goal is fulfilled. The generated code has successfully been used to call various Stripe API endpoints.

All product documentation must be in English, and in a form that is appropriate for continued development, publicly or otherwise. Project documentation that is not relevant to the continued development of the project may be in German. This goal is fulfilled.

5.4 Future Scope

5.4.1 Resolve Cyclic Dependencies

As noted in the comparison to Markus Schirps product in item 5.1.1 Cyclic Dependencies cyclic dependencies could be resolved with a *.hs-boot* file, which would result in significantly shorter compilation time because *CyclicTypes.hs* would not be needed anymore.

5.4.2 Resolve Limitations

All limitations described in section 4.4 Limitations could be resolved in future work on this project. Especially non-scalar parameters (4.4.1) as they would be useful for *expand* with Stripe.

5.4.3 Auto Generated Tests for Generated Code

To automatically generate tests for the generated code would increase trust in the generated code. Especially property-based testing would be a possibility, for example JSON serialization and deserialization could be tested this way. Property-tests for the generated models would make it possible to use them in other property tests outside the generated code.

5.4.4 Code Separation and Server Code Generation

Currently, only client code is generated. Most of the code could be reused to generate server code stubs. These stubs could also be used to support *callbacks*.

To make server code generation possible a better code separation could be preferable. This split up would have to be evaluated in detail but it could look something like this.

- OpenAPI data containers (already separated)
- Extract relevant information into separate data structures specialized for code generation
- Code generation for code used by client and server

  - Models
  - SecuritySchemas
  - Configuration
  - Common Code (static)
- Code generation for client code (mostly Operations)
- Code generation for server code

5.4.5 JSON Schema

The code generator uses a for OpenAPI modified version of JSON Schema [12]. A standalone Haskell code generator for JSON Schemas or only a validator could be beneficial for a wide range of projects.

5.4.6 Quasiquotation

With *Quasiquotation* [6] it would be possible to embed OpenAPI specifications inside Haskell source code. This could be interesting if only parts of the specification could be specified. In conjunction with 5.4.5 it would be possible to generate data types in the form of a JSON Schema inside a Haskell file.

5.4.7 Cooperation with Markus Schirp and Future Maintainance

As noted in subsection 5.1.1 Markus Schirp's OpenAPI Code Binding Generator, a cooperation with Markus Schirp could be beneficial. Some discussion around this subject have already been carried out. Unifying some aspects such as the handling of OpenAPI data seems like the best first step. It could reduce maintenance effort for both products. It is possible that Markus Schirp's generator will also support other programming languages and that the code generator from this project will provide some language specific features.

The development team is interested to develop and maintain this project after the thesis if the product gets some usage. A cooperation with Markus Schirp is interesting for the devel-

[6] https://wiki.haskell.org/Quasiquotation

opment team. The end of the thesis marks an important point in the life of the team members with a lot of change so that no future promises about maintenance can be made.

5.4.8 Viability

This project has no commercial goals. The future maintenance of the library depends mostly on free work. This free work may come from the project team because they have personal interest in the longevity and usage of this product. Other people may take up interest in developing and maintaining the project further because they have some usage of the library (5.4.9) or some other personal interest (e. g. a bachelor thesis). Interest in cooperation already exists as described in 5.4.7. Tom Sydney Kerckhovewill probably use the product in a project of his. To make it easy for other people to work on this project:

- The project is published with a non-restrictive licence.
- Most of the code is documented well. Advanced Haskell features were used with care to make the code easier to understand.
- The code is published on Hackage and the installation instructions are very easy.
- The project team is open to questions and willing to help.

5.4.9 Future Use Cases

People may take up interest in this project because of the following use cases. This list is not complete and people may be interested by other reasons.

- Use Stripe API in Haskell
- Generate code for another OpenAPI specification to connect to an API
- Generate code for a self-written OpenAPI specification to provide code for other developers
- Renew an existing library with the code generator to reduce maintenance cost
- Modify the code generator to generate server code (Models could be reused) (5.4.4)

- Modify the code generator for generalised JSON Schema usage (5.4.5)

Part II

Appendix

Appendix A

Requirements Specification for Code Generator

A.1 Prioritization

Reliability A.3.1 and Changeability A.3.4 are the most important parts of the non-functional requirements.

A.2 Use Cases

A.2.1 Generate Code

The *Code Generator* has only one use case: To generate code out of a OpenAPI specification. The OpenAPI specification may be in a JSON or YAML file. The *Code Generator* must be callable and parameterizable over a CLI.

Typically this is not done very often and started manually, but for frequently changing specification this could be part of a build process.

Output Directory An output directory must be declarable.

Stack Project It must be possible to declare if a Stack project should be generated or not.

Project name For the generated code a name must be declarable. The name can be used for the Stack project and the namespace.

Dry run It must be possible to call the code-generator without actually generating code.

Types Some OpenAPI-schema-types can be mapped to different Haskell-types with different trade offs. For example a *number* can be either mapped to a *Double* or *Numeric.Decimals.GeneralDecimal*. The user of the *code generator* must be able to decide which type to use in these cases.

A.3 Non-functional Requirements

These requirements are the basis for the architectural decisions of the code generator.

A.3.1 Reliability

A.3.1.1 Error Tolerance

Description Additional data structures in the specification should be ignored. If the data structures in the specification are invalid (type which does not exists, missing mandatory fields, etc.) an error message should be produced. If the specification is not valid JSON or YAML, an error message should be produced.

Measure of Fulfilment A valid specification with invalid and additional data inside the document root, the paths object and a schema can still generate valid Haskell code and warns about invalid data structures.

Status This requirement is fulfilled.

A.3.1.2 Naming Tolerance

Description Naming possibilities in the specification should not be reduced by programming language restrictions.

Measure of Fulfilment All names conflicting with Haskell-keywords get either prefixed or postfixed. All non ASCII or non-alphabetic characters get ignored or converted to alphabetic characters.

Status This requirement is fulfilled.

A.3.2 Usability

A.3.2.1 Understandability

Description All error messages and information are understandable. Error messages from third party libraries are excluded from this requirement.

Measure of Fulfilment All error messages use whole english sentences and use the same naming as OpenAPI.

Status This requirement is fulfilled.

A.3.2.2 Learnability

Description All the functionality of the code generator should be learnable in a short time.

Measure of Fulfilment If the argument *--help* is given to the code generator a message describing all the functionality is shown.

Status This requirement is fulfilled.

A.3.2.3 Accessibility

Description People with hearing impairments, visual impairments or mobility impairments should be able to use the code generator

Measure of Fulfilment No sound is used and all information is available as text. Color is just used for visualization of information.

Status This requirement is fulfilled.

A.3.2.4 Automation

Description The code generator should be easily integrated in automatic processes.

Measure of Fulfilment The code generator can be called and parameterized over a CLI.

Status This requirement is fulfilled.

A.3.3 Efficiency

A.3.3.1 Response Time

Description The code should be generated in a reasonable time.

Measure of Fulfilment Per 1000 lines of specification and not exceeding 58000 in YAML format not more than 2 seconds should be used to generate the code. For example: For 5000 lines of specification not more than 10 seconds should be used to generate the code.

Status This requirement is fulfilled.

A.3.3.2 Memory Consumption

Description The generation of the code should use a reasonable amount of memory.

Measure of Fulfilment Per 1000 lines of specification and not exceeding 58000 in YAML format not more than 25 mb should be used to generate the code. For example: For 5000 lines of specification not more than 125 mb should be used to generate the code.

Status This requirement is fulfilled.

A.3.4 Changeability

A.3.4.1 Traceability

Description The code generation outputs enough information to understand what happened and importantly why it happened, so that problems can be identified.

Measure of Fulfilment For every stage of the code generation, the start and the end is discoverable from the output. The original path in the specification is discoverable from the output for every operation and schema. Every error message gives a reason why it was created. For this requirement to be fulfilled, the output can be log-information, cli-output or the generated code.

Status This requirement is fulfilled.

A.3.4.2 Availability

Description The source code is available as open source.

Measure of Fulfilment The source code is available at an open source hosting platform (e.g. gitlab.com) under an open source licence (e.g. MIT licence).

Status This requirement is fulfilled.

A.3.5 Portability

A.3.5.1 Building Process

Description The code generator should be buildable on different platforms.

Measure of Fulfilment The code generator can be built on Windows (version 10), Mac OS (version 10.15) and Ubuntu (version 18.04).

Status This requirement is fulfilled.

A.3.5.2 Usage

Description The code generator should be usable on different platforms.

Measure of Fulfilment The code generator can be used on Windows (version 10), Mac OS (version 10.15) and Ubuntu (version 18.04).

Status This requirement is fulfilled.

A.3.6 Scalability

A.3.6.1 OpenAPI Specification Size

Description Code should be able to be generated from large specification files.

Measure of Fulfilment The code generator can generate code from specifications with up to $50\,000$ lines.

Status This requirement is fulfilled.

Appendix B

Requirements Specification for Generated Code

B.1 Prioritization

Usability B.3.2 and Reliability B.3.1 are the most important parts of the non-functional requirements.

B.2 Use Cases

The requirements for the generated code have no traditionally defined use cases. The generated code can be used to call an API conforming to the specification used for generating the code.

B.3 Non-functional Requirements

These requirements are the basis for the architectural decisions of the generated code.

B.3.1 Reliability

B.3.1.1 Error Tolerance

Description Invalid API answers or network errors should not crash the program.

Measure of Fulfilment An arbitrary answer can be given to an API call and the program does not crash.

Status This requirement is fulfilled.

B.3.1.2 Static Analysis

Description Invalid Request not conforming to the specification should be prohibited at compile time.

Measure of Fulfilment All Requests must conform to a type restriction the values according to the specification.

Status This requirement is fulfilled.

B.3.2 Usability

B.3.2.1 Understandability

Description All error messages and information are understandable.

Measure of Fulfilment All error messages use whole english sentences and use the same naming as OpenAPI.

Status This requirement is fulfilled.

B.3.2.2 Learnability

Description The usage of the generated code should be understandable.

Measure of Fulfilment From the generated code a documentation can be generated and a general documentation about the usage of the generated code is available.

Status This requirement is fulfilled.

B.3.3 Efficiency

B.3.3.1 Time Efficiency

Description The generated code should be fast.

Measure of Fulfilment The difference between a HTTP call over the library compared to a HTTP call with static data should be under 10 ms.

Status This requirement is fulfilled.

B.3.3.2 Memory Consumption

Description The generated code should use little memory.

Measure of Fulfilment The overhead of using the generated code to send a plain message without body and receiving an answer without a body should be under 1 mb.

Status This requirement is fulfilled.

B.3.4 Changeability

B.3.4.1 Traceability

Description If an error occurs it is possible to identify the problem. Error messages from third party libraries are excluded from this requirement.

Measure of Fulfilment All error messages give information about what and why.

Status This requirement is fulfilled.

B.3.4.2 Availability

Description The generated code can be used in a commercial application.

Measure of Fulfilment The generated code is not bound to a license that prohibits commercial use.

Status This requirement is fulfilled.

B.3.5 Portability

B.3.5.1 Building Process

Description The generated code should be buildable on different platforms.

Measure of Fulfilment The generated code can be built on Windows (version 10), Mac OS (version 10.15) and Ubuntu (version 18.04).

Status This requirement is fulfilled.

B.3.5.2 Usage

Description The generated code should be usable on different platforms.

Measure of Fulfilment The generated code can be used on Windows (version 10), Mac OS (version 10.15) and Ubuntu (version 18.04).

Status This requirement is fulfilled.

B.3.6 Scalability

B.3.6.1 Parallelization

Description The generated code should be usable in a threaded environment.

Measure of Fulfilment Parallel requests can be issued with the generated code. This does not mean, that the generated code should give functionality to issue requests parallel, but that it can be used in a parallel environment

Status This requirement is fulfilled.

Appendix C

Requirements Specification for Stripe Library

The Appendix B Requirements Specification for Generated Code also applies to the Stripe library. Only additional requirements are listed in this chapter.

C.1 Use Cases

All relevant API endpoints from https://stripe.com/docs/api to perform an online payment can be called.

C.2 Non-functional Requirements

C.2.1 Usability

C.2.1.1 Learnability

Description The usage of the library should be understandable.

Measure of Fulfilment A documentation about the library is online available.

Status This requirement is fulfilled.

Appendix D

Quality Measures

D.1 Code Style

D.1.1 Programming Guidelines/Linting

As a source of inspiration, https://wiki.haskell.org/Programming_guidelines can be used regarding programming guidelines. But the main source of programming guidelines is *hlint*[1] which is used to automatically lint the source code. This ensures consistency to a reasonable degree. If hints of *hlint* are ignored, this should be done with care and documented.

D.1.2 Formatting

To ensure a consistent formatting, *Ormolu*[2] is used. The formatting is applied during Git pre-commit hooks.

[1] https://hackage.haskell.org/package/hlint
[2] https://github.com/tweag/ormolu

D.2 Automated Testing

Before accepting a merge request, all tests must be successful. If applicable, the tests for the new code should already be included in the merge request. The code coverage of the code generator should be at least 80% for the property/unit test and system tests combined.

D.2.1 Property Tests

Property tests allow testing with randomly generated input data to ensure some properties of a function always holds. For this, the library *Validity*[3] will be used, will be used in conjunction with the widely used testing libraries *hspec*[4] and *QuickCheck*[5].

D.2.2 Unit Tests

Unit tests are created in addition to property tests and test Haskell functions as well. These tests are useful for cases where it is not possible to find good properties. Unit tests are written with *hspec*.

D.2.3 System Tests

System tests test the code generator from a user perspective as they run the executable and check the created output. These tests are created on three different levels. On the first level are the *Compile tests*, on the second the *Client tests* and on the third the *API tests*. Every level is based on the previous level, therefore there are many *Compile tests* and fewer *API tests*.

D.2.3.1 Compile Test

The compile tests use valid OpenAPI specifications which are transformed to Haskell code with the code generator. The output is compiled with GHC and the Haddock documentation is generated. This workflow is shown in Figure D.1. These tests are successful if the compi-

[3] https://github.com/NorfairKing/validity
[4] https://hackage.haskell.org/package/hspec
[5] https://hackage.haskell.org/package/QuickCheck

lation and the generation of the documentation succeeded. The generation is tested with different CLI configurations and the documentation has to cover all of the exported elements (functions, types etc.).

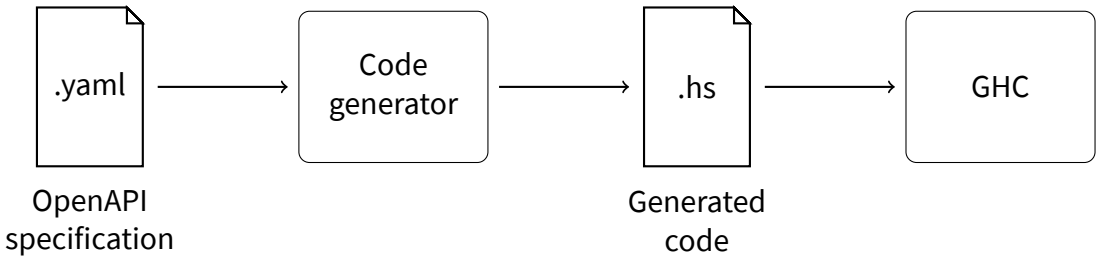

Figure D.1: Workflow Compile Test

D.2.3.2 Client Test

The client tests extend the *Compile tests* with additional client code which uses the generated code, as shown in Figure D.2. The client code calls functions of the generated code and therefore checks, if the types are still compatible. To achieve more confidence, those calls are done within unit tests which can check the results as well. Those tests are successful if the test executable can be built and runs successfully.

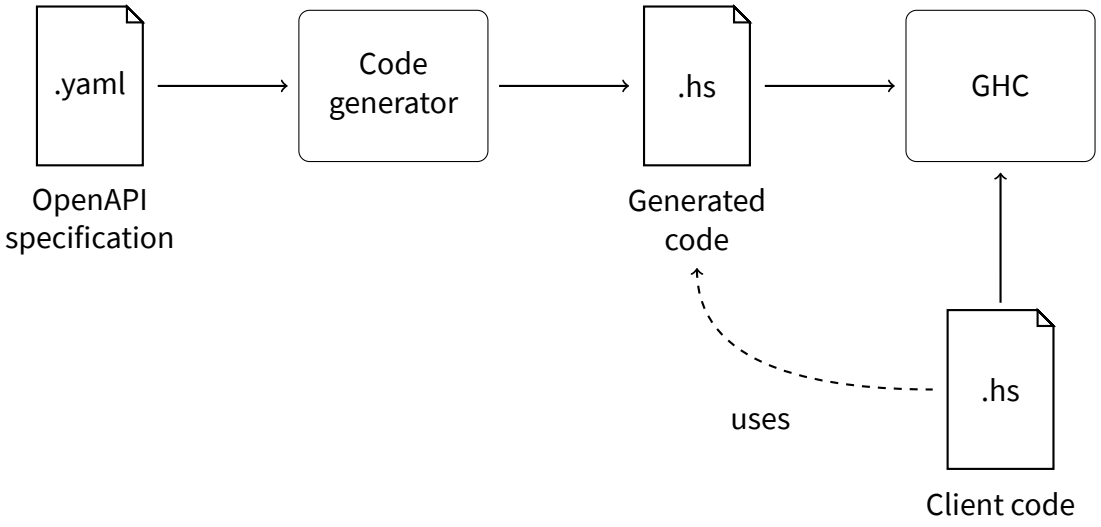

Figure D.2: Workflow Client Test

D.2.3.3 API Test

As shown in Figure D.3, the API tests go one step further as they execute the actual HTTP calls the generated code is intended to do. The client code checks if the results match the expected response. Those tests are successful if the test executable can be built and runs successfully.

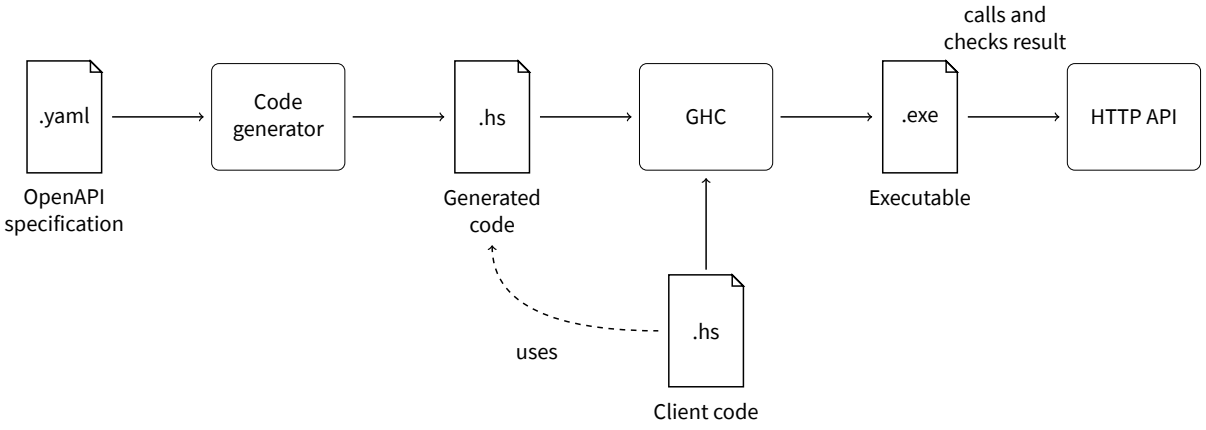

Figure D.3: Workflow API Test

D.2.4 Feedback Loop

During local development it is important to get fast feedback if the changes made are correct. Therefore, it is desirable to have a fast feedback loop. On a first level, this is achieved through property and unit tests which only test if single functions work as expected. Local system tests provide a second level of confidence. If the speed of these tests becomes too slow for local development, it is necessary to create a smaller sample set just for local execution.

Appendix E

Test Plan

This chapter serves the purpose of declaring manual tests. For automatic testing see section D.2 Automated Testing.

E.1 Test Procedure

E.1.1 When will tests be carried out?

During construction the tests are informally without a protocol carried out. During *Construction: Stabilization* all tests are carried out with a protocol.

E.1.2 Non-Functional Requirements

During the test at the end of the *Construction: Feature Freeze* iteration all non-functional requirements are tested. The success is documented directly in the requirements specifications (Appendix A, Appendix B and Appendix C).

E.2 Tests

E.2.1 Test *SYS.LIN* System Test Linux

Test if code can be built and used on a Linux (Ubuntu version 18.04) platform. The linux distribution Ubuntu was chosen, as one of the team members is using Ubuntu.

Prerequisites

- Stripe-API YAML file available
- *Code Generator* source is available
- *Helper Code* which uses the *Generated Code* is available
- Ubuntu version 18.0 or higher is available

Procedure

- Build an executable from the *Code Generator* from source with Stack
- Use the executable with the Stripe-API YAML file to generate code
- Build an executable from the *Generated Code* and *Helper Code*
- Run the executable.

Expectations

- Every build process works without error
- Every executable can be called

E.2.2 Test *SYS.WIN* System Test Windows

Test if code can be built and used on a Windows (version 10) platform. Repeat test SYS.LIN E.2.1 with Windows in place of Ubuntu.

E.2.3 Test *SYS.MAC* System Test macOS

Test if code can be built and used on a macOS (version 10.15) platform. Repeat test SYS.LIN E.2.1 with macOS in place of Ubuntu.

E.2.4 Test *DOC.CGE* Documentation Test Code Generator

Test if the Code Generator can build a good documentation.

Prerequisites

- *Code Generator* source is available

Procedure

- Build documentation from source

Expectations

- Documentation can be built
- Most important parts are well documented. This expectation is up to human interpretation.

E.2.5 Test *DOC.GCO* Documentation Test Generated Code

Test if the Generated Code can build a good documentation.

Prerequisites

- *Generated Code* from the Stripe-API source is available

Procedure

- Build documentation from source

Expectations

- Documentation can be built
- Most important parts are well documented. This expectation is up to human interpretation.

E.2.6 Test *DEM.PAY* Demo Test Payments API

Test Demo Use Case 2.2.2 with the *Payments* API.

Prerequisites

- Demo server is up and running

Procedure

- Declare intent to pay over *Payments* API
- Fill in payment details like amount and a test card
- Complete payment

Expectations

- User never left the page
- Success Message is displayed
- In the log-information the status is available from a server to server call

E.2.7 Test *DEM.CHE* Demo Test Payments API

Test Demo Use Case 2.2.2 with the *Checkout* API. This test is optional.

Prerequisites

- Demo server is up and running

Procedure

- Declare intent to pay over *Checkout* API
- Fill in payment details like amount and a test card
- Complete payment

Expectations

- User was redirected to Stripe and back again
- Success message is displayed
- In the log-information the status is available from a server to server call

E.2.8 Test *PAC.HAC* Package Test Hackage

Test if the package can be used from Hackage.

Prerequisites

- Package is on Hackage

Procedure

- Install the package from Hackage via Cabal.

Expectations

- Package can successfully be used

E.2.9 Test *PAC.NIX* Package Test Nix

Test if the package can be used from Nix. This test is optional.

Prerequisites

- Package is on Nix

Procedure

- Install the package from nix via nix package manager.

Expectations

- Package can successfully be used

E.2.10 Test *USA:COD* Usability Test

In this test a person which has no prior knowledge of this project tries to use the product on his own. Help from the team members is prohibited, only the online available instructions can be used. The person needs to:

- Install the code generator

- Generate code with the help of the code generator

The test person does not need to use the generated code in a project. The reason for this is the lack of testing personal available.

E.3 Protocol

Table E.1: Test protocol

Test	Date	Tester	Result
SYS.LIN	20.05.2020	Remo Dörig	Test was successful
SYS.WIN	20.05.2020	Remo Dörig	Test was successful
SYS.MAC	20.05.2020	Joel Fisch	Test was successful
DOC.CGE	20.05.2020	Remo Dörig	Test was successful
DOC.GCO	20.05.2020	Remo Dörig	This feature is not implemented yet
DEM.PAY	20.05.2020	Remo Dörig	Test was successful
DEM.CHE	20.05.2020	Remo Dörig	Test was successful
PAC.HAC	20.05.2020	Remo Dörig	This feature is not implemented yet
PAC.NIX	20.05.2020	Remo Dörig	This feature is not implemented yet
DOC.GCO	27.05.2020	Remo Dörig	Test was successful
PAC.HAC	27.05.2020	Remo Dörig	Test was successful
PAC.NIX	27.05.2020	Remo Dörig	This feature is not implemented yet
USA.DOC	01.06.2020	Remo Dörig & Flavio F.	Test was successful

E.3.1 Notes about USA.DOC

The test USA.DOC was conducted at a later time, because the code first needed to be published. Flavio could not install Cabal himself on Windows. After some help installing cabal, he could finish the test himself.

E.3.2 Conclusion

All tests which are not optional were successful. The optional test *PAC.NIX* is the only test which is not implemented. *DOC.GCO* and *PAC.HAC* were not implemented on the 20.05.2020 because the code was not published at the time.

Appendix F

Architecture Overview

F.1 Demo System

The demo system is used to demonstrate the *Code Generator* with the Stripe API [19].

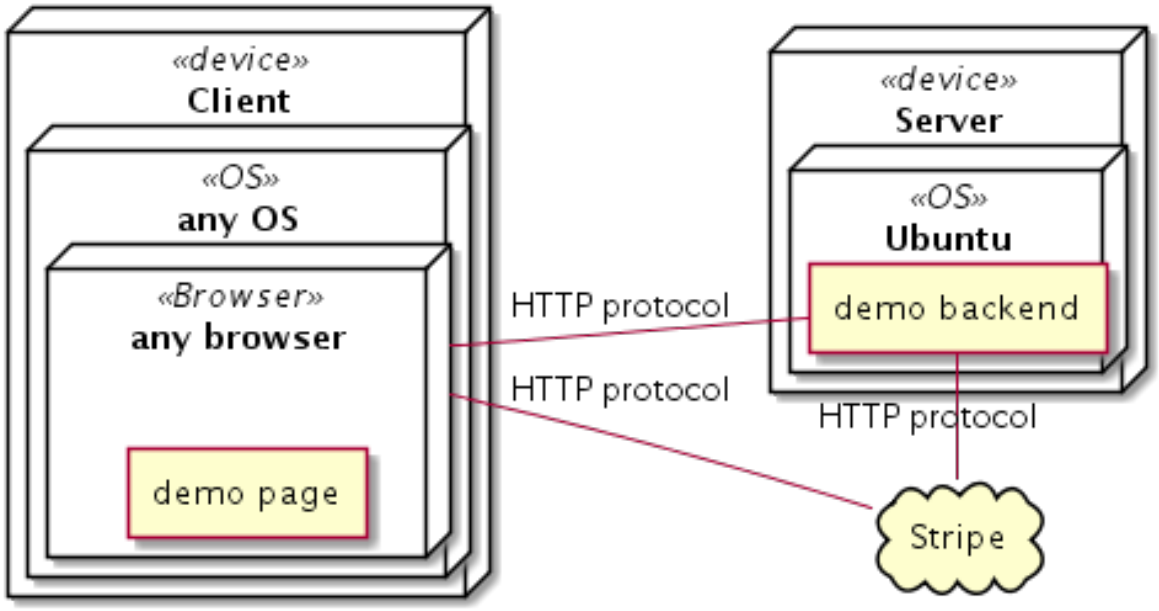

Figure F.1: Demo system deployment diagram

As can be seen in the Figure F.1 the demo runs in the browser. The backend runs on a server provided by the HSR on an Ubuntu system [1]. For the HTTP communication *warp* [24] as a part of the backend is used without any dedicated web server like Apache. The web page is served

[1] For development or presentation the server may run directly on the development computer.

by the backend. The demo page and the backend both communicate with the external Stripe API.

For more information see section 4.1.

F.2 Code Generator

The figure F.2 gives an overview of how the *Code Generator* can be used. The *Code Generator* takes any OpenAPI specification [14] and generates code for Haskell. The *Generated Code* can be used by any Haskell code (*Client-Code*) to communicate with the API specified by the OpenAPI specification.

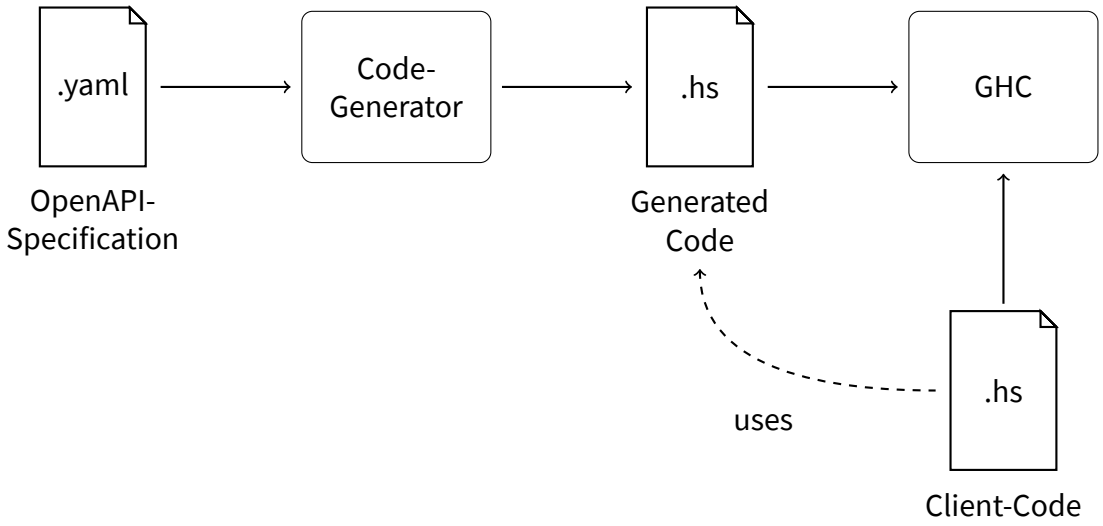

Figure F.2: Workflow Code Generation

F.2.1 Phases

To simplify and minimize the scope the code generation is split up into different independent phases, as visualized in Figure F.3. The generated code from the different phases references each other, but the generation is independent of each other and could therefore run in any order.

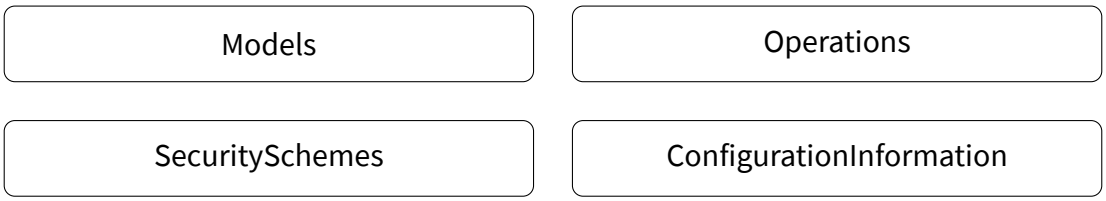

Figure F.3: Phases of the code generator

Operations Transforms *paths* to different operations. Models directly specified in the *paths* are generated in this phase, other models are only referenced.

ConfigurationInformation Transforms *servers* to different configurations. Configurations are used by Operations and contain server information including SecuritySchemes.

Models Transforms *components.schemas* to different types/models. Models are used by Operations. Overview about the models can be found at F.2.2

SecuritySchemes Transforms *components.securitySchemas* to securitySchemes, which can be part of a configuration defined in ConfigurationInformation.

F.2.2 Models

More detailed information about the different types for the models can be found at subsection 3.1.3 Data Models. The Figure F.4 Flow of the code generation of models gives a rough overview over the different kind of models that are generated, in which steps they are created and depending on which conditions they are created. If a schema is a reference, no model is created, only the referenced type is used. The referenced type is created when the generator encounters the concrete type. For primitive OpenAPI-types no type is created normally, if it needs to be referencable, a Haskell type-alias is created.

The generation of a model typically results in:

- **Type:** The type of the generated model
- **Docs:** The declarations used to define the *Type*
- **Dependencies:** A set of dependencies (all the types which are referenced).

If the dependencies form a cyclic dependency, a model is put together with the others in the *CyclicTypes* module or in its own module otherwise.

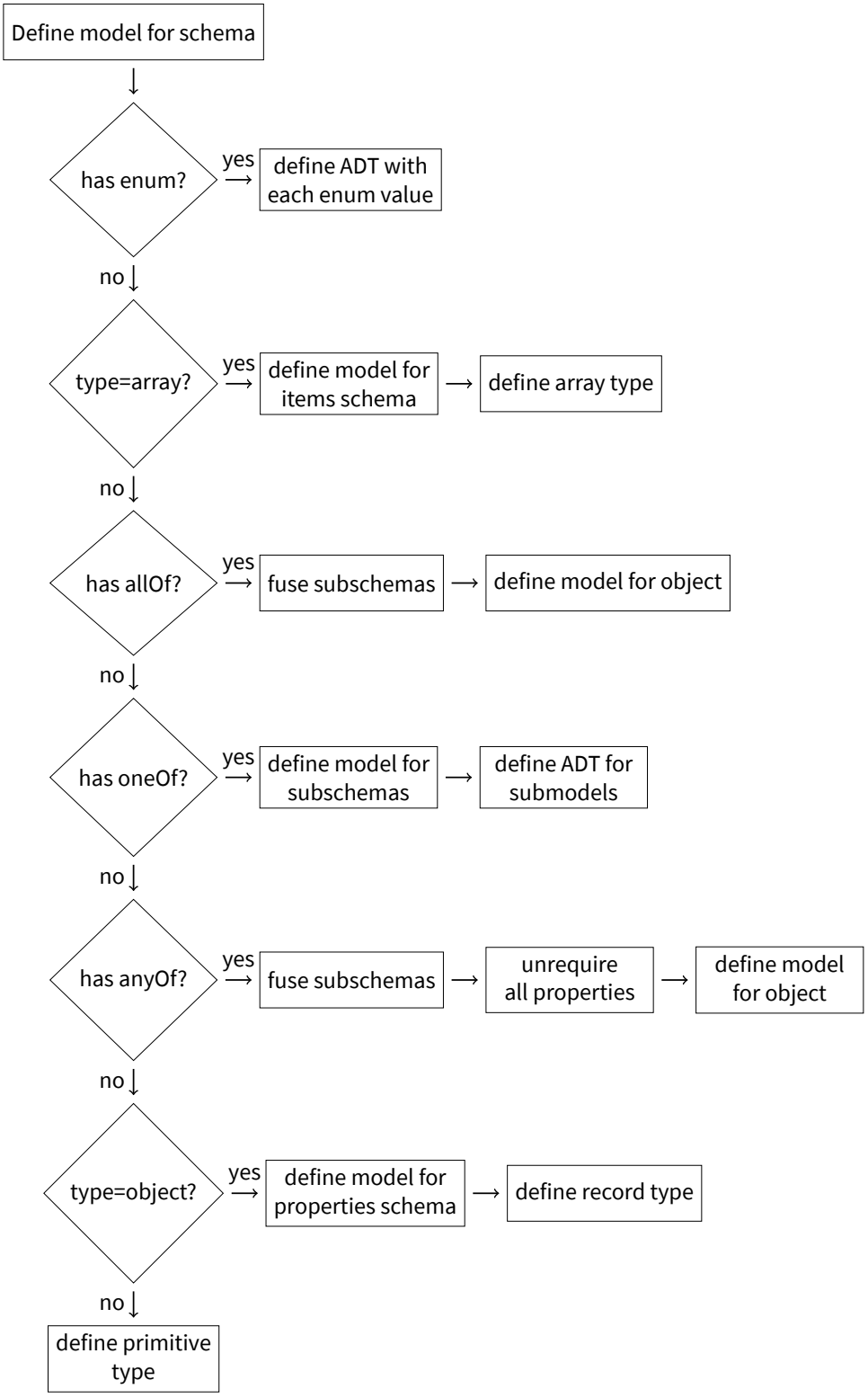

Figure F.4: Flow of the code generation of models

F.2.3 Layering

The layering is visualised in the image Layers for Code Generator. *OpenAPI Data* represents the OpenAPI data structure in Haskell types. *Utils, Flags, Docs, Monad* are helpfull utils which nearly all of the code depends on. *Models*, *Operations*, *SecuritySchemes* and *SecuritySchemes* are four independent phases of the code generation. See F.2.1 for more information about the phases. In the end, *Generate* depends on all four phases and constructs the end result.

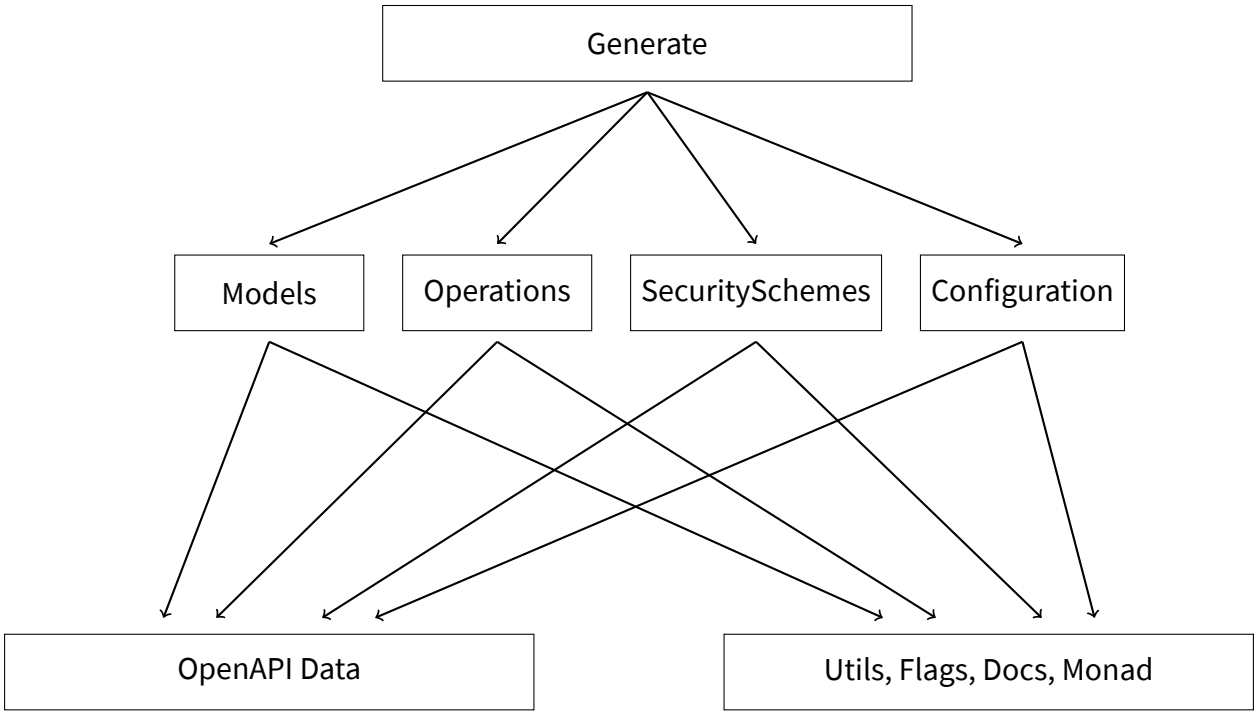

Figure F.5: Layers for Code Generator

F.2.4 Generator Monad

During code generation it is necessary to resolve references in the OpenAPI specification and to trace information. For this purpose an environment in the form of the *Generator Monad* is used. This monad combines the *WriterT* and *Reader* design pattern.

Reader is used to read from:

- **current path:** Used for tracing.

- **references:** Used to resolve references
- **flags:** CLI options

WriterT is used to write log-information.

F.2.5 Resulting Modules

All symbols are globally unique and are reexported in the module *OpenAPI* (Module name can be changed with CLI option `--module-name`). To reduce compile time, the code is split up into multiple modules. Mainly for every operation and for every schema. Schemas with cyclic dependencies are in the module *OpenAPI.CyclicTypes*.

F.2.6 Actual Code Generation

For the actual code generation, an AST is generated with *Template Haskell* and pretty printed to generate code. Information about code generation can be found at 3.2 and for *Template Haskell* at 3.2.2.

F.3 Stripe Library

The Stripe library provides an interface for client code to call the Stripe API, for this the generated code is used. The figure F.6 demonstrates the full workflow of how the Stripe Library can be used, usually client code only uses the Stripe Library.

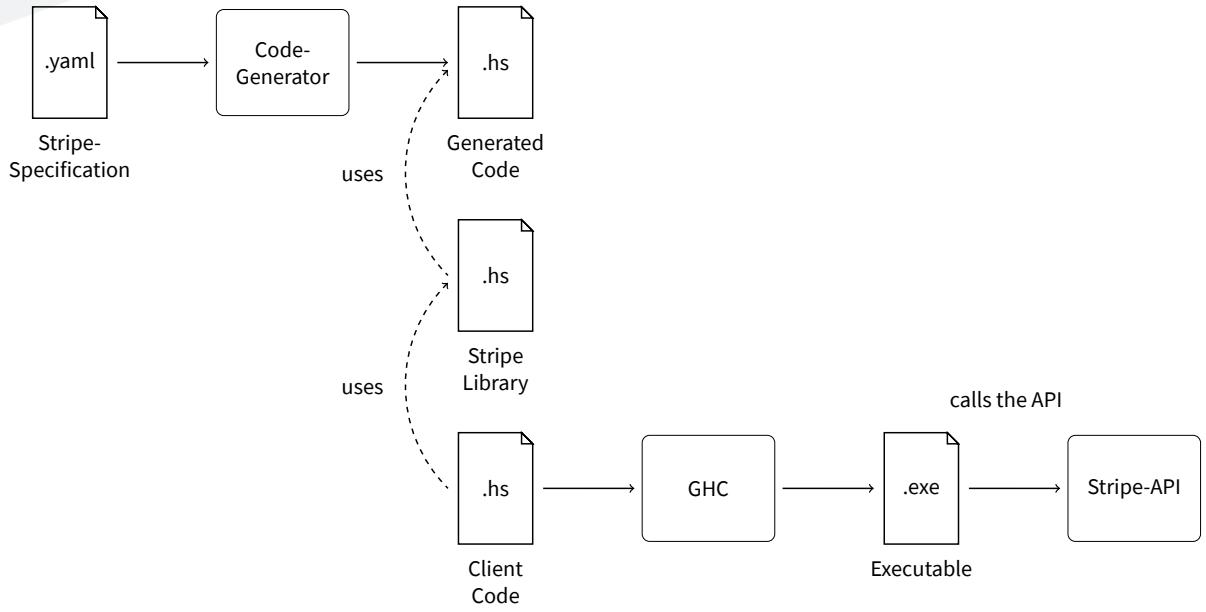

Figure F.6: Workflow Code Generator for Stripe Library

F.3.1 Layers

The following layers visualized in F.7 are present in a program using the Stripe Library.

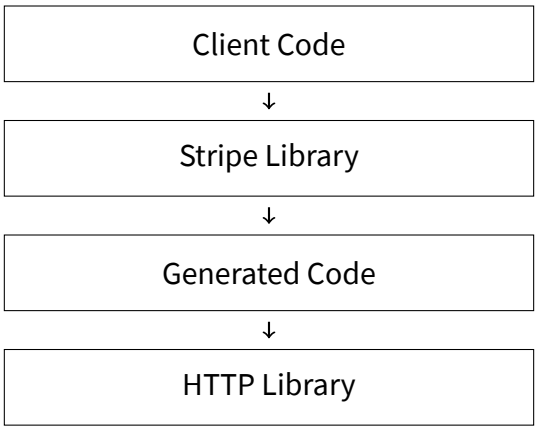

Figure F.7: Layers for Stripe Library

Client Code can be any code which wants to use Stripe.

Stripe Library is provided by this project. It uses the *Generated Code* for Stripe and extends it with some package information and formatting.

Generated Code is generated by the *Code Generator*.

HTTP Library is provided by the Haskell eco system. It is used by the *Generated Code* to make HTTP calls. Apart from the HTTP library, some other common utilities will be present in any generated code, but they are of no concern for the greater picture.

F.3.2 Design Pattern

The generated code uses the *ReaderT* design pattern [1]. This allows users to execute multiple requests using the same configuration instead of passing it explicitly every time.

Appendix G

Installation Instructions

G.1 Code Generator

The installation instructions are also available online at https://github.com/Haskell-OpenAPI-Code-Generator/Haskell-OpenAPI-Client-Code-Generator#readme. The code is available at:

- https://github.com/Haskell-OpenAPI-Code-Generator/Haskell-OpenAPI-Client-Code-Generator
- https://gitlab.com/hsr-ba-openapi-3/hsr-ba-openapi-3/
  (Not public) in the directory *openapi3-code-generator*.
- https://hackage.haskell.org/package/openapi3-code-generator

G.1.1 Install from Source

- `cd openapi3-code-generator` if the private *GitLab* repository is used.
- Install stack [1]
- `stack run --my_specification.yml`

[1] https://docs.haskellstack.org/en/stable/install_and_upgrade/

G.1.2 Install from Hackage

- Install cabal [2]
- `cabal install openapi3-code-generator`

  This may take a while.
- `openapi3-code-generator-exe my_specification.yml`

[2] https://www.haskell.org/cabal/

G.2 Stripe Library

The library is available at:

- https://github.com/Haskell-OpenAPI-Code-Generator/Stripe-Haskell-Library
- https://hackage.haskell.org/package/stripeapi

G.2.1 Install from Hackage

Run `cabal install stripeapi` in the root of a project that wants to use the library.

G.2.2 Example Usage

An example can be found at:
https://github.com/Haskell-OpenAPI-Code-Generator/Stripe-Haskell-Library/blob/master/example/src/StripeHandling.hs

G.3 Generated Code

An example how generated code can be used directly from the file system, can be found at https://github.com/Haskell-OpenAPI-Code-Generator/Haskell-OpenAPI-Client-Code-Generator/ in the directory *example*.

Appendix H

Task Description

H.1 Bachelor Thesis *OpenAPI 3 Code Binding Generator for Haskell*

And its Application to Generate a Library for the Stripe Payment System

H.1.1 Supervisor

Prof. Dr. Farhad Mehta, HSR Rapperswil

H.1.2 Students

- Joel Fisch
- Remo Dörig

H.1.3 Setting

The OpenAPI Specification (originally known as the Swagger Specification) is a specification for machine-readable interface files for describing, producing, consuming, and visualizing RESTful web services. It is claimed to be a broadly adopted industry standard for describing modern APIs. The current version of OpenAPI is 3.0.2, released on 08.10.2018. Unless otherwise specified, this will be the version of OpenAPI referred to in this project. Haskell is a general-purpose, statically typed, purely functional programming language with type in-

ference and lazy evaluation. Developed to be suitable for teaching, research and industrial application, Haskell has pioneered a number of advanced programming language features, especially in the area of type systems. Haskell is increasingly used in a number of applications where efficiency and reliability are valued, particularly in the financial sector. The current version of the Haskell development environment GHC is 8.8.2, released on 16.01.2020. Unless otherwise specified, this will be the version of Haskell referred to in this project. Stripe is a company that provides technical, fraud prevention, and banking infrastructure required to operate online payment systems. Stripe provides an API that web developers can use to integrate payment processing into their websites and mobile applications. Stripe provides an OpenAPI 3.0 specification for this API at https://github.com/stripe/openapi. There is interest within the Haskell community to use Stripe. The current Haskell library (https://github.com/dmjio/stripe) only supports the older Stripe API version 2014-10-071. Since this [1] library is coded manually, it is labour-intensive to keep up to date. A Haskell library that supports the most recent Stripe API version is desired. Stripe provides an OpenAPI 3.0 specification for its API at https://github.com/stripe/openapi. Since the OpenAPI specification is machine-readable, it is possible to use it to automatically generate code bindings in the programming language used to develop a target application, making this task more efficient and less error-prone. Code binding generators for a number of languages exist2, but there is currently no such code binding generator for Haskell 3. The development of an OpenAPI code-binding generator for Haskell could be used to generate an easy to maintain Haskell library for Stripe and would also have much broader use within the Haskell community.

H.1.4 Goals

The main goals of this project are:

- **OpenAPI code binding generator:** To implement an OpenAPI code binding generator for Haskell in Haskell.
- **Haskell Library for Stripe API:** To use this generator to generate code bindings that can be used to implement a Haskell library for the Stripe API.

[1] https://github.com/dmjio/stripe/blame/e00910f59b065bc68335c3f91d956ec0c662b0b4/README.md#L81

- **Demo Application:** To implement a simple working demo application that uses the Haskell Stripe API library for online payments to demonstrate the use of the developed tools.

Special care must be taken to take advantage of the advanced features of Haskell's type system to enforce data consistency at compile time, as well as the current state of the art in Haskell development (e.g. property-based testing). At the same time, the resulting tools must be understandable, usable and maintainable by a wide community of Haskell developers. While developing the "OpenAPI code binding generatoe", priority should be given to parts of the OpenAPI specification that are required to generate the code bindings required for the "Haskell Library for Stripe API". This is the minimum subset of the OpenAPI specification that must be supported. All product documentation must be in English, and in a form that is appropriate for continued development, publicly or otherwise. Project documentation that is not relevant to the continued development of the project may be in German.

H.1.5 Workload

A successful bachelor thesis project counts towards 12 ECTS credit points per student. One ECTS credit point corresponds to a work effort of 30 hours. All time spent on the project must be recorded and documented.

Appendix I

Listings

I.1 Scalar Type Examples

```yaml
# "foo bar"
type: string

# "U3dhZ2dlciByb2Nrcw=="
type: string
format: byte

# "U3dhZ2dlciByb2Nrcw=="
type: string
format: binary

# 7.4
type: number

# 7.4
type: number
format: float

# 7.4
type: number
format: double

# 7
type: integer

# 7
type: integer
format: int32

# 7
type: integer
format: int64

# false
type: boolean
```

Listing I.1: OpenAPI example scalar types

```haskell
{-# LANGUAGE OverloadedStrings #-}

module ExampleScalarTypes where

import qualified Data.ByteString as B
import qualified Data.Int as I
import qualified Data.Text as T

exampleString :: [T.Text]
exampleString = ["hello", "world"]

exampleByte :: B.ByteString
exampleByte = B.empty

exampleBinary :: B.ByteString
exampleBinary = B.empty

-- note, that this is not actually an infinite number.
exampleNumber :: Double
exampleNumber = 7.4

exampleFloat :: Float
exampleFloat = 7.4

exampleDouble :: Double
exampleDouble = 7.4

exampleInteger :: Integer
exampleInteger = 7

exampleInt32 :: I.Int32
exampleInt32 = 7

exampleInt64 :: I.Int64
exampleInt64 = 7

exampleBoolean :: Bool
exampleBoolean = False
```

Listing I.2: OpenAPI Haskell example scalar types

I.2 Research HTTP Library

The following code example is copied from https://docs.servant.dev/en/stable/tutorial/Client.html and only slightly modified to better demonstrate, that the functions are autogenerated.

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE TypeOperators #-}

module Main where

import Data.Aeson
import Data.Proxy
import GHC.Generics
import Network.HTTP.Client (defaultManagerSettings, newManager)
import Servant.API
import Servant.Client
import qualified Servant.Client.Streaming as S
import Servant.Types.SourceT (foreach)

data Position
  = Position
      { xCoord :: Int,
        yCoord :: Int
      }
  deriving (Show, Generic)

instance FromJSON Position

newtype HelloMessage = HelloMessage {msg :: String}
  deriving (Show, Generic)

instance FromJSON HelloMessage

data ClientInfo
  = ClientInfo
      { clientName :: String,
        clientEmail :: String,
        clientAge :: Int,
        clientInterestedIn :: [String]
      }
  deriving (Generic)

instance ToJSON ClientInfo

data Email
  = Email
      { from :: String,
        to :: String,
```

```haskell
        subject :: String,
        body :: String
      }
  deriving (Show, Generic)

instance FromJSON Email

type API =
  "position"
    :> Capture "x" Int
    :> Capture "y" Int
    :> Get '[JSON] Position
    :<|> "hello"
      :> QueryParam "name" String
      :> Get '[JSON] HelloMessage
    :<|> "marketing"
      :> ReqBody '[JSON] ClientInfo
      :> Post '[JSON] Email

api :: Proxy API
api = Proxy

-- functions position, hello and marketing are autogenerated
position :<|> hello :<|> marketing = client api

queries :: ClientM (Position, HelloMessage, Email)
queries = do
  pos <- position 10 10
  message <- hello (Just "servant")
  em <-
    marketing
      (ClientInfo "Alp" "alp@foo.com" 26 ["haskell", "mathematics"])
  return (pos, message, em)

main :: IO ()
main = do
  manager' <- newManager defaultManagerSettings
  res <-
    runClientM
      queries
      (mkClientEnv manager' (BaseUrl Http "localhost" 8081 ""))
  case res of
    Left err -> putStrLn $ "Error: " ++ show err
    Right (pos, message, em) -> do
      print pos
      print message
      print em
```

Listing I.3: Servant: Querying an API

I.3 OpenAPI Tools Code Generators

```
{-# LANGUAGE ConstraintKinds #-}
{-# LANGUAGE CPP #-}
{-# LANGUAGE DefaultSignatures #-}
{-# LANGUAGE DeriveDataTypeable #-}
{-# LANGUAGE DeriveFoldable #-}
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE DeriveTraversable #-}
{-# LANGUAGE ExistentialQuantification #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE MonoLocalBinds #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE NamedFieldPuns #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE PartialTypeSignatures #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE RecordWildCards #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TupleSections #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeOperators #-}
```

Listing I.4: Language extensions used by the generated code from OpenAPI Tools

Appendix J

Domain Diagrams

The following domain models are included here for completeness. The domain should be understandable too with only the information presented in section 2.1 Code Generation.

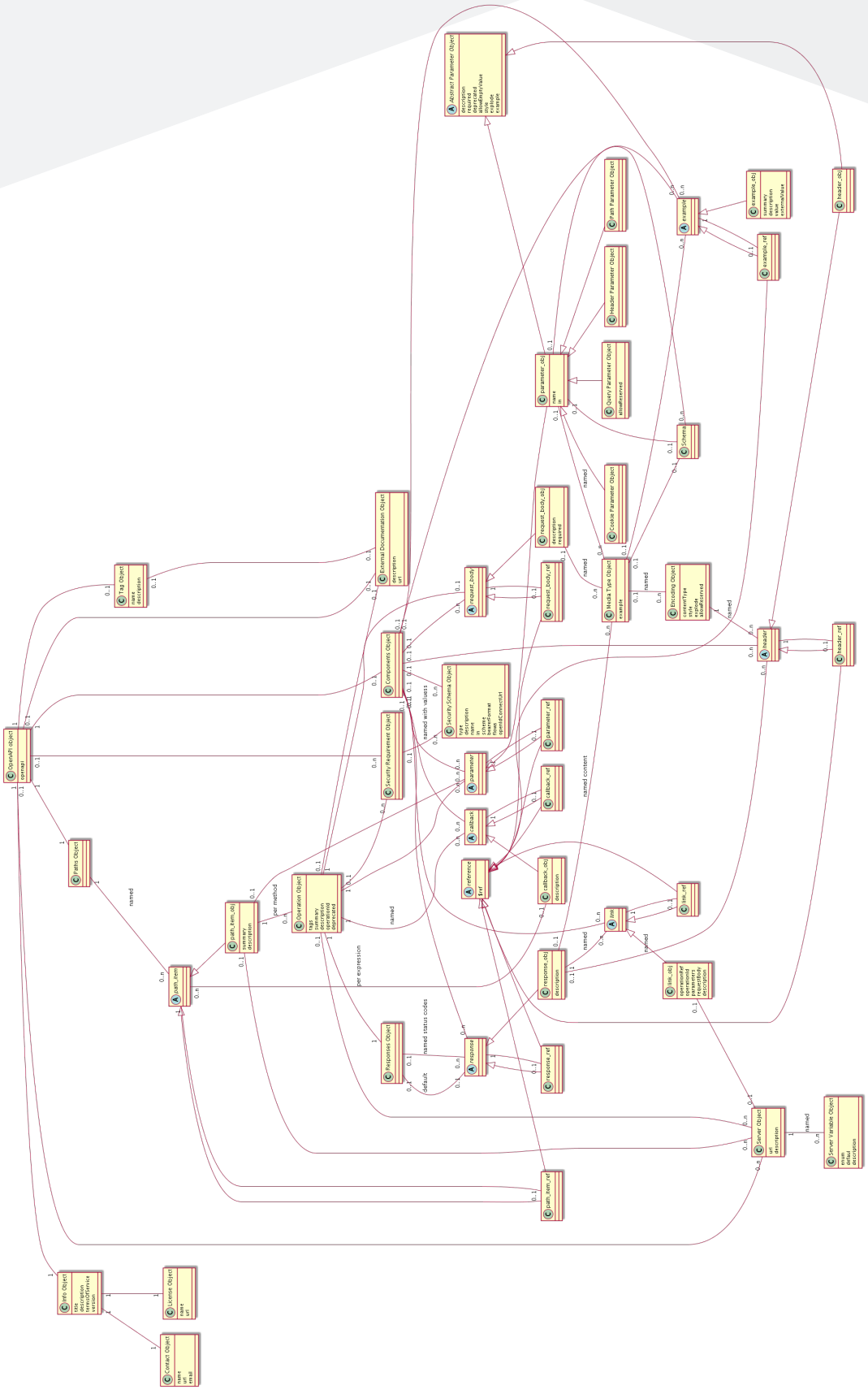

Figure J.1: Complete Domain model of the OpenAPI Specification

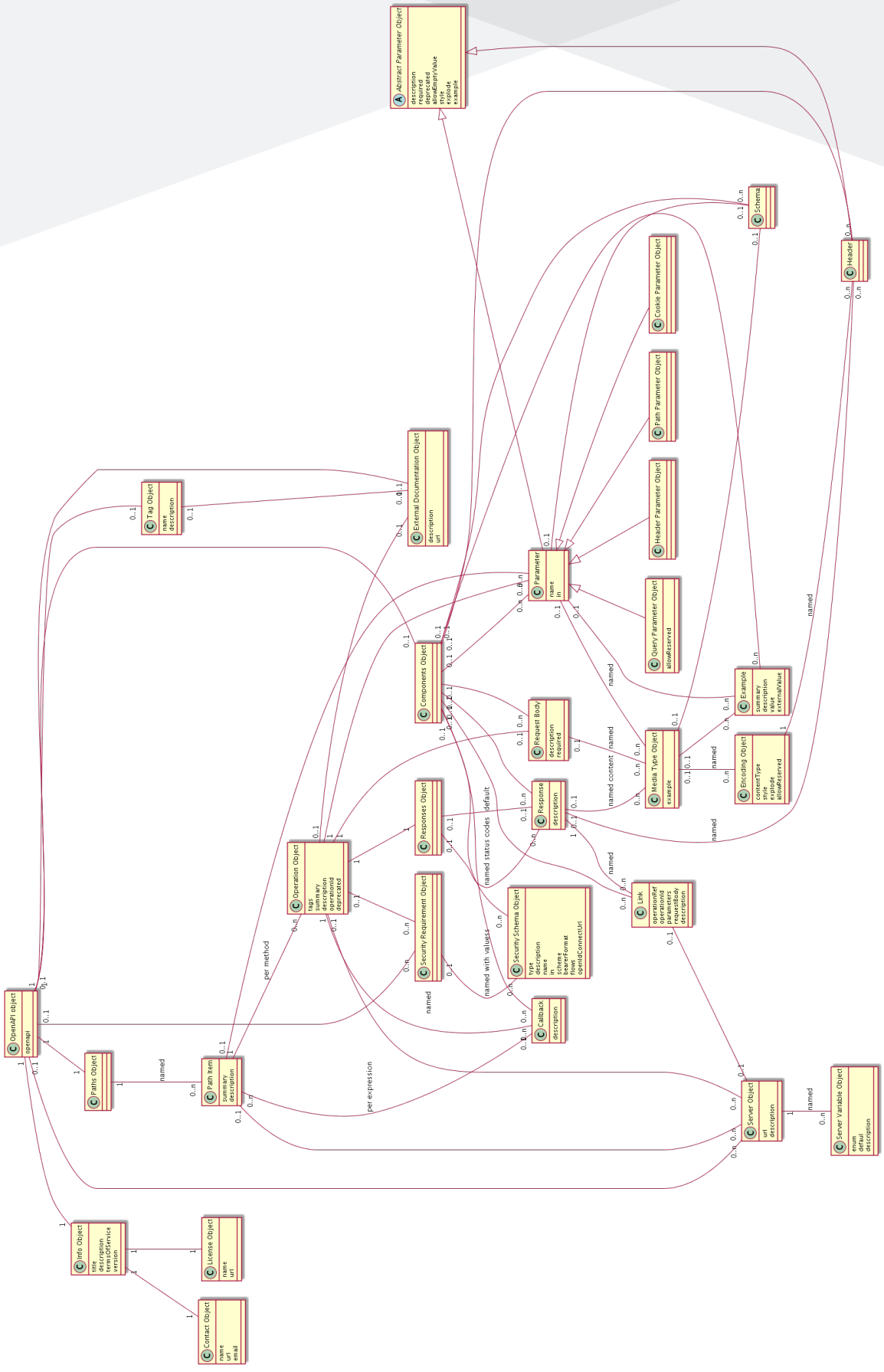

Figure J.2: Domain model of the OpenAPI Specification without reference types

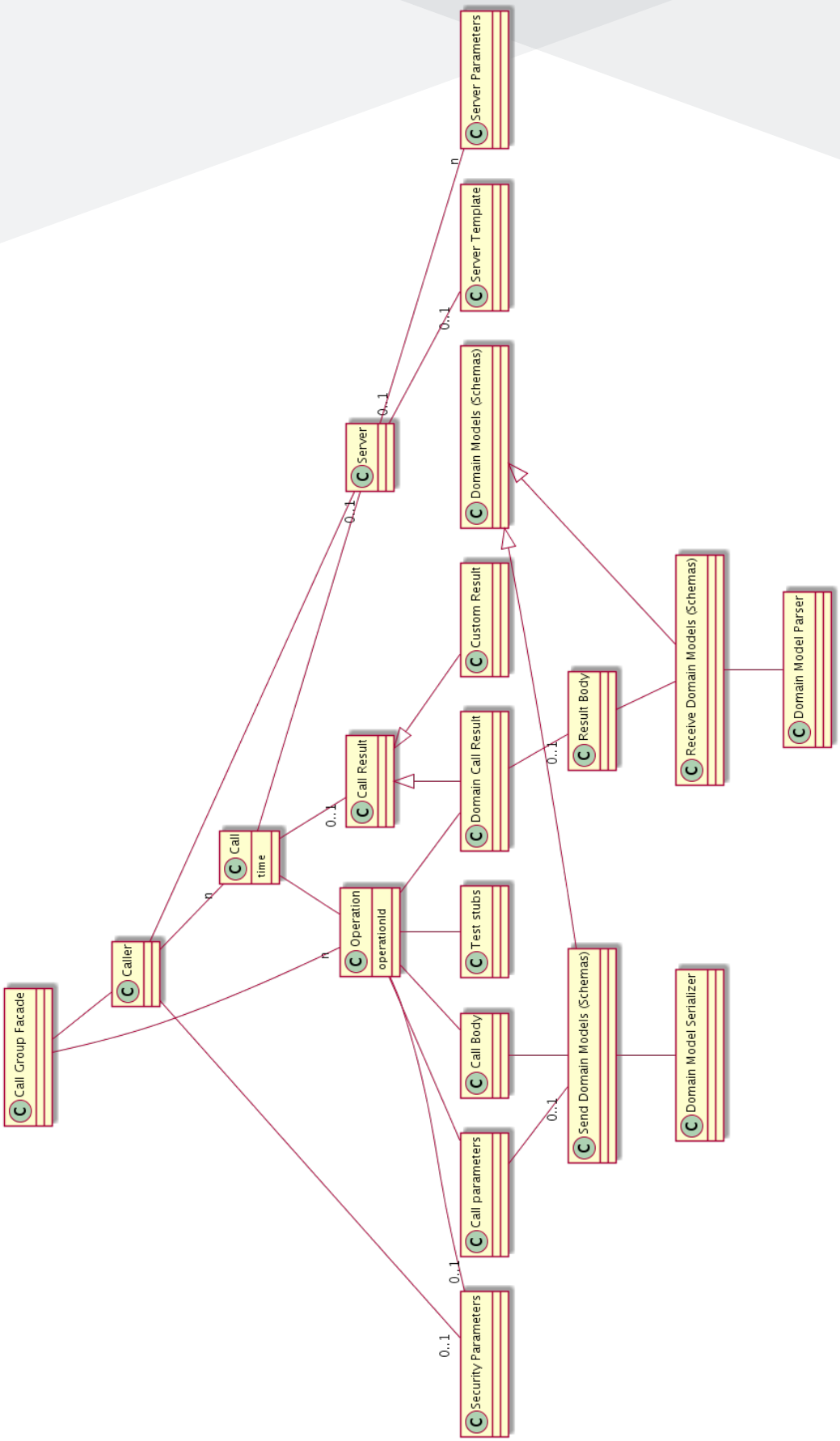

Figure J.3: Informal domain model of the generated code as originally theorized

Addendum

Bibliography

[1] Michael Snoyman. *The ReaderT Design Pattern*. June 12, 2017. URL: https://tech.fpcomplete.com/blog/2017/06/readert-design-pattern (visited on 06/03/2020).

[2] Matthias Benkort. *Continuous integration in Haskell*. Apr. 19, 2020. URL: https://medium.com/@_KtorZ_/continuous-integration-in-haskell-9ad2a73e8e46 (visited on 05/29/2020).

[3] Tom Sydney Kerckhove. Private Correspondence. May 18, 2020.

[4] Markus Schirp. Private Correspondence. June 6, 2020.

[5] *ghc-source-gen*. URL: https://github.com/google/ghc-source-gen (visited on 03/05/2020).

[6] *Glasgow Haskell Compiler User's Guide: Language options*. URL: https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#extension-DuplicateRecordFields (visited on 03/06/2020).

[7] *Haskell Source Extensions*. URL: https://github.com/haskell-suite/haskell-src-exts (visited on 03/05/2020).

[8] *haskell-generate*. URL: https://github.com/bennofs/haskell-generate (visited on 03/05/2020).

[9] *haskell-src-exts-sc: Pretty print haskell code with comments*. URL: https://github.com/achirkin/haskell-src-exts-sc (visited on 03/17/2020).

[10] *http-client: An HTTP client engine*. URL: https://hackage.haskell.org/package/http-client (visited on 03/12/2020).

[11] *http-conduit: HTTP client package with conduit interface and HTTPS support.* URL: http://hackage.haskell.org/package/http-conduit (visited on 03/25/2020).

[12] *JSON Schema*. URL: https://json-schema.org/ (visited on 04/01/2020).

[13] *lens: Lenses, Folds and Traversals*. URL: https://hackage.haskell.org/package/lens (visited on 03/05/2020).

[14] *OpenAPI Specification Version 3.0.3*. URL: https://swagger.io/specification/ (visited on 02/28/2020).

[15] *openapi-generator.tech feature set*. URL: https://openapi-generator.tech/docs/generators/javascript#feature-set (visited on 02/28/2020).

[16] *openapi-generator.tech from OpenAPI Tools*. URL: https://openapi-generator.tech/ (visited on 03/11/2020).

[17] *QuickCheck: Automatic testing of Haskell programs*. URL: https://hackage.haskell.org/package/QuickCheck (visited on 03/12/2020).

[18] *Servant*. URL: https://www.servant.dev/ (visited on 06/05/2020).

[19] *Stripe*. URL: https://stripe.com/ (visited on 03/28/2020).

[20] *Stripe API documentation*. URL: https://stripe.com/docs/api/ (visited on 03/04/2020).

[21] *Stripe OpenAPI specification*. URL: https://github.com/stripe/openapi (visited on 03/04/2020).

[22] *Template Haskell*. URL: http://hackage.haskell.org/package/template-haskell (visited on 03/05/2020).

[23] *Time - HaskellWiki*. URL: https://wiki.haskell.org/Time (visited on 03/06/2020).

[24] *warp: A fast, light-weight web server for WAI applications.* URL: https://hackage.haskell.org/package/warp (visited on 03/28/2020).

[25] *wreq: An easy-to-use HTTP client library.* URL: https://hackage.haskell.org/package/wreq (visited on 03/25/2020).

Complete Content