Describing Inter Parameter Constraints in Web APIs Using Dependent Types

Master's Thesis

Gerben Oolbekkink

Describing Inter Parameter Constraints in Web APIs Using Dependent Types

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Gerben Oolbekkink
born in Utrecht, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Adyen
Simon Carmiggeltstraat 6-50, 1011 DJ
Amsterdam, the Netherlands
www.adyen.com

Describing Inter Parameter Constraints in Web APIs Using Dependent Types

Author: Gerben Oolbekkink
Student id: 4223896

Abstract

Web APIs are being used for increasingly larger and complex use cases. Right now it can be hard to make sure that what is documented about an API is correct everywhere and to know if a change will have impact on the users of a web API.

When details are missing in an API specification users of that API need to make assumptions about how the API works. The creators of the web API also wants to know what users expect from the API. There are two sides to this problem, enforcing that the implementation is actually the same as what is specified, and making it possible to define API specifications as precise as possible.

The type system of a programming language is a useful tool for enforcing the structure of an implementation. In this thesis we use a dependent type system to enforce an API specification in the implementation. By using the dependent type system we can define additional, more specific, constraints on the API. These constraints are more specific than constraints expressible in possible research.

With this approach we can be sure that the specification and implementation are actually describing the same API. And with the added flexibility we can create a more complete description of web APIs.

Preface

This project bridges a gap between two fields in Computer Science. First the field of Programming Languages, which provides a really interesting concepts that we am applying in the Software Engineering field. We use type systems, and more specifically dependent type systems to describe web API specifications. Using this we aim to improve how web APIs are developed and how API specifications are written.

At Adyen I have been able to see how web API development is done at scale. To see what the challenges and opportunities are when developing web APIs. I would like to thank Henk Grent and Aleksei Akimov for their guidance and visionary ideas.

Many thanks to Maurício Aniche for his feedback and for representing the software engineering side of this project.

Finally I would like to give a special thanks to Casper Poulsen for his continuous involvement in this project, for his great ideas and for introducing me to many interesting concepts and technologies.

Gerben Oolbekkink
Delft, the Netherlands
April 1, 2022

Contents

List of Figures

Chapter 1

Introduction

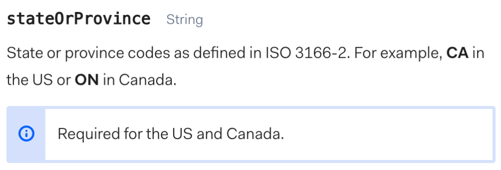
Web APIs (Application Programming Interface) are provided by many computer systems to allow external systems to interact with the service they provide. Examples of this are the Twitter API, which allows third party applications or bots to post tweets, another example is the Adyen API which allows applications to initiate payments. Web APIs provide a set of endpoints that can be used by external systems to request data or perform actions. They are often publicly available on the internet, allowing any system that is connected to the internet to interact with these APIs. Usually these APIs communicate in the JSON (JavaScript Object Notation) format, but there are also other format that are used for communication.

Over the time web APIs have grown larger and more complex. To help with this complexity web API specifications are created, these specifications describe how a web API can be interacted with. For this it describes expeted input and output for every endpoint in the API and it describes general information about the API. API specifications are widely used in modern web APIs[27].

An API specification is in essence a contract between the web API and it's users. The users of the API know what they can expect and the developers of the web API know what they can and cannot change without hurting the user. From this viewpoint it is important that an API specification is as complete as possible.

The standard way of creating a specification for a web API is using an OpenAPI document. This is a JSON file containing all the endpoints that are present in a web API, what kind of values they accept and what values they return. These files also contain metadata about the API, such as where it can be found and who is developing the API. The OpenAPI specification[19] describes how OpenAPI documents are defined.

An API specification document written conforming to the OpenAPI specification can express the structure of most APIs. It is also used to define which parameters are accepted on a specific endpoint and what the values of these parameters can be, for example if a parameter must be a string or a number. There are however sometimes, especially in more complex web APIs, specific structures or constraints on a specific structure that are not expressible in an OpenAPI document. For example in Figure 1.1, where a plain text note is added to a field with additional information about the contents of that field under specific circumstances. This case, where the OpenAPI specification does not have a way to describe parts of an API, is what we are investigating in this thesis. Complex constraints which involve multiple parameters in a request is also called: Inter Parameter Constraints.


stateOrProvince String

State or province codes as defined in ISO 3166-2. For example, **CA** in the US or **ON** in Canada.

Required for the US and Canada.

Figure 1.1: An example of an inter parameter constraint in the Adyen API

Inter Parameter Constraints are static constraints that can be present in a web API. They impose constraints on two or more fields of an object inside a web API request. For example a field that is required when another field has a specific value. For example in the case previously shown, where an address object with a state field which is required when the country is The United States or Canada, and is optional in all other cases. Defining these specific details about the web API gives users of the API more insights in the implementation details of the API. On the other side it helps the developers of the API implementation with knowing which implementation details are possibly used by users of the API and cannot be changed without notifying the users of the API.

Knowing that these constraints exist when implementing an integration with an API or when updating an API implementation helps in several ways. Being able to validate requests before sending them to a server reduces the amount of invalid requests. Knowing which constraints are communicated with users prevents accidentally removing or adding constraints without the users' knowledge. Formally defining constraints makes it possible to automate validating these constraints. Having these constraints only defined in the documentation and not in a more formal way makes it impossible, or very hard, to create automatic tooling that validates these constraints or validates if an implementation is correct.

There already exist possible solutions to this, for example by adding additional expressions which reason about the structure of an object in an OpenAPI specification file. This is proposed in Martin-Lopez et al. [14] and Oostvogels et al. [20]. In this research the goal is to communicate more fine grained details about how an API is implemented. In these solutions there is usually an existing system that has inter parameter constraints. The inter parameter constraints are extracted from the implementation and formally documented. The work by Grent et al. [10] automates this approach by automatically analysing documentation and the control flow of the API implementation.

But there are cases where the domain specific languages in existing research where it is not possible to express every constraint that we can find in web APIs. For example a constraint on a list, where a field is dependent on the sum of the values in a list. In this thesis we try to find a more general solution for this problem. For example when the sum of a list of values is constrained by some other value. In this case the existing domain specific languages for expressing inter parameter constraints are not able to express this constraint.

First we encode an OpenAPI document as a type in a programming language. The research objective for this is the following:

How can we enforce the contract of a web API using the programming language?

This means that the specification enforces the structure of the program, this enforcing is handled by the programming language. If the API implementation conforms to the type, meaning that all routes defined in the specification must also be present in the implementation and vice-versa. The input and output values for every route must also be correct. In this thesis we define a formal structure for this type. When an OpenAPI document is encoded as a type in a programming language the users of the web API can also make use of this definition.

We implement this formal structure of OpenAPI documents as a type in the dependently typed programming language Agda. This programming language has a very powerful type system which is allows us to implement the API specification semantics. The dependent type system allows us to define the types in much detail. This makes it possible to implement the formal structure.

We use the typesystem provided by the Agda programming language, which can express fine-grained types, to create web API specifications which can express constraints in more detail than existing technologies. We apply the following research objective for this:

How can we make web API contracts more expressible and more complete?

Dependent types allow us to define arbitrary decidable constraints on the structure of the input and output data, which is embedded in the type system. With this it is possible to have compile time checks in place which check if there are no wrong assumptions about the constraints present in the input or output data.

We use this dependently typed definition to add inter parameter constraints to the OpenAPI document type. With dependent types we are able to define arbitrary inter parameter constraints to a specification. We show how we can express constraints found in previous research, and also how we can express constraints which are not expressible in previous research. This implementation with dependent types can describe web APIs in a much more precise way.

1.1 Contributions

- We describe OpenAPI documents as a type in a programming language and use this to make the semantics of OpenAPI documents explicit. (Chapter 3)
- We create a formal representation of OpenAPI documents in the Agda programming language. With this formal representation it is possible to construct an OpenAPI document as a type in the Agda programming language. (Chapter 4)
- A validator for JSON input on an OpenAPI document is made. This validator checks if an input is conforming to what is specified in the OpenAPI document. (Chapter 4)
- The formalized implementation of OpenAPI documents is expanded to support any decidable inter parameter constraints. (Chapter 5)

In Chapter 2 we introduce the domain further by explaining how web APIs we are talking about work. In Chapter 3 we describe OpenAPI documents as a type and make the semantics of OpenAPI documents explicit. We then formalize this specification in the Agda programming language (Chapter 4). This formalized specification is then used to formalize OpenAPI documents with inter parameter constraints in Agda (Chapter 5). We then discuss in what ways this solution improves the way we look at OpenAPI documents (Chapter 6).

Chapter 2

Introduction to Web APIs

First we give an overview of the context, what a web API is in the context of this thesis and what technologies are currently used to create web APIs. We also show how web API specifications are made and what role they play. From this we explain what inter parameter constraints are, why they are relevant and why research on these inter parameter constraints is interesting.

Web API specifications describe what a valid input for a web API is, anything that is not valid according to the specification should not be accepted by the API. Any value that is not rejected by the specification should be processable by the API. Valid input can still be rejected when an action is impossible to execute, like removing a value that does not exist or if there is specific logic forbidding a specific combination of values.

These specifications should include as much information as possible, to make the margin for error as small as possible. Everything that is known without knowing the internal state of the service should be documented in the specification. One part of the specification that is still an active research subject is Inter Parameter Constraints (Section 2.3), these are not yet generally supported by API specifications, but there are already some ideas on how they can be supported (Section 3.1).

2.1 Web APIs

In this section we briefly describe what a Web API is. We go over how a web API is interacted with by an end user, and what parts of the web API are usually implemented by the developer of the API and which parts are part of an existing framework or library.

We focus on web APIs which are created for usage by third parties, the source code for these web APIs is not generally available. In this case everything a developer who is integrating with an API knows is based on the following:

- Documentation. Written text, interpreted by a human.
- Specification. Formal definitions of the structure of the API, interpreted by humans and computers.
- Responses. Data returned from the web API when a request is made.

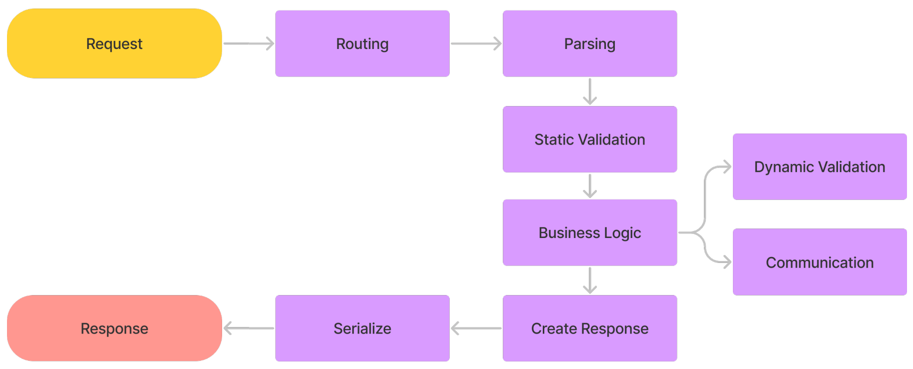


Figure 2.1: Flow of a web API implementation

- Error messages. Errors returned by the web API when an input is malformed, some user error occurs or some server error occurs.

When creating an implementation you want to know as much as possible beforehand. This way you run into less errors when an integration is used in production. To prevent errors in an integration, it should be possible to check if an implementation is actually correct[2].

In Figure 2.1 a simplified overview of the different steps in a web API is given. The request is received by a web server, which passes the request to the application code, the application code executes some steps and the server is used to send a response back to the service that send the initial request.

What happens between the request and response differs between implementations, the diagram given gives a high level overview of the process. Usually some parts of the implementation are automated and handled by a framework. Routing, parsing and serializing are usually handled by a framework. This means that the main API implementation can be expressed as a normal function in the programming language.

Routing Routing is usually handled by a framework, the router uses the URL of the request and maps this to a specific function in the implementation. The router can usually be created in a declarative way.

Parsing The parser is also usually part of a framework. An incoming request can send some data to the server, the parsers is responsible for converting the data from a string into a structured format which can be easily used in the programming language that is used.

Serialization The serializer takes an object in the programming language and converts it to a string which can be used in the response. The main part of the implementation consists of different blocks, which can be roughly separated in three parts. Validating the request, business logic and creating a response.

Validation In the validation part of the implementation it is made sure that an incoming request has the correct structure and that the values in the request are valid. This step can reject a request if there is any error in the request. What is validated should be clearly documented, such that applications communicating with the API can make sure that no unexpected errors occur. Some parts of the validation are based only on the data in the request, we call this static validation, for these rules it is possible to know before sending the request if the request will give an error. And sending the request again will always result in an error. Dynamic validation happens with rules that are based on some external information, like checking user rights or deducting a balance. This validation is based on some external state, which is not generally available to the sender of the request.

Business Logic The main business logic of the implementation is responsible for processing the incoming data and obtaining data which is returned. What happens in this step is usually hidden away from the end user.

Create Response Using the data obtained in the business logic a response is created. It should be clearly documented what a response from a specific endpoint looks like.

We want to be able to communicate how incoming requests are validated to developers who are interacting with our APIs. As mentioned previously this communication should happen in a formal way, so that other tools can reuse the specification we have created. Being able to communicate exactly what a valid request looks like eliminates any errors related to this.

In the the following section we describe an existing way of communicating the structure and behaviour of a web API in a formal way.

2.2 Web API Specifications

In the previous section we mentioned creating a specification for a web API. We will look into OpenAPI Specification (OAS)[1] for this project as it is the most widely used and previous research also focuses on OpenAPI. An OpenAPI document contains all information that describes the API endpoints. The specification is defined in either JSON or YAML format.

The OpenAPI Specification is created by the OpenAPI Initiative, which is an open source community which consists of developers from the industry and has the goal to create vendor-neutral, portable and open specifications for providing technical metadata for APIs. The OpenAPI specification is the main focus of the initiative.

OpenAPI documents consist of different parts, a specification at least contains some information about the API, like a title and a version, and it also contains a list of endpoints. Each endpoint contains a definition of how a valid request is structured. In Listing 2.1 an example of a minimal specification is shown, this example defines a single Schema `Pet` and an endpoint `POST /pet` which accepts a `Pet` and returns a `Pet`.

[1]`https://spec.openapis.org/oas/v3.1.0`

Specifications can also contain information about how to authenticate, access control and which URLs can be used to connect to the server. In Section 3.1 we will zoom in on how valid request are defined, as most other parts of the specification do not contain any logic.

2.2.1 Use cases for web API specifications

There are different use cases for API specifications. For a single web api implementation one or more of these use cases can be used.

Documentation The most common use case for OAS is documentation. The specification is used to communicate what the implementation of the web API is and how it can be used. For example Swagger UI[2] and ReDoc[3] use OAS files to generate a documentation website. This documentation can be used by users of the API to explore the available endpoints, see examples and also interact with the API.

Server generation It is possible to generate parts of a server implementation using an OpenAPI specification. After updating the specification an updated implementation can be generated. At large scale the size of the code generator implementation is often smaller than the code that is generated. Requiring less code to be maintained, there already exist code generators for multiple different frameworks and programming languages. With this approach you can be sure that, if there are no bugs in the generator, the generated implementation conforms to the specification. With a generated implementation the actual business logic still needs to be implemented.

SDK or Client generation Similarly to generating a server implementation, a client implementation can be generated using an OAS file. Generators exist for different languages[4], making it possible to offer SDKs or clients for different languages and opening up the possibility to quickly create a client in a new language.

Test generation Contract testing can be done by using an OpenAPI specification to generate test cases which test if an implementation conforms to the specification[7]. Using the specification it is possible to generate test cases which are close to valid requests.

Configuration Other applications which interact with web APIs, like API gateways[5], automatic security scanners[6] and standalone API clients[7] can be configured using OpenAPI Speci-

[2]`https://swagger.io/tools/swagger-ui/`

[3]`https://github.com/Redocly/redoc`

[4]`https://swagger.io/tools/swagger-codegen/`

[5]`https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-import-api-update.html`

[6]`https://docs.stackhawk.com/hawkscan/configuration/openapi-configuration.html`

[7]`https://learning.postman.com/docs/integrations/available-integrations/working-with-openAPI/`

```yaml
openapi: 3.0.3
info:
  title: Sample Pet Store App
  version: 1.0.0
paths:
  /pet:
    post:
      description: Add a new pet to the store
      responses:
        '200':
          description: Successful operation
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Pet'
        '405':
          description: Invalid input
      requestBody:
        description: Create a new pet in the store
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Pet'
components:
  schemas:
    Pet:
      required:
        - name
      properties:
        id:
          type: integer
          format: int64
        name:
          type: string
          minLength: 2
      type: object
```

Listing 2.1: Example OpenAPI specificaiton

fication files. Making it possible to have one specification which can be used for many different tools.

2.3 Inter Parameter Constraints

In a complex OpenAPI document there can be additional constraints on what is considered a valid input. When it is important that users of the API know what constraints there are the current way of documenting this is by adding a note to a field or path in the OpenAPI document. There is previous research which categorises the different ways kinds of constraints that are generally found in web APIs and creates a domain specific language to express these constraints and have a way of automatically checking the constraints.

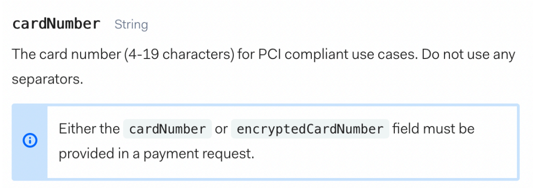
Inter Parameter Constraints are constraints between different fields in a web API. These constraints are static, that means they are known beforehand and do not change based on some external state of the application.

This means that it is possible to document them in a specification. Having these constraints formally documented makes it possible to automate the validating these constraints, for example in an SDK which is automatically generated from the OpenAPI specification.

There are different classes of inter parameter constraints that can exist in web APIs, in this section we give an overview of the inter parameter constraints that are usually found in web APIs[20].

- Value Constraints: Constraints on a specific value, examples of this are:
  - A number must be non-negative
  - A string must look like an email address
  - A string must have a specific length
- Group constraints: Given a set of parameters exactly none, exactly one, at least one or all should be available.
- Dependent constraints: Constraints on a parameter depend on a property of another parameter.
  - Present-Present (PP) dependent constraint: the presence of a parameter depends on the presence of the base parameter
  - Present-Value (PV) dependent constraint: the presence of a parameter depends on the value of the base parameter
  - Value-Value (VV) dependent constraint: the accepted set of values for a parameter depends on the value of the base parameter.

In some cases these constraints are already documented as we have shown in Chapter 1, but they can also exist as part of existing business logic. There are different scenarios where a constraint is implicitly defined.

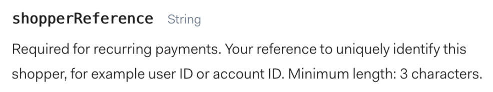


cardNumber String

The card number (4-19 characters) for PCI compliant use cases. Do not use any separators.

Either the cardNumber or encryptedCardNumber field must be provided in a payment request.

Figure 2.2: A group constraint in the Adyen API



shopperReference String

Required for recurring payments. Your reference to uniquely identify this shopper, for example user ID or account ID. Minimum length: 3 characters.

Figure 2.3: A Present-Present dependent constraint in the Adyen API

An example of a group constraint is shown in Figure 2.2. In this constraint the request is valid if either `cardNumber` or `encryptedCardNumber` is set.

In Figure 2.3 an example is given of a field that is required under a specific condition, this field is required when the `recurring` field contains a value. This is a case of a Present-Present dependent constraint.

There are two possible results of sending requests which do not conform to the inter parameter constraints present in that request.

The API returns an error The API can return an error with an error message when some constraint is not met. This error message informs the end user about missing values under certain conditions.

The API makes a silent choice The API can also make a choice depending on the input, it can use a default value or choose to ignore a field if another field is not present.

2.4 Inter Parameter Constraints in Web API implementation

In this section we go over how inter parameter constraints are present in web API implementations. To gain a better understanding of what inter parameter constraints are.

2.4.1 Implementing Inter Parameter Constraints

Most web API implementations use some imperative logic to validate Inter Parameter Constraints. This is caused by the fact that most Inter Parameter Constraints are implicitly defined in the business logic. Another reason is that programming languages often do not support describing these inter parameter constraints in a declarative way. In Listing 2.2 we show an example is shown of validating the `IF country=='US' THEN state` expression. There are multiple ways these constraints can be implemented and in existing implementations the validation code is usually already there and the inter parameter constraints are still implicit.

```java
public void validateAddress(Address address) {
  if (address.getCountry().equals("US")) {
    if (address.getState() == null) {
      throw new ValidationException("State is required for
       ↪  country \"US\".");
    }
  }
}
```

Listing 2.2: Example of validating a constraint in Java

```java
public void validateSum(Payment payment) {
  int sum = 0;
  for (Split split : payment.getSplits()) {
    sum += split.getValue();
  }

  if (sum != payment.getValue()) {
    throw new ValidationException("Sum of splits is not equal
     ↪  to payment value");
  }
}
```

Listing 2.3: Example of validating a constraint in Java

In Listing 2.3 we give an example of a constraint which is a bit more complex and requires a loop to check if the constraint is satisfied. This example cannot be expressed using domain specific languages in previous research.

In Appendix B we compare what implementing inter parameter constraints in different contexts is like. This gives us insight in which parts of the implementations are impacted by inter parameter constraints.

2.4.2 Detecting Inter Parameter Constraints

Most web API implementations contains some inter parameter constraints. Sometimes these constraints are hidden away in the implementation and might not be very obvious. To find these constraints in web APIs multiple techniques can be used.

One way of finding inter parameter constraints is by extracting constraints from the web API documentation[28], this looks at documentation of a field that refers to another field. This approach can find constraints which are already known, but have no formal definition.

Another way is by analysing the source code of a web API implementation, this approach can help find inter parameter constraints that are not yet explicitly known. Grent et al. [10] combines these techniques to find inter parameter constraints in the Adyen APIs.

```
type: object
required:
- name
properties:
  name:
    type: string
    pattern: [A-Z][a-z]+
  address:
    $ref: '#/components/schemas/Address'
  age:
    type: integer
    format: int32
    minimum: 0
```

Listing 2.4: Constraints on primitives in OpenAPI definitions

```
MyResponseType:
  oneOf:
  - $ref: '#/components/schemas/Cat'
  - $ref: '#/components/schemas/Dog'
  - $ref: '#/components/schemas/Lizard'
  discriminator:
    propertyName: petType
```

Listing 2.5: Constraints on multiple fields

2.4.3 Expressiveness of OpenAPI

In this section we will explore the expressiveness of OpenAPI Documents. There is a certain precision with which you can define web APIs using an OpenAPI document. There are limitations to what is possible to express using an OpenAPI Document. We will focus on the schema definition part in OpenAPI to see what can be expressed using OpenAPI documents and what is not possible to express.

The OpenAPI specification uses an extended subset of JSON Schema[8] to impose constraints on values. With these constraints it is possible to define constraints on primitives, such as strings and numbers. Listing 2.4 shows how constraints like `required`, `pattern` and `minimum` are used in an OpenAPI specification.

The JSON Schema part of OpenAPI has more possibilities to describe constraints on the accepted input of a request. Using a `discriminator` it is possible to do polymorphism and make specific fields available based on the value of a specific field. It is also possible to use composition to create specific combinations of values that are accepted.

There are limitations to what is possible to express using OpenAPI. Besides the

[8]See `https://json-schema.org/`

```
x-dependencies:
  - ZeroOrOne(radius, rankby=='distance');
  - IF rankby=='distance' THEN keyword OR name OR type;
  - maxprice >= minprice;
```

Listing 2.6: Example constraints in IDL4OAS

```
x-constraints:
  - and(implic(value(rankby) = 'distance',
   ↪  not(present(radius)), implic(present(radius),
   ↪  value(rankby) != 'distance'))
  - implic(value(rankby) = 'distance', or(keyword, name,
   ↪  type))
  - value(maxprice) >= value(minprice)
```

Listing 2.7: Example constraints in OAS-IP

`discriminator` it is not possible to have the value of a field impose constraints on other fields. Making a field required when some other field is required can be encoded using a discriminator field, but without this it is not possible. There is also no way of describing constraints based on arithmetic logic and have mutual exclusive fields.

2.4.4 Expressiveness of OpenAPI with extensions

There are APIs that can be expressed using an OpenAPI specification, but there are also APIs which cannot be formally described by OpenAPI. When it is not possible to formally describe an API using an OpenAPI specification, the author of the specification has to resort to leaving out details or using the description blocks to explain informally what additional requirements the API has.

It is possible to define extensions for OpenAPI. Examples of extensions for OpenAPI are extensions which add fields to the documentation, other extensions configure the generation of code from the specification, for example defining the namespace for the generated code or adding extra validation steps.

One OpenAPI extension which adds expressiveness of request object definitions to the specification is IDL4OAS (Inter-parameter Dependency Language for Open API Specification). This extension adds the possibility to add additional constraints to object defined in OpenAPI specifications. Using this extension it is possible to create specifications with more detail. The fields defined using this extension can be used to generate an implementation with extra checks, or to validate a given request object. In Listing 2.6 we give an example of some constraints in IDL4OAS.

Another OpenAPI extension is Oostvogels et al. [20], it proposes an extension to the OpenAPI Specification by adding an extra field for an endpoint which contains dependent constraints (Listing 2.7).

Both solutions can express a similar set of constraints. The constraints expressible by these domain specific languages is limited, but they can express most constraints found in web APIs in the wild. In Chapter 5 we show how dependent types can be used to describe any inter parameter constraints found in web APIs and we compare our solution with these existing solutions.

Chapter 3

Types

In this chapter we create a formal semantics of OpenAPI documents, these formal semantics describe what the meaning of an OpenAPI document is. We use this semantics to show that in the basis the OpenAPI specification describes a type in a programming language. With this point of view we will look into what is required to describe OpenAPI documents with inter parameter constraints as a type in a programming language.

3.1 Comparing OpenAPI specifications to a type system

A web API specification is a contract telling users of an API how an API works and what values are accepted. This is similar to types in programming languages. Depending on the expressiveness of the type system in the programming language a specific set of types can be described. There are programming languages with a dynamic type system where type-safety is traded for ease of development. In other programming languages the type system is stricter, making it possible to encode more information about the program before the code is compiled.

There is a limit on how precise an API can be described using an OpenAPI document. As most fields in the specification are optional a valid OpenAPI document can be incomplete. For the OpenAPI documents we are looking at we assume that the creator of the document wants to be as complete as possible in their description of the web API. When this is the goal there are limitations to what can be described. This is the reason for the previous research on inter parameter constraints.

There are ways to describe an OpenAPI document as a type in a programming language, for example using OpenAPI.NET[1]. In this case the OpenAPI document is a value in the program and can be interacted with and information can be extracted about the document. The solution we propose here goes further than this, in our solution the OpenAPI document itself is the type in the program and the values are the input and output of the web API.

[1]`https://github.com/Microsoft/OpenAPI.NET`

3.2 Formal OpenAPI Semantics

The semantics of OpenAPI definitions are described in the OpenAPI specification. This document outlines what a valid OpenAPI definition looks like and what it's semantics are. In this chapter we will express these semantics in a formal way. For this definition we limit the scope to the part of the specification that defines schemas, here it is possible to describe semantics of validating some JSON in the context of a specific request or response. In Figure 3.1 the semantics of the schema part of an OpenAPI definition is defined. Only the schema and path definitions of the OpenAPI specification are described here as they are used to validate requests. Other parts are omitted from this definition as they are purely declaring metadata about the API.

We use the following methods to define interactions with JSON objects, we use JSON objects as defined in the ECMA 404:2013 [6] standard.

typeof j**:** Obtains the type of a JSON value, this is **arr**, **obj** or one of t.

flds(j)**:** Obtains the set of fields defined in a JSON value, only defined when **typeof** $\equiv$ **obj**.

$j \cdot x$**:** reads the field x from a JSON value j, is only defined when **typeof** $j \equiv$ **obj** and $x \in$ **flds**(j)

elems j**:** Obtains a set of items in j, only defined when **typeof** $j \equiv$ **arr**.

The formal semantics of an OpenAPI document starts with the root of the document, called **api** in this definition. An API contains a collection of Σ, which is a lookup map of schema objects by name. and a collection of **paths**, which defines an accepted input and output. Each input and output is accompanied by a schema which defines what the accepted values are.

The schema definition uses a simplified version of the JSON schema specification. A JSON value is either a list, an object or a value (Equation 3.5). An object (**oschema**) contains a list of names combined with schema definitions and a list of required names. An array (**aschema**) contains a single type that every value of the list should conform to. A value (**tschema**) has a single type (Equation 3.2), either a string or a number.

This definition defines a subset of the OpenAPI specification. Any additional metadata, which is generally only used for documentation is left out. For the sake of simplicity only one method for each endpoint is defined, in an actual OpenAPI document it is possible to define behaviour for multiple different methods and content types. But these definitions follow the same rules. How the input is defined is also simplified, in the OpenAPI specification there are parameters and the request object, in this definition there is only the request object. The request object modelled in our definition is powerful enough to also capture the parameters defined in the OpenAPI specification.

With these formal semantics we reason about the semantics of OpenAPI and we can use these semantics to write a type in a programming language. Using this type we can create a program that conforms to this type and the if there are any mismatches between the implementation and the specification, the type checker of the programming language will raise an

$$
\begin{array}{rcll}
x & \in & \{x,...\} & (3.1)\\
t & ::= & \textbf{str} \mid \textbf{number} & (3.2)\\
\Sigma & ::= & (x,r)^* & (3.3)\\
r & ::= & \textbf{ref}\ x \mid s & (3.4)\\
s & ::= & \textbf{oschema}\ (x,r)^*\ x^* \mid \textbf{aschema}\ r \mid \textbf{tschema}\ t & (3.5)\\
o & ::= & \textbf{api}\ \Sigma\ (p,(r,r))^* & (3.6)\\
 & & \vdash_o p\ j\ j : o & (3.7)\\
 & & \Sigma \vdash j : r & (3.8)
\end{array}
$$

$$
\frac{\begin{array}{c}\textbf{typeof}\ j \equiv \textbf{obj}\\ (\forall x \in \textbf{reqs}\ \cdot x \in \textbf{flds}(j))\\ (\forall (x,r) \in \textbf{props} \cdot x \in \textbf{flds}(j) \rightarrow \Sigma \vdash j \cdot x : r)\end{array}}{\Sigma \vdash j : \textbf{oschema props reqs}}
$$

$$
\frac{\begin{array}{c}\textbf{typeof}\ j \equiv \textbf{arr}\\ \forall j' \in \textbf{elems}\ j\\ \Sigma \vdash j' : r\end{array}}{\Sigma \vdash j : \textbf{aschema}\ r}
$$

$$
\frac{\textbf{typeof}\ j \equiv t}{\Sigma \vdash j : \textbf{tschema}\ t}
$$

$$
\frac{\Sigma \vdash j : \Sigma(x)}{\Sigma \vdash j : \textbf{ref}\ x}
$$

$$
\frac{\begin{array}{c}(p,(r_i,r_o)) \in \textbf{paths}\\ \Sigma \vdash j_i : r_i\\ \Sigma \vdash j_o : r_o\end{array}}{\vdash_o p\ j_i\ j_o : \textbf{api}\ \Sigma\ \textbf{paths}}
$$

Figure 3.1: Semantics of OpenAPI definitions

error during compile time. In the next chapter we will take the definition from this chapter and implement it in the Agda programming language.

Chapter 4

Formalizing OpenAPI Specifications in Agda

In this chapter we implement the formal semantics of an API in the dependently typed programming language Agda. This creates a program in which an OpenAPI document can be defined as a type in the programming language and the language will be able to validate if a specific input or output is considered valid. In the next chapter we will add inter parameter constraints to this implementation.

We first introduce Agda and the advantages Agda has when implementing a system like this one. We then show how we can encode OpenAPI documents in Agda to leverage the Agda typesystem.

Dependent types can be used to describe inter parameter constraints in web APIs. In this chapter we go over what dependent types are and explore using dependent types to express inter parameter constraints in the dependently typed Agda programming language.

In Section 3.1 we have seen that OpenAPI Specifications and OpenAPI Specifications have a specific expressibility, as schemas only contain information about which fields are defined and cannot convey any constraints across multiple fields. For most use cases this expressibility is sufficient, but in some cases it is desirable to be able to express more fine-grained constraints.

In this chapter we show an API implementation in a dependently typed programming language. This implementation is able to express all constraints previously discussed and should be able to express any constraint[15].

In Appendix C you can find all the Agda code written for this thesis.

4.1 Type Theory and Dependent Types

Type theory is a way of reasoning about the validity of programs. In type theory the type system is used to describe what a valid program looks like. In a dynamic language like JavaScript or Lisp all syntactically valid programs can be executed. A function can have arguments, but there is no way of knowing what the type of that argument will be. It is possible that during runtime an exception will be thrown if a wrong assumption about the types is made in the program. This makes it easy to write programs, but also easy to write malfunctioning programs.

Java or Haskell are typed languages, this means that in a function definition we can limit what types are accepted and be sure that the compiler will not accept invalid calls to that specific function. This way a programmer has more knowledge about external factors, such as third party libraries, while developing a program and has to rely less on the documentation of those external factors.

All these languages are considered partial languages. This means that any expression of type `T` is one of the following:

- the program terminates and returns a value in type `T`
- the program does not terminate
- the program throws an exception (caused by an incomplete function definition)

Programming languages like Agda and other languages based on type theory are total languages in this sense. Any expression of type `T` will always terminate and return a value in type `T`. Runtime errors are impossible and non-terminating programs can only be written if explicitly stated.

4.2 Dependent Types in Agda

Agda is a dependently typed programming language developed by Norell [18]. This programming language uses a dependently typed type system. Dependent types is based on type theory by Martin-Löf [13]. With dependent types it is possible to create a type indexed by another type or values of another type. In dependent type theory a Π type is used to define a dependent pair, where the second part of the pair depends on the type of the first part of the pair. This allows us to encode very much in the type of an object. In the next section we will look at an implementation of dependent types.

One example to show the power of dependent types is a `head` function for a list. If we have a normal generic list definition in Agda, which is defined as a recursive data type in Agda.

```
data List (A : Set) : Set where
  []  : List A
  _::_ : (x : A) (xs : List A) → List A
```

We can create a head function for a list of this type.

```
head : ∀ {A : Set} → List A → Maybe A
head [] = nothing
head (x :: x₁) = just x
```

This head type takes a list as it's argument and returns an optional value. The optional value is empty when the list has no values. There is no way to be certain that the list has at least one value, and programs in Agda are not allowed to return an error. It is of course possible to create a non-empty list type which does not have a constructor for an empty list. But this only solves the problem for this specific case, lists with a maximum length are much harder to implement this way.

When using dependent types it is possible to create types which make use of values. For example a list type which also contains it's own length. The `Vec` data type defined below is dependent on a natural number which denotes it's length, it starts at zero for the empty vector and increases the length for every next item in the vector. Having this number inside the type opens up the possibility to constrain the value of this number. In this example by requiring the number to be `1 + n`, making it impossible to construct the type with length 0 because n is a natural number. In this place it is also possible to impose more complex constraints on the dependent value, like requiring it to be between two numbers or requiring it to be odd.

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)
```

We can now define a head function which will always return a value for a given input. We can do this by requiring the length of the vector to be `1 + n`, where `n` is an arbitrary natural number. This expression

```
head : ∀ {n A} → Vec A (1 + n) → A
head (x :: xs) = x
```

Using values in types is where the power of dependent types lies. With this it is possible to create really fine grained types. This does come with added complexity, because you need to construct these types. Agda can help developers to construct these types, when there is a straightforward way to create a type Agda can be instructed to generate the code for that type.

4.3 Agda Standard library

The Agda programming language in itself has quite a small footprint, to account for this there is a separate standard library. This standard library helps with quickly developing new Agda programs, because most frequently used proofs are defined in this standard library.

4.4 JSON in Agda

The input and output for a web API is usually JSON. This format has no formal structure so no assumptions can be made of the types of values in a JSON tree. A value in JSON can either `null`, `string`, `float` or `bool`, a value can also be a list of JSON values or another object containing JSON values. An object is defined as a list of values indexed by strings. The following data type defines the basic structure of a JSON object in Agda.

```
data JSON : Set where
  null : JSON
  string : String → JSON
  float : Float → JSON
  bool : Bool → JSON
  array : List (JSON) → JSON
  object : List (String × JSON) → JSON
  number : ℕ → JSON
```

This definition has an explicit difference between real numbers and integers, because Agda makes this distinction, JSON itself there is only a number type which contains both kinds of numbers. This specific representation of JSON is chosen because of it's simplicity. It does allow objects with duplicate keys, which is not allowed by JSON. Making sure that keys in objects are unique is possible with Agda, but it would require extra complexity in the code.

Given the following JSON object.

```
{
  "a": 0.1,
  "b": {
    "d": "Foo"
  },
  "c": null
}
```

Would look like the following when creating an object for it in Agda.

```
myObject : JSON
myObject = object
  ( ("a" , float 0.1)
  :: ("b" , object (("d" , string "Foo") :: []))
  :: ("c" , null)
  :: [] )
```

This definition of a JSON object in Agda can be used to model the input and output of a web API. In the implementation it is assumed that there exists some implementation that takes a string as input and returns a JSON object. We can also convert JSON to an object inside the Using the Maybe monad a JSON object can be converted to an object in Agda an example of this is shown in Listing 4.1, here an object is returned if the JSON is valid.

4.5 Implementing OpenAPI documents in Agda

To map the definition of an OpenAPI document to Agda we start with the root of the document which contains a list of paths, which is a string for the path combined with a schema defining what is valid input and output JSON.

```
jsonToCategory : JSON → Maybe Category
jsonToCategory json = do
  id ← json • "id"
  nameString ← json • "name"
  name ← parseString nameString
  just (category id name)
```

Listing 4.1: Converting JSON to Agda

```
data OpenAPI : Set where
  openapi : List (String × (Schema × Schema)) → OpenAPI
```

We define a schema as follows, just like in the formal semantics there are three different schema types. Each schema type has different parameters.

```
data Schema : Set where
  schema : Type → Schema
  aschema : Schema → Schema
  oschema : (flds : List (String × Schema)) → (reqs : List
   ↪  String) → Schema
```

We now define what a well typed schema looks like, for this we create a data type which is dependent on a `Schema` an `JSON` value.

```
data ⊢s : Schema → JSON → Set where
```

We define what it means for a JSON value to have a specific type.

```
  wt : ∀ { type js } →
    JsonHasType js type →
    ---------------
    ⊢s (schema type) js
```

When we expect a JSON array, we check the JSON value is actually an array and that every value in that array is in itself again a well typed value in the schema given in the definition of the array schema.

```
  wta : ∀ { s js } →
    (∃ λ ls → js ≡ (JSON.array ls) × All (⊢s s) ls) →
    ---------------
    ⊢s (aschema s) js
```

When we expect a JSON object in this value, we check if the JSON value is an object, if all required fields exist and have a value in the object and if the fields in the schema definition are also found in the given JSON.

```
  wto : ∀ { reqs js s2 } →
    (∃ λ fields →
      js ≡ (JSON.object fields) ×
      HasRequireds fields reqs ×
      All (λ (f , js) → ∃ λ (o) → (o ∈ s2) × (f ≡ proj₁ o) ×
       ↪  (⊢s (proj₂ o) js)) fields
    ) →
    ----------------
    ⊢s (oschema s2 reqs) js
```

A well typed route is defined by `⊢t`, it can be constructed using an OpenAPI document, a string for the path and two JSON objects for the request and the response. If the path is in the OpenAPI document and both the JSON objects are well typed schemas for this path the route is well typed.

```
data ⊢t : OpenAPI → String → JSON → JSON → Set where
  wt : ∀ { κ πs p ji jo σ₁ σ₂ } →
     ((p , (σ₁ , σ₂)) ∈ πs) → (⊢s σ₁ ji) → (⊢s σ₂ jo) →
    --------------
    ⊢t (openapi κ πs) p ji jo
```

We can now create a function which will return if a specific JSON value is conforming to a specific schema. `solveSchema` is the signature for this function, given a schema and JSON value it will return `nothing` or a well typed schema object for these values, this object can be used to extract information about the JSON. The implementation of this function is can be found in Appendix C.

```
solveSchema : (s : Schema) → (json : JSON) → Maybe (⊢s s json)
```

4.6 API Implementation in Agda

This section describes an overview of the API implementation written in Agda.

For the API implementation itself we made some assumptions about what is already there and what is interesting to implement for this proof of concept. In this implementation an endpoint is a function which accepts a request containing JSON and a context as parameters and returns a response containing JSON and a new context. The context contains anything that should be persisted, in this specific case the context is a list of `Pet` instances.

Handling HTTP requests is not implemented in this version, as we did not consider this relevant at this stage.

Below we use an OpenAPI definition combined with the name of an endpoint and some incoming JSON. We can only construct this type if the endpoint is found in the list of endpoints defined in the OpenAPI definition and if the schema matches the JSON.

```
data ⊢tin : OpenAPI → String → JSON → Set where
  wtin :  ∀ { κ πs p ji σ₁ σ₂ } →
```

```
   ((p , (σ₁ , σ₂)) ∈ πs) → (⊢s σ₁ ji) →
  --------------
  ⊢tin (openapi κ πs) p ji
```

A function can now require this ⊢tin type as a parameter, this allows the implementation of the function to re-use the assertions made within the specification. Below we give an example of a basic function that has a properly formed request as input and returns a json object that is part of a valid response from the same endpoint.

```
api : ∀ {json p} → (wf : ⊢tin openApi p json) → ∃ λ jsout →
 ↪  (⊢tout openApi p jsout)
api (wtin (here refl) (wto (hrs []) x₁)) = JSON.number 1 ,
 ↪  wtout (here refl) (wt (jht refl))
```

Chapter 5

Formalizing OpenApi Specifications with Inter Parameter Constraints in Agda

In the previous chapter we implemented the OpenAPI specification in Agda, in this chapter we will expand this implementation by adding support for arbitrary inter parameter constraints. With this we can express inter parameter constraints in web API using dependent types.

5.1 Inter parameter constraints in Agda

Constraints in web API specifications are about describing constraints which are part of specific objects, in traditional languages validating these constraints means writing validation logic. In a programming language with dependent types it is possible to define objects with fields that have a type which depends on the value of some other field. This allows embedding constraints in a type, making it impossible to create instances of that type which do not satisfy the constraints. We give an example of a program which defines a `data` type, which is dependent on two values `f1` and `f2`. The constructor `ex` is the only constructor for this datatype, this constructor requires us to show that `f1 < f2`.

```
data Example : ℕ → ℕ → Set where
  ex : ∀ {f1 f2} →
    f1 < f2 →
    Example f1 f2
```

We can create an instance of this data type when we know the values for `f1` and `f2` and we know that these values are valid. For example for `f1 = 3` and `f2 = 4`, the value `3<4` contains a proof of this requirement. Agda can help us here by generating parts of the proof.

```
3<4 : 3 < 4
3<4 = s≤s (s≤s (s≤s (s≤s z≤n)))
```

Decidable	Optional
true + proof	true + proof
false + proof	false

Table 5.1: Comparing decidable to optional proofs.

Using this proof we can create an instance of `Example`, the example value now contains a proof that both the parameters of the instance conform to the constraints defined in the data type.

```
example1 : Example 3 4
example1 = ex 3<4
```

We can also make a function which will return an instance of `Example` when it is possible or an empty value when it is not possible. This allows us to ask this function to create an instance for us when we don't know the exact values of the parameters.

```
-- Create from arbitrary natural numbers
createExample? : (f1 : ℕ) → (f2 : ℕ) → Dec (Example f1 f2)
createExample? f1 f2 with f1 <? f2
... | yes w = yes (ex w)
... | no ¬w = no λ {(ex x) → ¬w x}
```

Here we use Agda's decidable logic[1] to create a function that decides if there is an instance of the type based on some input. A decidable statement can be proven true or false. In Agda an instance of a decidable statement consists of two parts: A boolean value whether the statement is true or false. The boolean value is accompanied by a proof, proving the statement true or false respectively. We can use the resulting proof later in the program to extract the relation between the two fields again.

In Table 5.1 we compare the differences between Decidable and Optional proofs. Decidable statements always contain a proof, whether the statement is true or false. Optional statements only contain a proof when the statement is true, this means that it is possible for a statement to be true, but for the function to return false. In further parts we will use the Optional based logic, because it allows for faster development.

We use this approach to create more complex constraints, to support the inter parameter constraints described in [14].

The constraints which are expressible in this way surpass what is possible to express in OAS and in IDL4OAS, as described in Section 3.1. For example when we have a value containing a list and a value which is the sum of the values in the list.

```
data Amount : Set where
  amount : (value : ℕ) → Amount
```

[1] `https://plfa.github.io/Decidable/`

$$\mathbf{dep} ::= \mathbf{JSON} \to Set \tag{5.1}$$
$$s ::= \mathbf{oschema}\ (x,r)^*\ x^*\ \mathbf{dep}^* \mid \ldots \tag{5.2}$$

Figure 5.1: New rules for formal semantics

The request has two fields, a list of `Amount` and a natural number. There is a constraint which requires the sum of the amounts to be the same as the natural numbers.

```
data Request : (List Amount) → ℕ → Set where
  req : ∀ {amounts total} →
    total ≡ (sum (map (λ {(amount value) → value}) amounts)) →
     Request amounts total
```

When creating an instance of `Request` we now need to provide a proof that the sum of amounts is the same as the given natural number. When the values are know this proof is trivial.

```
amounts : List Amount
amounts = (amount 13) :: (amount 12) :: (amount 8) :: []

val : Request amounts 33
val = req refl
```

5.2 Extending the Formal Semantics

We extend the formal semantics defined in Chapter 3 to support adding an arbitrary dependent type to the definition of an object in the formal semantics. We extend the previously defined formal semantics with the rules defined in Figure 5.1. The rule in Equation 5.2 replaces the **oschema** case in Equation 3.5.

The **JSON** $\to$ *Set* type in Equation 5.1 defines a class of dependent types which are dependent on some JSON value and exists if some the JSON value can satisfy the constraints in the constructor of the type.

We do not create an additional dependently typed domain specific language here, because there already exist dependently typed languages which provide us with the ability to express the constraints we want.

5.3 Describing inter parameter constraints in the dependently typed implementation

We can add inter parameter constraints to our implementation by adding an additional field to the definition of a schema describing a JSON object. We define a `Dep` datatype which contains a constraint on JSON, and a function that returns for a given JSON value if it is conforming to the defined constraint.

```
data Dep : Set where
  dep : (D : (JSON → Set)) → (∀ (json : JSON) → Maybe (D
   ↪  json)) → Dep
```

In this thesis we are only interested in the positive case, where the JSON actually satisfies the constraints. This allows us to focus only on the valid cases and postpone complex issues that might potentially arise when also looking at proving the invalid cases. In a complete version we can use decidable logic, also discussed in Section 5.1, then the negative case is also proven. This can be used to find out why a specific JSON object was not considered valid. With decidable logic you can also be sure that the validation is complete, because it is required to either prove that a value either satisfies the requirements or prove that the value does not satisfy the requirements. This means that an implementation with decidable logic will be guaranteed to cover all cases, the weaker optional logic we use here it is possible that a valid case is dismissed. This is a trade-off we make to simplify the implementation.

We place this `Dep` datatype inside the schema definition, and when creating a new schema we can now also add this validation object to the type. This forces the implementation to verify if the constraints are satisfied when validating a schema. We make the following change to the `Schema` data type defined in the previous chapter.

```
  wto : ∀ { dep reqs s2 fields } →
    HasRequireds fields reqs →
    All (λ (f , json) → ∃ λ (o) → (o ∈ s2) × (f ≡ proj₁ o) ×
     ↪  (⊢s (proj₂ o) json)) fields →
    ValidDep dep (JSON.object fields) →
    ---------------
    ⊢s (oschema s2 reqs dep) (JSON.object fields)
```

The following constraint requires a JSON object which either has a field age, which must be defined and contain a number when the field name is ”Foo”. We begin by defining a data type which is dependent on some JSON value.

```
data ValidJs : JSON → Set where
```

The first constructor for this data type requires the JSON value to be an object that contains both a name and age field, it requires the name field to contain the string `"Foo"` and the age field to by of type `number`.

```
  validNameHasVal : ∀ {flds json name age jsage} →
    json ≡ JSON.object flds →
    ("name" , name) ∈ flds →
    ("age" , age) ∈ flds →
    age ≡ (JSON.number jsage) →
    name ≡ (JSON.string "Foo") →
    ValidJs json
```

The second constructor requires the JSON value to also be an object, but requires the name field to be not equal to "Foo".

```
  validNameHasNoVal : ∀ {flds json name} →
    json ≡ JSON.object flds →
    ("name" , name) ∈ flds →
    name ≢ (JSON.string "Foo") →
    ValidJs json
```

We can define and implement a decision method which tells returns an instance of the constraint datatype if it is possible. This function is used by the implementation to create instances of this specific constraint.

```
validJsCheck : (json : JSON) → Maybe (ValidJs json)
validJsCheck json = do
  (flds , flds☑) ← jsonObj? json
  ((_ , flap) , flap☑ , refl) ← containsField? "name" flds
  (flapStr , refl) ← jsonStr? flap
  yes refl ← just (flapStr String.≟ "Flap")
        where no ¬a → just (validNameHasNoVal flds☑ flap☑ λ
         ↪  {refl → ¬a refl})
  ((_ , age) , age☑ , refl) ← containsField? "age" flds
  (ageNum , refl) ← jsonNum? age
  just (validNameHasVal flds☑ flap☑ age☑ refl refl)
```

We now define an instance of the Dep type. This instance of Dep contains the previously created constraint and a proof implementation for this constraint that can be used later to validate if a specific JSON value is actually conforming to this constraints. This value returns a proof that the constraint holds or an empty value if it does not hold.

```
deps : Dep
deps = dep ValidJs validJsCheck
```

5.4 Expressing Inter Parameter Constraints in Agda Records

We can also embed inter parameter constraints explicitly in Agda. This approach is shows how inter parameter constraints can be used when programming in a dependently typed programming language.

We begin by creating a new record with some fields; a partial Address record.

```
record Address : Set where
  constructor address
  field
    country : String
    stateOrProvince : Maybe String
```

We add two additional fields to this record, they both require a proof that the given statement is true. These fields can be automatically generated by Agda if the value for the field can be automatically inferred.

```
    {isStateCA} : True (if country ≟ "CA" then (is-present
     ↪  stateOrProvince))
    {isStateUS} : True (if country ≟ "US" then (is-present
     ↪  stateOrProvince))
```

We can create new instances of this record by using known values. In this case we do not need to provide a proof, because Agda can infer it for us.

```
myAddress : Address
myAddress = address "US" (just "New York")

myAddress2 : Address
myAddress2 = address "NL" nothing
```

We can also convert a JSON object to an optional Address, if the JSON has a valid structure this function will return an Address value, in the other case it will return nothing. In this case we do need to provide a proof to construct the record.

```
jsonToAddress : JSON → Maybe Address
jsonToAddress json = do
  country ← json • "country"
  let stateOrProvince = json • "stateOrProvince"
  isStateCA ← decToMaybe (if country ≟ "CA" then (is-present
   ↪  stateOrProvince))
  isStateUS ← decToMaybe (if country ≟ "US" then (is-present
   ↪  stateOrProvince))
  just record
    { country = country
    ; stateOrProvince = stateOrProvince
    ; isStateCA = fromWitness isStateCA
    ; isStateUS = fromWitness isStateUS
    }
```

	IDL4OAS[14]	OAS-IP[20]	Agda
Dependent constraints			
Present-Present	●	●	◐
Present-Value	●	●	◐
Value-Value	●	●	◐
Group constraints			
All or none	●	◐	◐
Exactly one	●	◐	◐
Zero or one	●	◐	◐
Logical constraints			
Or	●	●	●
And	●	●	●
Arithmetic constraints			
$+ \mid - \mid * \mid \div$	●	○	●
Other properties			
Nested values	○	○	●
Constraints on lists	○	○	●

○ = Not expressible, ◐ = Requires encoding, ● = Expressible

Table 5.2: Comparing Inter Parameter Constraint solutions.

5.5 Comparing Agda Inter Parameter Constraints with Previous Work

In Table 5.2 we compare our implementation with IDL4OAS and OAS-IP. We notice that IDL4OAS and OAS-IP can express a very similar set of constraints, but OAS-IP is a smaller language, so more encoding is needed. OAS-IP has the possibility to create functions to encode more complex constraints, but lacks constructs to express arithmetic constraints.

The Agda approach also requires some encoding for some of the constraints. But it is can leverage the existing language to create functions which simplify the construction of constraints. Agda also has ways to represent nested obects and lists, something which is not possible in the other approaches.

The following snippet implements an all or none constraint in Agda for an arbitrary length of values.

```
data AllOrNoneReq : JSON → Set where
  allornone : ∀ {json flds grp} →
    json ≡ JSON.object flds →
    (All (λ name → (name ∈ (List.map proj₁ flds))) grp) ⊎
     ↪  (All (λ name → (name ∉ (List.map proj₁ flds))) grp) →
    AllOrNoneReq json
```

```
allOrNoneCheck : (List String) → (json : JSON) → Maybe
 ↪  (AllOrNoneReq json)
allOrNoneCheck = λ allflds → λ json → do
  (flds , fldscheck) ← jsonObj? json
  just all☑ ← just (all? (λ y → in? y (List.map proj₁ flds))
   ↪  allflds)
    where nothing → do
      none☑ ← all? (λ y → notin? y (List.map proj₁ flds))
       ↪  allflds
      just (allornone fldscheck (inj₂ none☑))
  just (allornone fldscheck (inj₁ all☑))

allOrNoneDep : List String → Dep
allOrNoneDep = λ allflds → (someDep AllOrNoneReq
 ↪  (allOrNoneCheck allflds))
```

In IDL4OAS an all or none constraint is part of the domain specific language and can just be written as $AllOrNone(P_1, ..., P_n)$. In OAS-IP this constraint can be constructed by composing the primitives defined in that language. This language is less flexible when it comes to accepting a dynamic amount of parameters to a constraint.

```
group(f1, f2, f3) := iff(present(f1), iff(present(f2),
 ↪  present(f3)))
```

The other constraints have a similar way of encoding, where the full circles in the table denote that no special extra encoding is needed to express these constraints.

5.6 An API implementation in Agda using OpenAPI

In this section we use the formal definition of OpenAPI to create an API implementation. This implementation will only accept a specific JSON object for an endpoint if the JSON is well formed. We use this to show that we can use Agda to implement this validation and to show what adding dependent types to this validation would add.

Below we show what the signature of an API function would look like. This function takes a well-formed request as input and returns JSON and the proof that this is a well-formed response.

```
api : ∀ {json p} → (wf : ⊢tin openApi p json) → ∃ λ jsonout →
 ↪  (⊢tout openApi p jsonout)
```

We also need to have a mechanism in place that validates requests and checks that an incoming request conforms to a specific specification. Below we show a function definition for a validator.

```
validate : (openApi : OpenAPI) → (p : String) → (json : JSON)
 ↪   → Maybe (⊢tin openApi p json)
```

This function creates an instance of a well formed request. A general implementation of this function can be created.

There is still a lot of manual work to do in this setup. In a complete solution there are constructs that introspect the type of the specification and use this to generate a validator. Having this the only implementation a developer has to do is to execute additional business logic and interact with external services.

This shows that we can express an OpenAPI specification as a type in Agda, and that we can use dependent types to include arbitrary constraints in schema definitions. Having this we can express OpenAPI with dependent constraints as a type, making it possible to share the type of the API, allowing other developers to implement a connection to the API.

Chapter 6

Discussion

This chapter gives an overview of the project's contributions. After this overview, we will reflect on the results and draw some conclusions. Finally, some ideas for future work will be discussed.

6.1 Conclusion

For the first research objective we have shown that an OpenAPI document can be seen as a type in a programming language. We have also shown that this type can be used when implementing a web API. This solution enforces the contract of the web API by using the type system of the programming language, making it impossible to compile programs which are not conforming to the API specification.

Our solution is different from a system where an OpenAPI document is generated from annotations in code or a system where code is generated from an OpenAPI document. The proposed solution uses the type system in Agda to embed an OpenAPI document as a type in the programming language. This type lives alongside the implementation and is used by the compiler to check if the implementation is actually valid.

For the second research objective we have shown that when using a dependently typed programming language this can even be extended to express any decidable constraints in the web API at the level of the API specification. This moves closer to a system where most constraints are present in every layer of the implementation. Leaving as little room as possible for uncertainty about the structure of requests in a web API. With this solution the users of an API can know in advance if the code they have written will pass static validation on the web server.

6.1.1 Steps before reality

We acknowledge that this solution still requires technology that is not yet ready for general use. Dependent types are slowly becoming part of mainstream programming languages, but it will take some time before they can be generally used. Going back from the dependent types in the Agda implementation to an existing OpenAPI document is also another challenge that needs to be tackled.

To allow formally communicating the inter parameter constraints in the dependently typed program there needs to be some domain specific language that can be embedded in an OpenAPI document and that can be read and interpreted by other tools.

To actually use the knowledge provided by the dependently typed inter parameter constraints the language that is used to implement both the API itself and the API client need to have some notion of dependent types to enforce inter parameter constraints on a language level, something which is not generally possible at the moment.

Dependent types are still quite complex to work with and require an extensive knowledge of functional programming language paradigms. With more language adopting ideas from dependent types this problem is expected to become smaller. But describing inter parameter constraints inside a value in a programming language still requires a different way of thinking than is currently the case in traditional programming languages.

6.2 Discussion/Reflection

Using dependent types to create types with embedded inter parameter constraints is feasible, but bringing them to a more general public is still a challenge.

An implementation with dependent types moves the responsibility of showing that inter parameter constraints hold further away. Whereas in for example a Java implementation the constraints are validated inside business logic, in a dependently typed implementation all static constraints can be checked by the deserialization layer. This puts more responsibility on the deserialization layer. We also notice this when implementing web APIs in different programming languages in Appendix B.

Implementing a web API in a dependently typed programming language does require a different mindset when programming. This can be a challenge for programmers who are not used to functional programming languages.

The Agda code can still be improved, it might be possible that certain parts can be written in a more concise way by a more experienced Agda programmer. The system would also need to be expanded to accept decidable logic instead of the current optional logic. This requires at least twice as much work to express the constraints, because there now also needs to be a proof for values that do not pass the constraint. Decidable proofs will make the system more robust as it is then required to prove everything about the values.

We have seen that a type in a dependently typed programming language can express any decidable inter parameter constraint. There are also other domain specific languages which are created to express inter parameter constraints, in this thesis we have shown that using dependent types has a slight advantage in expressibility and can also leverage existing research on the topic. On the other hand, there are also advantages to having a small domain specific language for describing inter parameter constraints.

6.3 Related Work

Inter parameter constraints in Web APIs and dependent types are both well explored domains in computer science. Dependent types has been around since the work of Martin-Löf [13] and is still seeing a lot of research. It is also being incorporated in existing programming languages. Inter parameter constraints is a domain that has been around for a shorter time and is more specific. Research on inter parameter constraints started with trying to detect these constraints [28] to show that these constraints are present in existing web APIs. Later research was on being able to describe these constraints.

In this section we will go over related work.

6.3.1 Validation of Web APIs

There are other solutions to validate if a web API is implemented according to the specification and to check if complex constraints are properly implemented. One way to do this is by using contract testing[11, 8]. This technique uses the specification of a web API to generate test cases. These tests validate if the implementation of a web API conforms to the specification. This process can be automated, such that there is no specific code testing if the API implementation is the same as what is communicated with users of the API. There are currently no frameworks that can generate tests for web APIs with inter parameter constraints, it is also impossible to create a general solution for testing if all defined inter parameter constraints are present, as this is inherent to the testing approach[17].

6.3.2 Inter parameter constraints in programming languages

Being able to express constraints between values in programming languages is the underlying theme of this thesis. This is an idea which is still explored in programming languages research. We have seen that dependent types are powerful enough to express these constraints. There are also other solutions to this problem.

There is some research on adding some way of describing inter parameter constraints to a programming language. In Oostvogels et al. [21] an additional constraint block is added to TypeScript interfaces allowing the description of constraints of a specific interface. This research is based on earlier research on inter parameter constraints in web APIs and proposes a way of describing these constraints in the TypeScript programming language.

Statically checking API consumers is another related topic, Burnay et al. [3] uses static analysis in an extension to JavaScript to validate code that makes REST calls. This paper introduces a specification language which is able to describe all inter parameter constraints and another language which can be used to implement API calls, static analysis is able to verify if the API calls would satisfy all static constraints.

6.3.3 Refinement types

Another research topic which goes into creating more detailed types in programming languages is refinement types. Introduced by Freeman and Pfenning [9] it can be used to define types which are more specific than the types provided by the programming language. It refines types by adding additional preconditions to the type. It allows for example to create a type for a lowercase string. Refinement types are built up from smaller parts, this allows conversion between different refined types. A refined type of integers larger than five can for example be converted to a refined type of integers larger than three. Refinement types solve a problem is similar to the dependent types use-case discussed in this thesis. They are however usually limited to single values and do are not defined for relations between different values in a program or different parts of a nested object. There are (experimental) implementations of refinement types in Haskell[24], TypeScript[25] and Scala[23].

6.3.4 Dependent types in programming languages

There are languages which are able to express inter parameter constraints, such as Agda and to some extend Scala and Haskell. These languages can leverage their type system to represent the different constraints in the types.

Haskell supports dependent types to a certain extend[26], but it might already be possible to implement inter parameter constraints using Haskell.

Scala supports path dependent types[1] and implicits which can be used to assert if the type of an object has a specific trait. This is a less powerful version of dependent types and can be used to describe some inter parameter constraints, this also requires encoding the inter parameter constraints to conform to this structure.

6.4 Future work

This thesis is a first effort in combining the field of web APIs and programming languages in this way. From here there are multiple different paths ahead.

Continuing with the Agda implementation it would be interesting to create a framework for defining OpenAPI definitions in Agda. This framework could be used as a middleware to validate if incoming request conform to the given specification. It could also give insight in why a request is not accepted.

Using dependent types to describe inter parameter constraints is also something that could be made more approachable, we now know that we can use dependent types to express inter parameter constraints, but this still requires knowledge of dependent types, functional programming and Agda to implement. I could be possible to extract the definition of constraints to a smaller constraints language, similar to what previous research has done, but using dependent types in this constraint language.

We know that there are inter parameter constraints in complex web APIs. Being able to communicate them effectively seems to have clear benefits. What the impact is of being able to communicate these inter parameter constraints is something that is still an open question. Currently adding definitions for inter parameter constraints causes significant overhead, this is something that could be addressed.

In Web APIs the input from the request is just one possible input. There are other sources where input can come from, for instance from a database, the values from these sources can also contain constraints with respect to the request. Being able to model this can give an even more precise definition.

Generalizing the problem to describing complex types with constraints between fields in the type system of a programming language is also an interesting problem. Using dependent types is a way to do this, but there might also be more lightweight solutions to this, like extensions of refinement types. Being able to express these constraints in computer programs does not only help in the web APIs domain, but can help computer programmers in general to write more robust computer programs.

Bibliography

[1] Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. *Acm Sigplan Notices*, 49(10):233–249, 2014.

[2] Joop Aué, Maurício Aniche, Maikel Lobbezoo, and Arie van Deursen. An exploratory study on faults inweb api integration in a large-scale payment company. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 13–22. IEEE, 2018.

[3] Nuno Burnay, Antónia Lopes, and Vasco T Vasconcelos. Statically checking rest api consumers. In *International Conference on Software Engineering and Formal Methods*, pages 265–283. Springer, 2020.

[4] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[5] Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, page 133–144, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595936899. doi: 10.1145/1328438.1328457. URL https://doi.org/10.1145/1328438.1328457.

[6] ECMA 404:2013. The json data interchange syntax. Standard, Ecma International, October 2013.

[7] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Automatic generation of test cases for rest apis: A specification-based approach. In *2018 IEEE 22nd international enterprise distributed object computing conference (EDOC)*, pages 181–190. IEEE, 2018.

[8] Matt Fellows. Schemas are not contracts, Jul 2021. URL https://pactflow.io/blog/schemas-are-not-contracts/.

[9] Tim Freeman and Frank Pfenning. Refinement types for ml. *SIGPLAN Not.*, 26(6): 268–277, may 1991. ISSN 0362-1340. doi: 10.1145/113446.113468. URL `https://doi.org/10.1145/113446.113468`.

[10] Henk Grent, Aleksei Akimov, and Maurício Aniche. Automatically identifying parameter constraints in complex web apis: A case study at adyen, 2021.

[11] Isha, Abhinav Sharma, and M. Revathi. Automated api testing. In *2018 3rd International Conference on Inventive Computation Technologies (ICICT)*, pages 788–791, 2018. doi: 10.1109/ICICT43934.2018.9034254.

[12] Justin Lubin and Sarah E Chasins. How statically-typed functional programmers write code. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021.

[13] Per Martin-Löf. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

[14] Alberto Martin-Lopez, Sergio Segura, Carlos Muller, and Antonio Ruiz-Cortés. Specification and automated analysis of inter-parameter dependencies in web apis. *IEEE Transactions on Services Computing*, 2021.

[15] Conor McBride. First-order unification by structural recursion. *Journal of functional programming*, 13(6):1061–1075, 2003.

[16] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[17] J.D. Musa and A.F. Ackerman. Quantifying software validation: when to stop testing? *IEEE Software*, 6(3):19–27, 1989. doi: 10.1109/52.28120.

[18] Ulf Norell. Dependently typed programming in agda. In *International school on advanced functional programming*, pages 230–266. Springer, 2008.

[19] OAS 3.1.0:2021. Openapi specification. Specification, OpenAPI Initiative, February 2021.

[20] Nathalie Oostvogels, Joeri De Koster, and Wolfgang De Meuter. Inter-parameter constraints in contemporary web apis. In *International Conference on Web Engineering*, pages 323–335. Springer, 2017.

[21] Nathalie Oostvogels, Joeri De Koster, and Wolfgang De Meuter. Static typing of complex presence constraints in interfaces. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[22] SM Sohan, Craig Anslow, and Frank Maurer. A case study of web api evolution. In *2015 IEEE World Congress on Services*, pages 245–252. IEEE, 2015.

[23] Frank Thomas. refined: simple refinement types for scala. `https://github.com/fthomas/refined`, 2021.

[24] Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 269–282, 2014.

[25] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement types for typescript. *SIGPLAN Not.*, 51(6):310–325, jun 2016. ISSN 0362-1340. doi: 10.1145/2980983.2908110. URL `https://doi.org/10.1145/2980983.2908110`.

[26] Stephanie Weirich, Antoine Voizard, Pedro Henrique de Amorim, and Richard A Eisenberg. A specification for dependent types in haskell. *Proceedings of the ACM on Programming Languages*, 1(ICFP), 2017.

[27] Erik Wittern, Annie T.T. Ying, Yunhui Zheng, Julian Dolby, and Jim A. Laredo. Statically checking web api requests in javascript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 244–254, 2017. doi: 10.1109/ICSE.2017.30.

[28] Qian Wu, Ling Wu, Guangtai Liang, Qianxiang Wang, Tao Xie, and Hong Mei. Inferring dependency constraints on parameters for web services. In *Proceedings of the 22nd international conference on World Wide Web*, pages 1421–1432, 2013.

Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

API: Application Programming Interface, an interface where other applications can communicate with

REST: Representational state transfer, a web API style. REST APIs are generally decoupled from a specific consumer and can be used by many different applications. There is also no server state, such as sessions specific to a caller, any required state is sent along in the request.

API Consumer: An application which is the user of an API, also known as the client

JSON: JavaScript Object Notation, a light-weight data format, defined in ECMA 404:2013 [6].

XML: Extensible Markup Language, a structured data format.

YAML: YAML Ain't Markup Language™, a configuration language with capabilities similar to JSON.

OpenAPI Initiative: An organisation overlooking the standardization of API specifications

OpenAPI Specification: Describes a vendor neutral way of describing web APIs. Files written based on this specification are called OpenAPI Documents. Updated by the OpenAPI Initiative[19].

OpenAPI document: A document describing the specification of a web API, this document uses and conforms to the OpenAPI Specification. OpenAPI documents are written in either JSON or YAML.

API specification: A general name for an OpenAPI document.

SDK: Software Development Kit, a set of tools that help interacting with a specific library, service or other piece of software. In the context of web APIs an SDK is usually a library which helps setting up a connection with the web API servers and provides a set of functions that delegate API calls to the server and make sure that the input and output of the calls conform to the specification.

Appendix B

OpenAPI Implementation Benchmark

In order to evaluate the impact of changes to the programming language on an API implementation there needs to be a way to evaluate how an implementation performs. How well an implementation performs can be measured using qualitative and quantitative measurements. In this chapter an overview is given of the different metrics used to evaluate the implementation of an API. This evaluation is done on an implementation of a given specification and is only measured between implementations of the same specification using different programming languages or frameworks.

This evaluation focuses on the implementation of the API back-end. Because with a given specification the implemented APIs should be the same. The focus will also lie on measuring the code of the implementation and less on the performance of resulting application as this is mostly influenced by the underlying platform. Any performance issues which are inherent to a solution are reported.

This evaluation serves as an exercise to see what is different when implementing web APIs in different environments. This helps giving the ideas in this thesis a bit more context.

B.1 Implementations

The benchmark specification is implemented in three different frameworks. Each implementation is made to be as close to the benchmark specification as possible. In this section we introduce the different implementations.

We compare an implementation using Spring framework with an implementation in a more traditional framework used at Adyen and the Dependently typed language Agda. These environments are vastly different, for this evaluation we look at the differences of specific parts of the implementation. We do this to see in what way these different parts influence how web APIs are implemented.

B.1.1 Spring Framework

The Spring[1] framework is widely used to implement web APIs. The implementation of the benchmark using Spring is used as a baseline to compare the other implementations to.

This implementation is based on Spring Boot, which is an opinionated part of the Spring framework. This provides a starting point for implementing web APIs.

B.1.2 Adyen Framework

At Adyen a framework is built to implement web APIs. This framework tailors to Adyen's needs when building a complex web API. The framework used at Adyen has support for managing multiple versions of an API and creating complex workflows in an API endpoint.

The framework was initially a SOAP[2] and RPC framework and was later updated to also allow creating REST APIs. This legacy has made the framework very extensive, supporting much more than just REST APIs.

B.1.3 Agda

In Chapter 4 we gave a description of Agda and dependent types. This implementation builds on top of that. This implementation does not account for the implementation of the http layer, there is currently no Agda implementation for this. The implementation uses a model of what an implementation of the http layer might look like.

This implementation moves all validation to a very early point of the request, without causing other parts of the implementation to lose this information. This is done by embedding all information about the constraints in the corresponding types.

B.2 API implementation benchmark

To create a benchmark for an API implementation, there needs to be some reference specification that can be used to create the same API using different languages or frameworks. For usage in this research the benchmark API would need to have some interesting inter parameter constraint to show how these are implemented.

B.2.1 Constraints

The pet store API specification [3] is a starting point for this. It contains some basic endpoints to mutate data. Some more additions need to be made to add some inter parameter constraints, these additions cannot be expressed using the standard open API specification, but are defined below.

Present-Value constraint In Address, add the field `country`, if `country` is US or CA, then the `state` field is required.

[1]`https://spring.io`

[2]`https://www.w3.org/TR/soap/`

[3]`https://github.com/swagger-api/swagger-petstore`

Present-Present constraint In User, if `firstName` is set, then `lastName` must also be set. But setting only `lastName` is also valid.

Value-Value constraint Order: if `complete` is `true`, then `status` is `delivered`.

Value constraints In User: `email` must look like an email address (`/.+@.+/`). In Category, `name` must be between 2 and 10 characters long. In Pet, `name` must be at least two characters long.

Group constraint In Pet, the fields `category` and `status` are either both empty or both set. In User, at least `phone` or `email` must be set.

These constraints must throw an error when the validation fails.

Implementations with all these constraints are compared with each other using different metrics.

B.3 Quantitative Measures

Quantitative measures are a way of looking at the source code to extract metrics. In this section we will go over some quantitative metrics. Due to the small scale of this benchmark the actual values behind the measures was not deemed relevant. A short explanation of the impact of these variables is given for each metric.

B.3.1 LoC

Lines of code can give an indication of how verbose an implementation is. This metric should be combined with other metrics to give actual insight in the meaning of the lines of code.

Spring Creating a web API with Spring boot requires some boilerplate code. To configure the application, much of this can be generated by the OpenAPI generator.

Creating a new endpoint in Spring boot does not require many new lines of code. Spring can use annotations to define new endpoints. After registering a controller a new endpoint can be added by adding a function is annotated with the url of the endpoint.

Adyen The Adyen web API framework is similar to Spring in many senses, as it is also based on Java and borrows many ideas.

The only way to configure the Adyen web API framework is with XML, this is more verbose than using annotations. The framework also supports complex actions based on multiple tasks which are called in sequence, this also requires more lines of code, but can greatly improve readability and re-usability for very complex tasks.

Agda The Agda implementation does not integrate with a larger system, so it cannot be compared to other frameworks in this sense.

The Agda implementation does require more verbose types to describe requests and responses. This results in requiring more lines of code.

B.3.2 Cyclomatic Complexity

McCabe [16] is a metric to measure how complex a piece of code is. If the validation code itself has many branches or if the actual complexity can be handled by a framework or solver. This is a way to measure how complex an implementation is.

Spring and Adyen Both the Spring and Adyen frameworks behave very similar in terms of cyclomatic complexity. Most of the business logic code in both implementations is the same.

Agda In Agda the complexity is hidden away as most types use a declarative notation. This does not mean that there is no complexity. There are ways to calculate complexity of a functional program[5], but this is not a general solution.

B.3.3 Class Coupling

Class coupling[4] measures how many classes depend on each other. How tight classes in the implementation are coupled and if re-using of classes is encouraged or discouraged.

Spring and Adyen Both Spring and the Adyen framework are based on Java, they have similar ways of implementing business logic.

Agda In a functional programming language such as Agda the notion of class coupling has no meaning. There are other ways code can be coupled in a language such as Agda, but it is still mostly based on other components present in the implementation the same way as in the other implementations.

B.4 Qualitative Measures

B.4.1 Testability

How are different parts of the implementation testable? Is it possible to write tests on the validation side? Can controller methods be unit tested?

Testability is partly up to what a framework provides.

Spring Extensive testing tools for Spring exist.

Adyen In the Adyen framework there are there exist mostly integration and unit tests.

	Spring	Adyen	Agda
Testability	●	◐	◐
Maintainability	●	◐	●
Versionability	◐	●	○
Automation	◐	◐	●
Understandability	●	◐	◐
Debugging	●	◐	○
Code Generation	●	◐	◐

Table B.1: Qualitative measures

Agda In Agda there isn't a testing framework available, it is possible to create tests by writing assertions on types. If the contract changes you first need to fix these tests before the program will compile again.

B.4.2 Maintainability

How easy is it to add new validation logic or fields to a request. How many different files need to be changed when a field is added.

Spring Spring has a lot of tooling available, helping with maintainability.

Adyen The Adyen framework is a very specific solution and build in-house. This means that tooling is limited compared to other frameworks.

Agda An Agda implementation will fail to build until the implementation and the specification are both in sync, this makes it hard to miss when a field is removed but still used somewhere or when an assumption about an input is not valid anymore.

B.4.3 Versionability

Is there some way of maintaining versions supported by the framework? If there is, how flexible is it?

Maintaining different versions of the same web API can be a requirement for complex web APIs[22]. One way of maintaining multiple versions is to run multiple versions of the same api in different versions. Another way is to let the implementation handle the version information and return specific fields or values based on the requested version, this allows for continuous support of multiple versions, but does increase the chance that previous versions are updated by accident.

Maintaining multiple versions is something that is implemented by the framework.

Spring Spring has some support for implementing different versions. It is not supported out of the box.

Adyen The Adyen framework has extensive support for versioning, allowing an implementation to support multiple versions at once, going beyond what is possible with OpenAPI specifications.

Agda The Agda implementation does not support versioning, but this is something that could be added in a future version.

B.4.4 Automation

How much of the implementation can be handled by the framework? Specifically how much of the validation logic is handled by the framework. How expressive is defining validation logic?

Web API frameworks are responsible for automating parts of the implementation of a web api. Parts which are usually automated include serialization and de-serialization of requests and responses, which also includes the first validation steps to check if a request is actually well-formed. Routing of requests is also often handled by the framework.

Spring Spring can handle

B.4.5 Understandability

How much effort is it to find your way around in the implementation? Given that you know the programming language, but not the framework.

Spring There are many resources for learning how to work with the Spring framework and common issues that can be ran into.

Adyen As expected from an internal framework, the

Agda

B.4.6 Debugging

How easy is it to debug faults in the definitions or implementation? Does the language or framework provide a way to inspect what is going on, by stepping through the code for example.

Spring The Spring framework has extensive tooling to aid with debugging, different IDEs come with debuggers that are tailored to debugging a Spring framework application.

Adyen The Adyen framework can be debugged with the Java debugger, but there are parts of the process that are hard to follow with a debugger.

Agda The Agda code itself does not execute, so if there is an error in the specification for example, there are no ways to use a debugger to find these problems.

B.4.7 Code Generation

How much can be automatically generated? Are there large parts that strictly follow the specification such that an implementation can be easily generated.

Spring For the Spring framework there are already tools that can generate the scaffolding for a web API implementation. After generating the developer only needs to add the actual implementation for each endpoint.

Adyen The Adyen framework has not embraced code generation and is implemented in such a way that the implementation is the source of truth where a specification is generated from.

Agda There are currently no generators which generate code from an OpenAPI specification, but we show in Chapter 4 that an OpenAPI specification can map one to one to an Agda implementation.

B.5 Results

We have compared implementing a basic web API with inter parameter constraints in different environments. We found that while different environments share similarities, they are also very different on other parts. And changing the programming languages causes a developer to have an entirely different way of programming[12].

The programming language has the biggest impact on how a web API is implemented, web API implementations have parts which are not handled by the framework. These parts is often where the complexity of the implementation lies. The implementations here cannot be generalized to a framework and must be implemented without much help from the framework.

Some work still needs to be done before we can create a complete web API implementation in Agda.

Appendix C

Agda Code

```
{-# OPTIONS --type-in-type #-}

open import Data.Product hiding (map)
open import Data.Sum using (_⊎_ ; inj₁ ; inj₂)
open import Data.List as List
open import Data.String as String
open import Data.Maybe hiding (fromMaybe)
open import Data.Empty
open import Data.Unit
open import Level
open import Function
open import Data.Bool as Bool
open import Data.Nat

open import Data.Unit using (tt)

open import Relation.Binary.PropositionalEquality
open import Relation.Nullary
open import Relation.Nullary.Reflects using (invert)
open import Relation.Unary hiding (_∈_ ; _∉_ ; Decidable)

import Data.Maybe.Relation.Unary.Any as MaybeAny

open import Data.List.Membership.Propositional

open import MaybeRel

module OpenAPI where
```

```
open import FRP.JS.JSON as JSON using (JSON ; _·_ ; _∈·_ ;
 ↪ dlookup ; JsonObj ; getObject) renaming (lookup to
 ↪ jsonLookup)

open import OpenAPI.Type as Type renaming (_≟_ to _t≟_)
open import OpenAPI.Schema

decNoToMaybe : ∀ {a} {A : Set a} → Dec A → Maybe (¬ A)
decNoToMaybe (yes x) = nothing
decNoToMaybe (no x)  = just x

data Dep : Set where
  someDep : (D : (JSON → Set)) → (∀ (json : JSON) → Maybe (D
   ↪  json)) → Dep
  noDep : Dep

containsField? : ∀ {B : Set} (f : String) → (s2 : List
 ↪  (String × B)) → Maybe (∃ λ o → o ∈ s2 × f ≡ proj₁ o)
containsField? f s2 = do
  a ← any? (λ x → decToMaybe (f String.≟ proj₁ x)) s2
  just (find a)

data Component : Set where
  component : List (String × Schema Dep) → Component

data OpenAPI : Set where
  openapi : Component → List (String × (Schema Dep × Schema
   ↪  Dep)) → OpenAPI

data HasRequired : List (String × JSON) → String → Set where
  hr : ∀ { name flds } →
     (∃ λ val → val ∈ flds × proj₁ val ≡ name × proj₂ val ≢
      ↪  JSON.null) →
     HasRequired flds name

data HasRequireds : List (String × JSON) → List String → Set
 ↪  where
  hrs : ∀ { flds req } →
     All (HasRequired flds) req →
     HasRequireds flds req

notNull? : (json : JSON) → Maybe (json ≢ JSON.null)
notNull? JSON.null = nothing
notNull? (JSON.string x) = just (λ ())
```

```
notNull? (JSON.float x) = just (λ ())
notNull? (JSON.bool x) = just (λ ())
notNull? (JSON.array x) = just (λ ())
notNull? (JSON.object x) = just (λ ())
notNull? (JSON.number x) = just (λ ())

hasRequired? : (flds : List (String × JSON)) → (req : String)
 ↪  → Maybe (HasRequired flds req)
hasRequired? flds req = do
  (req’ , json) , req☑ , refl ← containsField? req flds
  null☑ ← notNull? json
  just (hr ((req’ , json) , req☑ , refl , null☑))

hasRequireds? : (flds : List (String × JSON)) → (rs : List
 ↪  String) → Maybe (HasRequireds flds rs)
hasRequireds? flds rs = do
 hasreq ← all? (hasRequired? flds) rs
 just (hrs (hasreq))

mapJsonType : JSON → Type
mapJsonType JSON.null = Type.null
mapJsonType (JSON.string _) = string
mapJsonType (JSON.float _) = float
mapJsonType (JSON.bool _) = bool
mapJsonType (JSON.array _) = array
mapJsonType (JSON.object _) = object
mapJsonType (JSON.number _) = number

jsonType : JSON → Type → Bool
jsonType json t with json | t
... | JSON.null | _ = true
... | JSON.string _ | string = true
... | JSON.float _ | float = true
... | JSON.bool _ | bool = true
... | JSON.array _ | array = true
... | JSON.object _ | object = true
... | JSON.number _ | number = true
... | _ | _ = false

data JsonHasType : JSON → Type → Set where
  jht : ∀ { json type } →
    (mapJsonType json) ≡ type →
    JsonHasType json type
```

```
data ListHasType : Schema Dep → JSON → (Schema Dep → JSON →
 ↪  Set) → Set where
  lht : ∀ { s json ⊢ } →
    All (⊢ s) ((List.concat ∘ fromMaybe ∘ JSON.getArray) json
     ↪  ) →
    ListHasType s json ⊢

data ObjectHasType : List (String × Schema Dep) → JSON →
 ↪  (Schema Dep → JSON → Set) → Set where
  oht : ∀ { properties json ⊢ } →
    All (λ (f , s) → All (λ j → ⊢ s j) (fromMaybe (json •
     ↪  f))) properties →
    ObjectHasType properties json ⊢

data FieldHasType : String → JSON → List (String × Schema
 ↪  Dep) → (Schema Dep → JSON → Set) → Set where
  fht : ∀ {⊢ f json o s2} →
    (f , o) ∈ s2 →
    ⊢ o json →
    FieldHasType f json s2 ⊢

jsonHasType? : (type : Type) → (json : JSON) → Maybe
 ↪  (JsonHasType json type)
jsonHasType? type json = do
  type☑ ← decToMaybe ((mapJsonType json) t≟ type)
  just (jht type☑)

data ValidDep : Dep → JSON → Set where
  validNoDep : ∀ {json} → ValidDep noDep json
  validSomeDep : ∀ { D d2 json } →
    (D json) →
    ValidDep (someDep D d2) json

data ⊢s : Schema Dep → JSON → Set where
  wt : ∀ { type json } →
    JsonHasType json type →
    ---------------
    ⊢s (schema type) json
  wta : ∀ { s json } →
    (∃ λ ls → json ≡ (JSON.array ls) × All (⊢s s) ls) →
    ---------------
    ⊢s (aschema s) json
  wto : ∀ { dep reqs s2 fields } →
    HasRequireds fields reqs →
    All (λ (f , json) → ∃ λ (o) → (o ∈ s2) × (f ≡ proj₁ o) ×
     ↪  (⊢s (proj₂ o) json)) fields →
    ValidDep dep (JSON.object fields) →
    ---------------
    ⊢s (oschema s2 reqs dep) (JSON.object fields)

data ⊢tout : OpenAPI → String → JSON → Set where
  wtout :  ∀ { κ πs p jo σ₁ σ₂ } →
     ((p , (σ₁ , σ₂)) ∈ πs) → (⊢s σ₂ jo) →
    -------------
    ⊢tout (openapi κ πs) p jo

data ⊢tin : OpenAPI → String → JSON → Set where
  wtin :  ∀ { κ πs p ji σ₁ σ₂ } →
     ((p , (σ₁ , σ₂)) ∈ πs) → (⊢s σ₁ ji) →
    -------------
    ⊢tin (openapi κ πs) p ji

data ⊢t : OpenAPI → String → JSON → JSON → Set where
  wt : ∀ { κ πs p ji jo σ₁ σ₂ } →
     ((p , (σ₁ , σ₂)) ∈ πs) → (⊢s σ₁ ji) → (⊢s σ₂ jo) →
    -------------
    ⊢t (openapi κ πs) p ji jo

{-# TERMINATING #-}
solveSchema : (s : Schema Dep) → (json : JSON) → Maybe (⊢s s
 ↪  json)

validateDep? : (c : Dep) → (json : JSON) → Maybe (ValidDep c
 ↪  json)
validateDep? noDep json = just validNoDep
validateDep? (someDep dep check) json = do
  dep☑ ← check (json)
  just (validSomeDep dep☑)

fieldHasType? : (s2 : List (String × Schema Dep)) → (fld :
 ↪  String × JSON) → Maybe (∃ λ o → (o ∈ s2) × (proj₁ fld ≡
 ↪  proj₁ o) × (⊢s (proj₂ o) (proj₂ fld)))
fieldHasType? s (f , json) = do
  (f′ , σ) , ∈☑ , f☑ ← containsField? f s
  σ☑ ← solveSchema σ json
  just ((f′ , σ) , ∈☑ , f☑ , σ☑)
```

```
jsonObj? : (json : JSON) → Maybe (∃ λ flds → json ≡
 ↪  JSON.object flds)
jsonObj? (JSON.object flds) = just (flds , refl)
jsonObj? _ = nothing

jsonArr? : (json : JSON) → Maybe (∃ λ b → json ≡ JSON.array b)
jsonArr? (JSON.array x) = just (x , refl)
jsonArr? _ = nothing

jsonStr? : (json : JSON) → Maybe (∃ λ b → json ≡ JSON.string
 ↪  b)
jsonStr? (JSON.string x) = just (x , refl)
jsonStr? _ = nothing

jsonNum? : (json : JSON) → Maybe (∃ λ b → json ≡ JSON.number
 ↪  b)
jsonNum? (JSON.number x) = just (x , refl)
jsonNum? _ = nothing

solveSchema (schema type) json = do
  type☑ ← jsonHasType? type json
  just (wt type☑)
solveSchema (aschema σ) json = do
  (arr , arr☑) ← jsonArr? json
  σ☑ ← all? (solveSchema σ) arr
  just (wta (arr , arr☑ , σ☑))
solveSchema (oschema fields reqs deps) (JSON.object obj) = do
  reqs☑ ← hasRequireds? obj reqs
  fields☑ ← all? (fieldHasType? fields) obj
  deps☑ ← validateDep? deps (JSON.object obj)
  just (wto reqs☑ fields☑ deps☑)
solveSchema ($ref x) json = nothing -- TODO Implement $ref
solveSchema _ _ = nothing

solve : (s : OpenAPI) → (p : String) → (json↓ : JSON) →
 ↪  (json↑ : JSON) → Maybe (⊢t s p json↓ json↑)
solve (openapi _ πs) p json↓ json↑ = do
  ((p' , σ₁ , σ₂) , ∈☑ , refl) ← containsField? p πs
  σ₁☑ ← solveSchema σ₁ json↓
  σ₂☑ ← solveSchema σ₂ json↑
  just (wt ∈☑ σ₁☑ σ₂☑)

module Example where
  petSchema : Schema Dep
  petSchema = aschema (schema string)

  outSchema : Schema Dep
  outSchema = schema string

  openApi : OpenAPI
  openApi = openapi (component []) (("/pets" , petSchema ,
   ↪  petSchema) ∷ ("/pet" , petSchema , outSchema) ∷ [])

  inJSON : JSON
  inJSON = JSON.array (JSON.string "a" ∷ JSON.string "b" ∷
   ↪  [])

  outJSON : JSON
  outJSON = JSON.string "bye"

  solved : ⊢t openApi "/pet" inJSON outJSON
  solved = from-just (solve openApi "/pet" inJSON outJSON)

module ExampleObj where
  data ValidJson : JSON → Set where
    hasnameandage : ∀ {flds json name age jsonage} →
      json ≡ JSON.object flds →
      ("name" , name) ∈ flds →
      ("age" , age) ∈ flds →
      age ≡ (JSON.number jsonage) →
      name ≡ (JSON.string "Flap") →
      jsonage Data.Nat.< 3 →
      ValidJson json

  validateValidJson : (json : JSON) → Maybe (ValidJson json)
  validateValidJson json = do
    (flds , flds☑) ← jsonObj? json
    ((_ , flap) , flap☑ , refl) ← containsField? "name" flds
    ((_ , age) , age☑ , refl) ← containsField? "age" flds
    (flapStr , refl) ← jsonStr? flap
    (ageNum , refl) ← jsonNum? age
    refl ← decToMaybe (flapStr String.≟ "Flap")
    lesAge ← decToMaybe (ageNum Data.Nat.<? 3)
    just (hasnameandage flds☑ flap☑ age☑ refl refl lesAge)

  deps : Dep
  deps = someDep ValidJson validateValidJson
  petSchema : Schema Dep
  petSchema = oschema (("name" , schema string) :: ("age" ,
   ↪  schema number) :: []) [] deps

  openApi : OpenAPI
  openApi = openapi (component []) (("/pets" , petSchema ,
   ↪  schema number) :: [])

  inJSON : JSON
  inJSON = JSON.object (("name" , JSON.string "Flap") ::
   ↪  ("age" , JSON.number 2) :: [])

  outJSON : JSON
  outJSON = JSON.number 1

  solvedS : ⊢s petSchema inJSON
  solvedS = from-just (solveSchema petSchema inJSON)

  -- dep : Dep
  -- dep = someDep (λ js → maybe (λ y → y == "asdf") false
   ↪  (js • "stuff"))

  api : ∀ {json p} → (wf : ⊢tin openApi p json) → ∃ λ jsout →
   ↪  (⊢tout openApi p jsout)
  api (wtin (here refl) (wto (hrs []) x₁ (validSomeDep
   ↪  (hasnameandage x x₂ x₃ x₄ x₅ x₆))))) = JSON.number 1 ,
   ↪  wtout (here refl) (wt (jht refl))

module ExampleInter where
  data ValidJs : JSON → Set where
    validNameHasVal : ∀ {flds json name age jsage} →
      json ≡ JSON.object flds →
      ("name" , name) ∈ flds →
      ("age" , age) ∈ flds →
      age ≡ (JSON.number jsage) →
      name ≡ (JSON.string "Flap") →
      ValidJs json
    validNameHasNoVal : ∀ {flds json name} →
      json ≡ JSON.object flds →
      ("name" , name) ∈ flds →
      name ≢ (JSON.string "Flap") →
      ValidJs json

  validJsCheck : (json : JSON) → Maybe (ValidJs json)
  validJsCheck json = do
    (flds , flds☑) ← jsonObj? json
    ((_ , flap) , flap☑ , refl) ← containsField? "name" flds
    (flapStr , refl) ← jsonStr? flap
    yes refl ← just (flapStr String.≟ "Flap")
          where no ¬a → just (validNameHasNoVal flds☑ flap☑ λ
           ↪  {refl → ¬a refl})
    ((_ , age) , age☑ , refl) ← containsField? "age" flds
    (ageNum , refl) ← jsonNum? age
    just (validNameHasVal flds☑ flap☑ age☑ refl refl)

  deps : Dep
  deps = someDep ValidJs validJsCheck

  petSchema : Schema Dep
  petSchema = oschema (("name" , schema string) :: ("age" ,
   ↪  schema number) :: []) ("name" :: []) deps

  openApi : OpenAPI
  openApi = openapi (component []) (("/pets" , petSchema ,
   ↪  schema number) :: [])

  inJSON : JSON
  inJSON = JSON.object (("name" , JSON.string "Flapas")  ::
   ↪  [])

  outJSON : JSON
  outJSON = JSON.number 1

  solved : ⊢t openApi "/pets" inJSON outJSON
  solved = from-just (solve openApi "/pets" inJSON outJSON)

  inJSON2 : JSON
  inJSON2 = JSON.object (("name" , JSON.string "Flap") ::
   ↪  ("age2" , JSON.number 5) :: [])

  -- solved2 : ⍰t openApi "/pets" inJSON2 outJSON
  -- solved2 = from-just (solve openApi "/pets" inJSON2
   ↪  outJSON)

  api : ∀ {json p} → (wf : ⊢tin openApi p json) → ∃ λ jsout →
   ↪  (⊢tout openApi p jsout)
```

```
  api {.(JSON.object _)} {."/pets"} (wtin (here refl) (wto x
   ↪  x₁ x₂)) = JSON.number (1) , wtout (here refl) (wt (jht
   ↪  refl))

module Example2 where
  openApi : OpenAPI
  openApi = openapi (component []) (("/pets" , schema string
   ↪  , schema number) :: [])

  json : JSON
  json = JSON.object (("stuff" , JSON.string "asdf") ::
   ↪  ("stuff2" , JSON.string "fdsa") :: [])

  c : Schema Dep
  c = oschema (("stuff" , schema string) :: ("stuff2" , schema
   ↪  string) :: []) [] noDep

  proofSchem : Is-just (solveSchema c json)
  proofSchem = MaybeAny.just tt

  -- solvedSchem : ⍰s c json
  -- solvedSchem with proofSchem
  -- ... | just {a} tt = a

  str : Schema Dep
  str = schema string

  data AndReq : JSON -> Set where
    and_req : ∀ {json flds a av b bv}
      → json ≡ JSON.object flds
      → (a , av) ∈ flds
      → (b , bv) ∈ flds
      → AndReq json

  andDep : String → String → Dep
  andDep = λ a → λ b → (someDep AndReq λ json → do
     (flds , flds☒) ← jsonObj? json
     ((_ , flap) , a☒ , refl) ← containsField? a flds
     (astring , refl) ← jsonStr? flap
     ((_ , bval) , b☒ , refl) ← containsField? b flds
     (bstring , refl) ← jsonStr? flap
     just  (and_req {json} flds☒ a☒ b☒))

  inverseDec : ∀ {A : Set} → Dec (A) → Dec (¬ A)
  inverseDec (yes a) = no λ z → z a
  inverseDec (no ¬a) = yes ¬a

  notin? : (x : String) → (y : List String) → Maybe (x ∉ y)
  notin? fld flds = none? (λ x → decToMaybe (inverseDec (fld
   ↪  String.≟ x))) flds

  in? : (x : String) → (y : List String) → Maybe (x ∈ y)
  in? fld flds = any? (λ x → decToMaybe (fld String.≟ x)) flds

  data AllOrNoneReq : JSON → Set where
    allornone_all : ∀ {json flds grp} →
      json ≡ JSON.object flds →
      (All (λ name → (name ∈ (List.map proj₁ flds))) grp) ⊎
       ↪  (All (λ name → (name ∉ (List.map proj₁ flds))) grp)
       ↪  →
      AllOrNoneReq json

  allOrNoneCheck : (List String) → (json : JSON) → Maybe
   ↪  (AllOrNoneReq json)
  allOrNoneCheck = λ allflds → λ json → do
    (flds , fldscheck) ← jsonObj? json
    just all☑ ← just (all? (λ y → in? y (List.map proj₁
     ↪  flds)) allflds)
      where nothing → do
        none☑ ← all? (λ y → notin? y (List.map proj₁ flds))
         ↪  allflds
        just (allornone_all fldscheck (inj₂ none☑))
    just (allornone_all fldscheck (inj₁ all☑))

  allOrNoneDep : List String → Dep
  allOrNoneDep = λ allflds → (someDep AllOrNoneReq
   ↪  (allOrNoneCheck allflds))

  d : Schema Dep
  d = oschema (("a" , schema string) :: ("b" , schema string)
   ↪  :: []) [] (andDep "a" "b")

module Validator where
  {-# TERMINATING #-}
  validator : (openApi : OpenAPI) → (p : String) → (json :
   ↪  JSON) → Maybe (⊢tin openApi p json)
  validator (openapi x []) p json = nothing -- empty
   ↪  specification
  validator (openapi x ((fst , snd , snd₁) ∷ x₂)) p json with
   ↪  (fst String.≟ p)
  ... | yes refl = do
    valid ← solveSchema snd json
    just (wtin (here refl) valid)
  ... | no ¬a = do
    (wtin a b) ← validator (openapi x x₂) p json
    just (wtin (there a)  b)
```

Listing C.1: OpenAPI.agda

```
{-# OPTIONS --allow-unsolved-metas #-}
open import Data.String
open import Data.List as List using (List)
open import Data.Bool

open import Data.Product
open import Data.Maybe
open import Relation.Binary.PropositionalEquality
open import Relation.Nullary
open import Data.List

open import Function
open import Level

open import MyAny

open import OpenAPI.Type renaming (_≟_ to _t≟_)
open import FRP.JS.JSON.Base

module OpenAPI.Schema where

data Schema : Set where
  schema : Type → Schema
  aschema : Schema → Schema
  oschema : (flds : List (String × Schema)) → (reqs : List
   ↪  String) → Schema
  $ref : String → Schema

infix 5 _==?_

lMaybeEqual : ∀ {A : Set} (x : ((c d : A) → Maybe (c ≡ d)))
 ↪  (a b : List A) → Maybe (a ≡ b)
lMaybeEqual x [] [] = just refl
lMaybeEqual x [] (x₁ :: b) = nothing
lMaybeEqual x (x₁ :: a) [] = nothing
lMaybeEqual x (x₁ :: a) (x₂ :: b) with x x₁ x₂
... | just refl = maybe (λ { refl → just refl}) nothing
 ↪  (lMaybeEqual x a b)
... | nothing = nothing


oSchemaEqual : (a b : (String × Schema)) → (x : ((x₁ x₂ :
 ↪  String) → Maybe (x₁ ≡ x₂))) → (y : ((y₁ y₂ : Schema) →
 ↪  Maybe (y₁ ≡ y₂))) → Maybe (a ≡ b)
oSchemaEqual (fst , snd) (fst₁ , snd₁) x y with x fst fst₁ |
 ↪  y snd snd₁
... | just refl | just refl = just refl
... | _ | _ = nothing

_s==?_ : (a b : String) → Maybe (a ≡ b)
a s==? b = decToMaybe (a Data.String.≟ b)


{-# TERMINATING #-}
_==?_ : (a : Schema) → (b : Schema) → Maybe (a ≡ b)
schema x ==? schema x₁ with x t≟ x₁
... | yes refl = just refl
... | no ¬p = nothing
schema x ==? aschema x₁ = nothing
schema x ==? oschema flds reqs = nothing
schema x ==? $ref x₁ = nothing
aschema x ==? schema x₁ = nothing
aschema x ==? aschema x₁ with x ==? x₁
... | just refl = just refl
... | nothing = nothing
aschema x ==? oschema flds reqs = nothing
aschema x ==? $ref x₁ = nothing
oschema flds reqs ==? schema x = nothing
oschema flds reqs ==? aschema x₁ = nothing
oschema [] [] ==? oschema [] [] = just refl
oschema _ [] ==? oschema _ _ = nothing
oschema [] _ ==? oschema _ _ = nothing
oschema _ _ ==? oschema [] _ = nothing
oschema _ _ ==? oschema _ [] = nothing
oschema ((a , a₁) :: axs) (x :: xs) ==? oschema ((b , b₁) ::
 ↪  bxs) (x₁ :: xs₁) with x s==? x₁ | a s==? b | a₁ ==? b₁
```

```
... | just refl | just refl | just refl = maybe′ (λ {refl →
 ↪  just refl}) nothing (oschema axs xs ==? oschema bxs xs₁)
... | _ | _ | _ = nothing
oschema flds reqs ==? $ref x = nothing
$ref x ==? schema x₁ = nothing
$ref x ==? aschema x₁ = nothing
$ref x ==? oschema flds reqs = nothing
$ref x ==? $ref x₁ with x s==? x₁
... | just refl = just refl
... | _ = nothing
```

Listing C.2: OpenAPI/Schema.agda

```
open import Function.Base
open import Level

import Data.Nat.Base as ℕ
import Data.Nat.Properties as ℕp

open import Relation.Nullary
open import Relation.Nullary.Decidable using (map′; isYes)
open import Relation.Binary
open import Relation.Binary.PropositionalEquality as PropEq
 ↪  using (_≡_; _≢_; refl; cong; sym; trans; subst)

module OpenAPI.Type where

data Type : Set where
  null string object float bool array number : Type

toℕ : Type → ℕ.ℕ
toℕ null = 0
toℕ string = 1
toℕ object = 2
toℕ float = 3
toℕ bool = 4
toℕ array = 5
toℕ number = 6

toℕ-injective : ∀ {i j : Type} → toℕ i ≡ toℕ j → i ≡ j
toℕ-injective {null} {null} _ = refl
toℕ-injective {string} {string} _ = refl
toℕ-injective {object} {object} _ = refl
toℕ-injective {float} {float} _ = refl
```

```
toℕ-injective {bool} {bool} _ = refl
toℕ-injective {array} {array} _ = refl
toℕ-injective {number} {number} _ = refl

infix 4 _≈_
_≈_ : Rel Type 0ℓ
_≈_ = _≡_ on toℕ

≈⇒≡ : _≈_ ⇒ _≡_
≈⇒≡ = toℕ-injective

≈-reflexive : _≡_ ⇒ _≈_
≈-reflexive = cong toℕ

infix 4 _≟_
_≟_ : Decidable {A = Type} _≡_
x ≟ y = map′ ≈⇒≡ ≈-reflexive (toℕ x ℕₚ.≟ toℕ y)
```

Listing C.3: OpenAPI/Type.agda

```
------------------------------------------------------------------------
-- The Agda standard library
--
-- Machine words: basic type and conversion functions
------------------------------------------------------------------------

module FRP.JS.JSON.Base where

open import Level using (zero)
import Data.Nat.Base as ℕ
open import Function
open import Relation.Binary using (Rel)
open import Relation.Binary.PropositionalEquality
open import Data.String using (String)
open import Data.Bool using (Bool)
open import Data.Float using (Float)
open import Data.List using (List)
open import Data.Product using (_×_)
open import Data.Maybe.Base

------------------------------------------------------------------------
```

```
-- Re-export built-ins publicly


data JSON : Set where
  null : JSON
  string : String → JSON
  float : Float → JSON
  bool : Bool → JSON
  array : List (JSON) → JSON
  object : List (String × JSON) → JSON
  number : ℕ.ℕ → JSON
```

Listing C.4: FRP/JS/JSON/Base.agda

```
open import Data.Bool as Bool using ( Bool ; true ; false ;
 ↪  not ; _∧_)
open import Data.String as String using ( String ) renaming (
 ↪  _==_ to _==s_ )
import Data.String.Properties as Sp
open import Data.Float as Float using ( Float ) renaming (
 ↪  _≡ᵇ_ to _==n_ )
open import Data.Maybe using ( Maybe ; just ; nothing ; _>>=_
 ↪  ; decToMaybe ; maybe)
open import Data.Nat as Nat using ( ℕ )

open import Data.List as List using ( List ; [] ; _∷_ ; map )
open import Data.List.Properties using (≡-dec ; ∷-dec)
open import Data.List.Membership.Propositional using (_∈_)
open import Data.Unit using (tt)
open import Data.Product using (_×_ ; _,_ ; proj₁ ; proj₂)
open import Data.Product.Properties as P using ()
open import Data.Empty

open import Function

import Data.List.Relation.Unary.Any as lAny

open import Level using (0ℓ)

open import Relation.Nullary using (Dec ; yes ; no)
open import Relation.Nullary.Decidable using (map′ ; isYes)

open import Relation.Binary using (Decidable ;
 ↪  DecidableEquality)
```

```
open import Relation.Binary.PropositionalEquality using (_≡_
 ↪  ; refl)

open import FRP.JS.Array using ( ) renaming ( lookup? to
 ↪  alookup? ; _≟[_]_ to _≟a[_]_ )

module FRP.JS.JSON where

open import FRP.JS.JSON.Base public

private
  variable
    b : Bool
    B : Set

_==b_ : Bool → Bool → Bool
true ==b a = a
false ==b a = not a

postulate
  show  : JSON → String
  parse : String → Maybe JSON

Key : Bool → Set
Key true  = String
Key false = ℕ

lookup? : Maybe (JSON) → ∀ {b} → Key b → Maybe (JSON)
lookup? (just (object js)) {true}  k = maybe (just ∘ proj₂)
 ↪  nothing (List.head (List.filter (λ x → proj₁ x String.≟
 ↪  k) js))
lookup? (just (array js))  {false} i = alookup? js i
lookup? _                          _ = nothing

lookup : JSON → ∀ {b} → Key b → Maybe (JSON)
lookup json key = lookup? (just json) key

getString? : Maybe (JSON) → Maybe String
getString? (just (string s)) = just s
getString? _ = nothing

getℕ? : Maybe (JSON) → Maybe ℕ
getℕ? (just (number n)) = just n
getℕ? _ = nothing
```

```
getObject : JSON → Maybe (List (String × JSON))
getObject (object a) = just a
getObject _ = nothing

getArray : JSON → Maybe (List (JSON))
getArray (array a) = just a
getArray _ = nothing

getArray? : Maybe (JSON) → Maybe (List (JSON))
getArray? (just (array a)) = just a
getArray? _ = nothing

getBool? : Maybe (JSON) → Maybe Bool
getBool? (just (bool a)) = just a
getBool? _ = nothing

record LookupJson (B : Set) : Set where
  field
    theJson : ∀ {b} → JSON → Key b → Maybe B
open LookupJson

infix 100 _•_
_•_ : {{_ : LookupJson B}} → JSON → Key b → Maybe B
_•_ {{bl}} = bl .theJson

instance
  lookupBoolean : LookupJson Bool
  theJson lookupBoolean x k = getBool? (lookup x k)

  lookupString : LookupJson String
  theJson lookupString x k = getString? (lookup x k)

  lookupℕ : LookupJson ℕ
  theJson lookupℕ x k = getℕ? (lookup x k)

  lookupJSON : LookupJson JSON
  theJson lookupJSON x k = lookup x k

  lookupList : LookupJson (List JSON)
  theJson lookupList x k = getArray? (lookup x k)
```

```
postulate
  _js==?_ : (a b : JSON) → Maybe (a ≡ b)

data JsonObj : JSON → Set where
  jsIsObj : ∀ {js flds}
    → js ≡ (object flds)
    → JsonObj js

data _∈•_ : String → List (String × JSON) → Set where
  objhas : ∀ {a flds}
    → a ∈ (map proj₁ flds) -- a is in the fields
    → a ∈• flds

getJsonObj : (js : JSON) → (isobj : JsonObj js) → List
 ↪  (String × JSON)
getJsonObj js (jsIsObj {js₁} {flds} x) = flds

dlookup : (a : String) → (flds : List (String × JSON)) → Dec
 ↪  (a ∈• flds)
dlookup a js with lAny.any? (Sₚ._≟_ a) (map proj₁ js)
... | yes inobj = yes (objhas inobj)
... | no ¬inobj = no λ {(objhas inobj) → ¬inobj inobj}
```

Listing C.5: FRP/JS/JSON.agda