UNIVERSITY OF TWENTE

Master's Thesis

Generating specifications for JSON APIs for formal and semantic use

Author:
Willem Siers [1]

Committee:
dr.ir. S.J.C. Joosten [2],[3]
ir. R.W. van Dalen [2],[4]
prof.dr.ir. A. Rensink [3]
dr.ir. M.E.T. Gerards [3]

Faculty of Electrical Engineering, Mathematics and Computer Science
Formal Methods And Tools
7500 AE Enschede, The Netherlands

August 18, 2019

[1]`w.h.siers@{student,alumnus}.utwente.nl`
[2]Supervisors: dr.ir. S.J.C. Joosten and ir. R.W. van Dalen
[3]University of Twente
[4]ING Group

Abstract

Computer programmers who create software benefit from the availability of API specifications, because they enable them to more reliably create correct software. This thesis presents our research on generating type specifications for JSON-based APIs in an automated manner. Such specifications describe the type of a JSON document in such a way that they are usable for formal applications such as static type checking. Additionally, we aimed to let the specifications convey the semantics of the API to a programmer. We noticed that despite the availability of API specification languages, many APIs do not have specifications. This may be because it is costly to create them. With our automated method to create such specifications we hope to alleviate the hurdle to create them, and thereby increasing adoption. The basis of our solution is type inference on JSON data that resides in responses to an API request. We present a minimisation algorithm to decrease the size of inferred types, and transform a purely formal specification into one that can be interpreted better by a programmer as well. It identifies recurring parts of a type and makes this explicit in the specification. It also has a heuristic-based method for identifying semantically equivalent types, which can be fused together, which is then also made explicit. Furthermore, an automated clustering algorithm is presented, which attempts to group JSON data by API operation. We conclude that our automated method can be of use for formal purposes, because generated specifications reliably detect typing errors in a program (which we call the completeness property), even though it should be taken as a serious consideration that absence of errors can not be proved (which we call the lack of the soundness property). The extent to which the generated specifications convey semantics of APIs varies, but the techniques we present in this thesis increase usefulness of specifications for programmers.

Contents

Chapter 1

Introduction

Traditionally, when a user ran a computer program, the software executed locally on their computer. In the current age, a lot of software functionality on one computer is provided externally over the internet, by another computer. A web service is such software, that offers functionality over the internet that is to be interacted with by other software. User-facing programs such as websites or smartphone applications are not web services, but the external software components they rely on to provide the functionality are. For example, these services provide functionality such as getting the weather forecast and posting a message on a forum.

A programmer uses web services by communicating with their APIs (Application Programming Interface). This is the interface to a web service, consisting of the *operations* the service supports, and the *data types* of the data these operations return. APIs enable re-use of software, because they allow multiple programs to make use of a single service. They can be used internally by an organisation, to communicate between several applications of its IT infrastructure, or they can be accessible to the public. Many popular web services, like Twitter, Instagram, and Wikipedia, have publicly available APIs.

In order to use an API, one needs to know its specification. An API specification is a technical document that specifies features of an API. In our case, we are interested in the different *operations* the API supports, and what the *data types* are of the data returned by these operations.

We are interested in APIs that uses data in the JSON (JavaScript Object Notation) format, which is a simple and widely-used data format that is human-readable and computer-readable . Broadly speaking, we call the structure of a JSON document its *type* (also called schema). Such types are captured in a specification, and the industry standard for stating such data type specifications is JSON Schema. JSON Schema is specification language, in which the type of (a corpus of) JSON documents can be described.

The current thesis presents a method to automatically generate type specifications for the *operations* in an API. These specifications are inferred from network traffic data logs, which is a log that contains what was sent over a network to a web service (*request*), together with the data that the web service sent back (*response*). To discover which operations exist we require the information in the *requests*, and to infer the data types that they return we require the *responses*. Through an example we now illustrate what our method does, after which we continue with the motivation for this research. Here we provide only the minimally required descriptions of some terminology, which we give details for in Chapter 2.

Consider the example traffic data in Figure 1.1, which is input for our method. This figure lists three distinct JSON responses of a web service that represents a coffee machine, supporting espressos and cappuccinos in cheap and in premium variations. The captions of the figures indicate the request that yielded this response. To understand these requests, it is sufficient to see that the *operation* `get=drink` yields the cheap version of a drink, `get=variations` yields the possible variations of a drink, and the *argument* `type=` indicates which drink we want.

The response in Figure 1.1a shows an *object* (indicated by braces {...}) with two properties: `name` with textual value `"Normal Espresso"` and `price` with numeric value `0.26`. Despite the responses in Figures 1.1a and 1.1b being different, they have the same type in our type system, because the objects have the same keys (`name` and `price`), and the types of their values are the same (textual and numeric respectively). Figure 1.1c shows an *array* (indicated by braces [...]), with three objects as elements. Because the three elements do not have the same type, we call the array heterogeneous.

Our algorithm infers a specification for these three requests, for which the JSON Schemas are shown in Figure 1.2. It consists of two types. Figure 1.2a states the type of the object in Figures 1.1a and 1.1b. For now, the most important aspects of the schema are `"type":"object"`, which indicates that the top-level type is an object, for which the value of `"properties"` states the names of the fields in that top-level object (`price` and `name`), which in turn state that their values are `numbers` and `strings` respectively. In the second schema, in Figure 1.2b, the values of `items` are the types of the elements of the array. Note that despite the array in Figure 1.1c being heterogeneous, there is only one *item*. The `anyOf` field is used to indicate that the `price` field has either numeric or string values.

One more thing to note is that we identify operations using *filters*. A filter of a schema specifies a condition on the *requests*, such that the schema is valid for only those requests that match the filter. For example, the filter `get=drink` (see the caption of Figure 1.2a) matches the requests of Figure 1.1a and 1.1b, but not of 1.1c.

In short, our method has the following properties:

- It identifies operations of an API, and their data types.
- It is an automated approach. That is, it does not require user-input.
- It generates correct specifications. That is, all data from which a specification is inferred is accepted by this specification.
- It aims to generate specifications that convey semantics of an API. That is, it conveys some meaning of the API to a human programmer.
- It works on heterogeneous data: the traffic log can be a mix of differently typed responses.

Our method computes type specifications from a set of JSON responses as follows. The first step is to compute types of each response separately. Then, these types are combined in various ways to create more general, yet still precise, types. Because the set of responses correspond to several operations (that is, the set is heterogeneous), the method analyses both the computed type of a response together with the corresponding request with the aim of identifying operations. The type specifications are then improved by computing types per operations instead of for the complete set of responses at once.

1.1 Motivation

The motivation of the current research is twofold. Firstly, having API specifications is useful. Secondly, availability of specifications is not as widespread as it should be, because they are time consuming to create. This section addresses these points.

```
{
    "name"  : "Normal Espresso",
    "price" : 0.26
}
```

(a) Request: get=drink&type=espresso

```
{
    "name"  : "Normal Cappuccino",
    "price" : 0.26
}
```

(b) Request: get=drink&type=cappuccino

```
[
    {
        "name"  : "Normal Espresso",
        "price" : 0.26
    },
    {
        "name"  : "Premium Espresso",
        "price" : 0.52
    },
    {
        "name"  : "Decaf Espresso",
        "price" : "Not available"
    }
]
```

(c) Request: get=variations&type=espresso

Figure 1.1: Three requests and their JSON responses

1.1.1 Why specifications?

In software engineering, software is often described using *informal* documentation[8]. These are documents consisting of explanations in natural language and usage-examples, intended for the programmer. Informal documentation of an API convey semantics of an API to the programmer. We define semantics of an API in the same manner as for programming languages, namely as how syntactic structures are to be interpreted by a human [3]. Although our method does not generate informal documentation but formal specifications, it is desirable that they convey semantics of an API as well as possible.

The problem with informal documentation is that they can be ambiguous, outdated and incomplete (or even non-existent). Learning how to use an API then becomes a trial-and-error process. Incorrect usage obviously leads to incorrect results and bugs.

A more precise, and useful, thing, is a *formal* specification. These are precise specifications written in, for example, mathematical notation or such that it can be interpreted by a computer program. These enable the use of formal methods[8], such as static (and dynamic) software analysis, which increases correctness of programs, and mitigates unexpected bugs and vulnerabilities. Tools such as type checkers, code completion, software generators, and data validators become available to programmers when a formal specification is available, which speeds up development by mitigating the trial-and-error aspect. The most important of these for software correctness is a type checker, which is a tool that says whether or not a piece of data in a program conforms to the expected type, and throws an error if it does not.

```
{
  "$id$" : "C.schema.json",
  "type" : "object",
  "properties" : {
    "price" :
        {"type" : "number"},
    "name" :
        {"type" : "string"}
    }
  },
  "required" : [ "price", "name"
      ]
}
```

(a) Filter: get=drink

```
{
  "$id$" : "M.schema.json",
  "type" : "array",
  "items" : [ {
    "$id$" : "P.schema.json",
    "type" : "object",
    "properties" : {
      "price" : {
        "$id$" : "N.schema.json",
        "anyOf" : [
          {"type" : "number"},
          {"type" : "string"}
        ]
      },
      "name" : {
        "type" : "string"
      }
    },
    "additionalProperties" : false,
    "required" : [ "price", "name" ]
  } ]
}
```

(b) Filter: get=variations

Figure 1.2: Two inferred types (with their corresponding filters)

1.1.2 Why automatically generate specifications?

There exist languages and tools for writing API specifications (which we describe later, in Section 2.4). This should increase how often specifications are available for APIs. According to a survey, adoption of OpenAPI specifications (an industry standard for API specifications) rose among respondents, from 25% in 2015, to 69% in 2019 [16]. It should however be noted that the respondents are users of their API tools, and therefore this might not represent absolute usage of specification languages.

Still, many APIs do not have specifications, which may be due to difficulty and costs associated with creating them. Creating a specification of existing software incurs high initial costs, because software systems are increasingly complex. Capturing behavior of software is far from trivial, and often involves reverse engineering. This is especially apparent in legacy systems, which are old but functional systems whose notoriously complex software is not fully understood by a single person. Additionally, just like other forms of documentation, specifications require maintenance. When the software changes, the specification may need to be updated to reflect this. This is another reason not to create APIs for existing systems, as it introduces another artifact that requires maintenance.

The current research provides a possible solution, by proposing automation of the creation of these specifications. By generating specifications, we mitigate the associated difficulties and costs. Whenever software is updated, its specification could be regenerated in a correct fashion. An additional advantage of generated specifications is that they may provide a well-defined link with the actual software, because even though a manually written specification can be correct, the thought process that led to this correct specification is not well-specified.

1.2 Outline

The current thesis is structured around three research questions. The first research question is treated in Chapters *Method* and *Related work* (Chapter 3 and 5). The second and third research questions are the concern of the chapter Evaluation (Chapter 4).

- **Research Question 1**: *How can API specifications be generated in an automated manner?*
  We answer this question by describing the method we developed for this goal, as this was the central point during our research. Also, we describe related research, techniques and tools that have similar goals.

- **Research Question 2**: *How useful are these automatically learned API specifications for formal purposes?*
  We want these specifications to be usable for formal methods, such as static analysis and type checking. To answer this question, we evaluate correctness of the specifications, by checking the specifications against the data it is learned from. Additionally, we check the specification against data it was not learned from, to assess whether or not our approach is useful if one has an incomplete traffic log (that is, it does not contain all possible responses of some API operation).

- **Research Question 3**: *How useful are these automatically learned API specifications for semantic purposes?*
  Because specifications are not merely consumed by computers, for formal purposes, it is also interesting to look at if they provide semantic value for a programmer. That is, they are a useful piece of documentation to a programmer. This question is answered through in part through subjective evaluation of the generated specifications, and also by evaluating statistics of the generated types, such as their sizes and (the number of) identified operations.

Chapter 2 describes concepts the reader needs to know to understand our research, some of which we have introduced in the current chapter. Chapter 3 describes how our method works and how we have approached the requirements of this method. Chapter 4 describes how we evaluate the implementation of our method, and discusses the results. It also describes the datasets we used and shows and discusses the results of the evaluation. Chapter 5 gives an overview of existing related work, to put our work into context. Finally, Chapter 6 concludes our research, answers our research questions, and describes future work.

Chapter 2

Background

This chapter describes background concepts for our research. It serves to clear up the terminology used throughout this document.

2.1 JSON

JSON[1] is a simple and widely used data-interchange format. It is commonly used for serialising data and sending it over a network, such as in many JavaScript based web applications. By serialising some data into this format it can be transferred from one piece of software to another.

Before we give the formal syntax of JSON, see Figure 2.1 for an example of a JSON object. It shows a JSON object that represents a person named John. An `object` is an unordered set of members. The object has three members, which are `name:value` pairs, starting with a { token, and ending with a } token. The first member has a string value `"John"`. The second member has an array value. `Arrays` are ordered collections of values, starting with a `[` token, and ending with a `]` token. Its first value is a number, its second value is another object. The third member has a boolean value.

Here we state a simplified version of the JSON syntax, and refer to ECMA-404 for a complete formal definition[7]. It is simplified by not addressing details such as whitespace and which characters make up a string or a number. Terminals are written in lowercase, and non-terminals start with a capital letter.

$$
\begin{aligned}
Json &:= Element \\
Elements &:= Element\ |Element\ ,\ Elements \\
Element &:= Value \\
Value &:= Object\ |Array\ |String\ |Number\ |true\ |false\ |null \\
Object &:= \{\ \}\ |\{\ Members\ \} \\
Members &:= Member\ |Member\ ,\ Members \\
Member &:= String\ :\ Element \\
Array &:= [\ ]\ |[\ Elements\ ]
\end{aligned}
$$

Where *String* is a sequence of Unicode characters that starts and ends with a quote, and a Number can be an integral number, decimal number, or a number in scientific notation.

[1]`http://json.org` (accessed 14th August 2019)

In this document, we call a JSON-document containing one `Element` an *instance*. An instance is what we use as training data to generate types, and as validation data to test the generated schemas.

Because objects and arrays allow nesting, graph like structures can be represented. However, because JSON does not support referencing other nodes, it does not allow cyclic graphs. Therefore, JSON object can be represented as trees. For this reason, we adopt terminology originating from trees, such as leafs, nodes, edges, root of the tree, et cetera, and apply it to JSON objects.

```
{
    "name": "John",
    "dates": [2019, { "year" : "1995", "month" : "July" } ],
    "alive": true
}
```

Figure 2.1: JSON object representing a person.

2.2 URL format

A Uniform Resource Locator (URL) is an address that points to some resource on the web. For our purpose, URLs are used to provide access to APIs. The most important thing to know is its basic syntax, which is the same as that of a Uniform Resource Identifier (URI).

A URL consists of a few components, as described in RFC3986 [4]. These are referred to as the *scheme*, *authority*, *path*, *query*, and *fragment*. The syntax can be given as:

$$\texttt{scheme}:[\texttt{//authority}]\texttt{path}[\texttt{?query}][\texttt{\#fragment}]$$

For example, the URL "`https://wikipedia.org/w/api.php?title=computer&action=search`" has the following components:

$$\begin{aligned} scheme &: https \\ authority &: wikipedia.org \\ path &: /w/api.php \\ query &: title = computer\&action = search \\ fragment &: (none) \end{aligned}$$

The path can be subdivided in path-segments: *w* and *api.php*. Furthermore, the query can be divided into parameters and arguments. In this example, *title* and *action* are parameters, and *computer* and *search* are arguments.

2.3 APIs

An Application Programming Interface (API) is the bridge between a human programmer and an implementation of some functionality [12]. Generally speaking, it defines the available functions and datatypes of a piece of software, which can be used to program an application. For example, Java has a library that contains code that implements Set, Map and List functionality, which a programmer can use through the Collections API. It lists datatypes and method, allowing a programmer can interact with the library without having to know how a Set is implemented. The current research focusses on APIs of web services.

In recent years, the software industry started noticing the importance of explicitly specifying APIs, and publishing them. Incentivised by gaining more users, companies now offer their service through publicly accessible APIs. This is accomplished because application programmers can now create applications based on the service. As an example, the online Twitter API spawned 3rd party Twitter applications, and websites that embed twitter functionality such as sharing a news article.

Usability and power (i.e. expressiveness) are the two basic qualities of an API [12]. When an API is hard to use, it costs more time to develop software, adoption rates will be lower and the resulting software often contains more bugs and security issues. Design methods can be used as guidelines, but usability evaluation of a created API yields better insight into the actual quality. Creating an API is a design task, and what constitutes a *good* one is subjective. For resources that discuss guidelines for creating a good API we refer to two documents discussing general API design principles, by Bloch [6] and Blanchette [5].

In the APIs that we have used to evaluate our current work, the URL schema, authority and path are constant throughout different requests to the API. The URL query changes between calls, and thus the query uniquely identifies a request. For different APIs, there are different, but fixed, sets of valid query parameters (we call them valid if they are defined by the API). The arguments to these parameters can be chosen freely. While this thesis assumes that API requests are identified only by their queries, many APIs have a different approach. For example, in the APIs based on the Representational State Transfer (REST) architecture, the URL path must also be used to identify requests[13]. Our related work, and OpenAPI (which we discuss later in this chapter) specifically target this style of API, but limit ourselves to APIs in which the path is irrelevant and only the query of the request is considered.

2.4 Specification languages

There exist standardised ways to represent JSON types and APIs, which are introduced now.

2.4.1 JSON Schema

JSON Schema was created as a standard for specifying the structure of JSON files[14]. A JSON Schema is a specification that describes the structure of some JSON data. A schema is itself is written in JSON, and has meta-information about another JSON object. The most current version is draft 7[2]. Use-cases for schemas are to implement type-checking for JSON objects and for documentation purposes, giving the developer insight into the structure of JSON data.

An example of a JSON Schema is shown in Figure 2.2, corresponding to the JSON object in figure 2.1. We explain the semantics of a relevant subset JSON Schema here. The work-in-progress draft is available to get details on the full language[3]. The lefthand-side terms we list below are keys in a JSON Schema with a special meaning, and the righthand-side describes what the value of this key represents.

[2]`https://json-schema.org/` (accessed 24th July 2019)

[3]`https://json-schema.org/latest/json-schema-core.html` and `https://json-schema.org/latest/json-schema-validation.html` (accessed 24th July 2019)

```
{
  "$id": "example.person",
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Person",
  "type": "object",
  "properties": {
    "name": {
      "type": "string",
      "description": "The person's first name."
    },
    "dates": {
        "type": "array",
        "description": "Important dates in this person's life.",
        "items": {
            "anyOf" : [ {
                "type" : "number",
                "minimum": 1900
            },
               {"$ref" : "example.birthdate"}
            ]
        }
    },
    "alive": {
        "description": "Is this person still alive?",
        "type": "example.status",
    }
  },
  "additionalProperties" : false,
  "required" : ["name", "alive"]
}
```

Figure 2.2: An example JSON Schema for the object in Figure 2.1. It specifies properties about each field in the JSON object.

`$id`	:=	A unique identifier used to reference the current (sub-)schema.
`$schema`	:=	Points to the meta-schema of the version of JSON Schema that is being used.
`$ref`	:=	Reference another schema that is defined elsewhere, using its $id. In our example, the `dates` array may contain values of type *example.birthdate* (not shown).
`title`	:=	An informal name for the schema.
`description`	:=	An informal description of the schema.
`type`	:=	The type of the value described by this schema. Either `null`, `boolean`, `object`, `array`, `number`, `integer` or `string`.
`items`	:=	The types of values in an array. If one type is given, all elements have that type. If an array of types is given, the type of an element is given by its position in the array.
`properties`	:=	The keys and their types of members in an object.
`required`	:=	Which members in `properties` are always present in an object. Non-required members still match the given type if it exists, but are ignored otherwise.
`additionalProperties`	:=	If true, the object may have members that are not specified in. `properties`. Otherwise, there are never more members than those in `properties`
`patternProperties`	:=	Like *properties*, except that all members whose key match the regular expression have the given type.
`anyOf`	:=	Lists one or more types the values may have.
`minimum/maximum`	:=	Specifies ranges for number types.
`minLength/maxLength`	:=	Specifies length ranges for string types.
`minItems/maxItems`	:=	Specifies ranges for the number values in an array.
`enum`	:=	Specifies which literal values a value may have. For example, to say that some date property only contain strings from the set "Januari", "Februari", ...

There are more constructs in JSON Schema, but this subset is sufficient to understand our research, and was sufficient to represent all types we generate in our type system.

2.4.2 Swagger and OpenAPI

Swagger[4], first created in 2011, is a toolkit for designing, documenting and testing REST APIs using JSON (and YAML) format. It defines a standard format for writing API definitions, and as a result it can offer tools for automatically generating documentation webpages, clients to call the API, and testing tools for experimenting on the API.

OpenAPI is the continuation of Swagger. In 2015, the Swagger Specification was renamed to OpenAPI specification. It is being maintained as an open source project by the OpenAPI initiative[5]. Swagger still provides the tools for APIs conforming to the Open API specification, but the specification itself is maintained by the Open API initiative. OpenAPI incorporate parts of JSON Schema in its definitions, namely the types in Swagger are defined using JSON Schema. On top of that, OpenAPI groups

[4]`https://swagger.io` (accessed 14th August 2019)
[5]`https://www.openapis.org/` (accessed 14th August 2019)

```
"/pets/{id}": {
      "get": {
        "description": "Returns a pet based on a single ID",
        "operationId": "findPetById",
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "id",
            "in": "path",
            "description": "ID of pet to fetch",
            "required": true,
            "type": "integer",
            "format": "int64"
          }
        ],
        "responses": {
          "200": {
            "description": "pet response",
            "schema": {
              "\$ref": "#/definitions/Pet"
            }
          },
          ...
      }
  }
}
```

Figure 2.3: A part of a openAPI specification, that defines an operation.

these operations by API operation, so that each operation indicates type of its response. An operations is defined per REST resource, that is a combination of the URL path, method (e.g. GET or POST), and its parameters (i.e. the URL query).

Figure 2.3 shows an example of an OpenAPI specification[6]. Starting at the top of the example, it shows the definition of an operation at the path */pets*, with a parameter *id*. So, for example, it matches the following request (with GET as method): `"http://example.org/pets/14"`. The `parameters` object defines the type of this parameter and its location (the `in` property). The `responses` object states that, if the statuscode is 200 (which is a code that means that the server handled the request without errors), the response has the type as described by the child object (which is written in JSON Schema). In this case, the response type is defined by an external schema `#/definitions/Pet`.

Just as with JSON Schema, there exists a wide variety of tooling for working with OpenAPI, such as converters between different specification formats, GUI specification editors, schema validators and data validators, parsers, documentation generators and SDK generators that generate a functioning client or a server stub that supports the operations that are in the specification[7].

[6]taken from `https://github.com/OAI/OpenAPI-Specification/blob/73b79bf7db9adb45a3d5d8076e614e64ab0f0897/examples/v2.0/json/petstore-simple.json` (accessed 14th August 2019)

[7]These tools are listed at `https://openapi.tools` (accessed 14th August 2019)

Chapter 3

Method

This chapter describes our method for generating specifications for JSON APIs, for which the motivation given in chapter 1. This chapter gives detailed descriptions of the parts it consists of, as well as an overview of how they relate to each other. We also discuss the requirements which led us to this method. The method is described in increasingly precise explanations. Section 3.1 starts by stating our requirements. Section 3.2 gives a high-level overview of our method. Section 3.3 gives the overview of our type system, and remaining sections 3.4 through 3.7 go into detail on the four parts of our method.

3.1 Requirements

This section states the approach we used to deal with the problem that was introduced in our Introduction chapter and certain sections of the Background. The global goal is that the generated specifications serve both formal usage and semantic usage. Our requirements bridge the gap between our motivation and our created method. The requirements for our method are expressed in terms of requirements on the specifications that it generates.

Requirement 1: The specification is complete.

We consider the method correct if creates specifications that are complete with respect to the input to the method. A specification is complete with respect to an input to our method (consisting of a set of JSON instances), if it flags as correct all these inputs. This means that a specification is inferred from a set of instances, accepts all instances in this set. For a type checker, completeness means that no correctly typed values are flagged as incorrectly typed (i.e., it gives no false negatives). Section 3.3 explains what it means for a value to be (in)correctly typed. We do not require the stronger notion of completeness, completeness with respect to an API, which means that it flags as correct all valid inputs for this API. Checking this requires knowledge of all valid inputs, but our datasets merely contain a subset of every valid input (incomplete dataset).

Note that we do not require the soundness property, which in the current context means that it flags only valid inputs to an API as valid. Soundness would imply that the specification could be used to prove typing-correctness of a program, that is, the absence of typing-errors. A type specification is sound if all inputs that it flags as valid, is in fact valid. Our method does not have this requirement.

Requirement 2: The specification conveys semantics.

It is desirable that the specifications is not only computer-interpretable, but also capture the human-interpretable meaning of elements of the API. Our method aims to achieve this is through identification of commonalities and variations in types, and through grouping types by their operations.

Requirement 3: The specification is small.

A small specification is less complex, and thereby easier to read for a programmer. This requirement does not mean that given two specifications, the smaller is always better. It only means that a small size, on an absolute scale, is preferable.

Requirement 4: The specification generalises over multiple instances.

A network traffic log is unlikely to contain every possible value an API may return. We call such a set incomplete. A specification that matches exactly such a set of instances conforms to the completeness requirement for that set, but fails to match any unseen instances. This is also known as over-fitting. We say that a specification generalises if it describes some class of values. Ideally, we want our specifications to generalise over values with the same semantics. Similarly to Requirement 3, a type that generalises more than another is not by definition better.

Requirement 5: The specification is specific.

A specification should match only a specific group of instances. Imagine that we would simplify our type specification to the universal type, that matches any instance. Such a specification is neither useful for type checking because it does not identify typing errors, nor is it useful for a programmer to learn anything about the API. This is also known as under-fitting. This requirement balances out Requirement 4.

Requirement 6: Minimal user input required.

Our problem statement is that it is costly to create specifications, and our solution is automation. Therefore, our method must not be guided by a user or require extensive configuration.

Requirement 7: Training data can be heterogeneous.

As addition to Requirement 6, we require minimal data preparation. This means that the method must handle an input of a mix of differently typed responses.

Requirement 8: Training data can be provided by network traffic.

Network traffic logs are relatively accessible if one has control over the API in question. We list this as a requirement to make it clear that we did not want to rely on data sources, such as the source code of the API.

3.2 Method Overview

Our method consists of multiple parts. This section walks through those parts, to explain what they do, and why they exist. Subsequent sections go into detail on each of these parts.

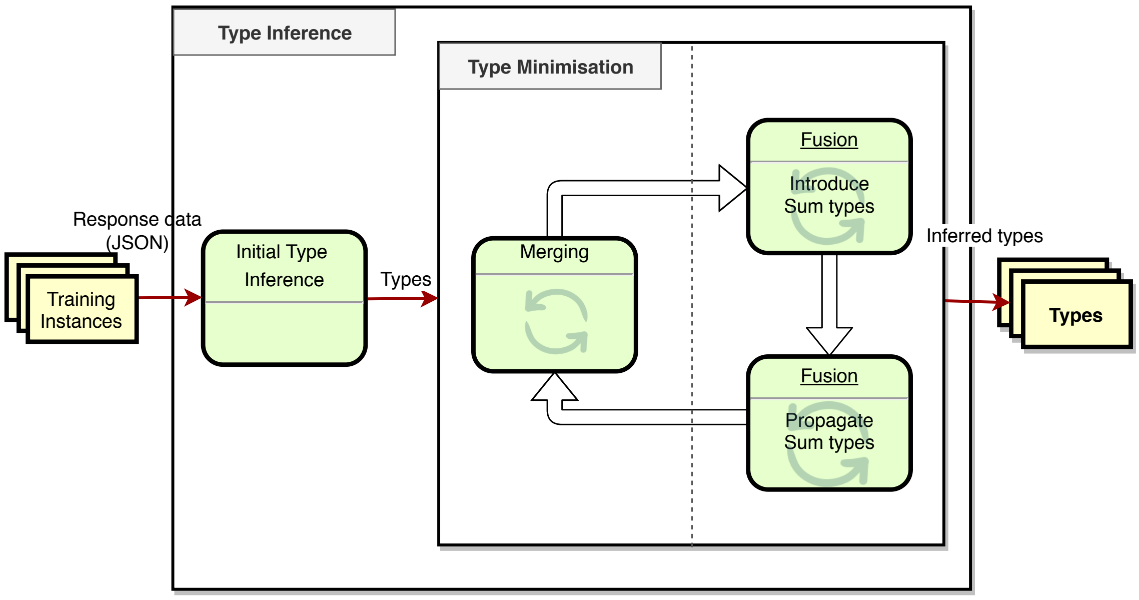


Figure 3.1: Overview of Type Inference

Figures 3.1 and 3.2 together show an overview of the architecture. The green boxes represent algorithms in our method, the yellow boxes represent data, and the red directed arrows indicate the flow of data.

Type Inference (Figure 3.1) takes as input a set of instances, and computes one or more types for it. It consists of two algorithms, *Initial Type Inference*, and *Type Minimisation*. The former creates one type per instance, and the latter processes these and reduces the number of types and their sizes. Type Minimisation consists of a *Merging* algorithm and a *Fusion* algorithm. *Merging* reduces the number of types and their sizes by merging (parts of) types that are equivalent. *Fusion* minimises types further, by introducing sum types based on heuristics, and propagating these through transformations on the types. In Type Minimisation, each of these two algorithms can introduce new work for another, and therefore they are called iteratively until a minimal type is returned. We explain Type Inference in Sections 3.4, 3.5 and 3.6.

Note that Type Inference normally yields one or more types, but we can force it to create a single type that describes all instances. From a set of types, we can create a single type by moving them into a Sum type, which is the supertype of all those types (details on this in the following section). This is done after Initial Type Inference and before minimisation, so that the single type is minimised. This has the advantage that all types are fused into one minimal type, but has the disadvantage that, for semantically unrelated inputs, it creates an equally semantically meaningless type. We refer to *Single Type Inference* when such a single type is created, and to *Multi Type Inference* otherwise.

Clustering (Figure 3.2) generates specific types for specific groups of instances. This allows our method to find better types for heterogeneous data. It also allows us to specify a relation between the request and the type of the response of an API call. *Filter Candidate Selection* selects potential Filters from the request data, which are used to split a cluster in two non-overlapping clusters. Then, similarity of its types is calculated, and new clusters are only accepted if the types of the two are dissimilar enough. This rests on the assumption that requests to different functions in an API yield significantly differently typed responses, and requests to the same function but with different arguments, yield relatively similarly typed responses. Clustering is detailed in Section 3.7.

All parts of the method contribute to fulfilling the requirements from Section 3.1. Some parts however, play a larger role for some requirement than others. Requirement

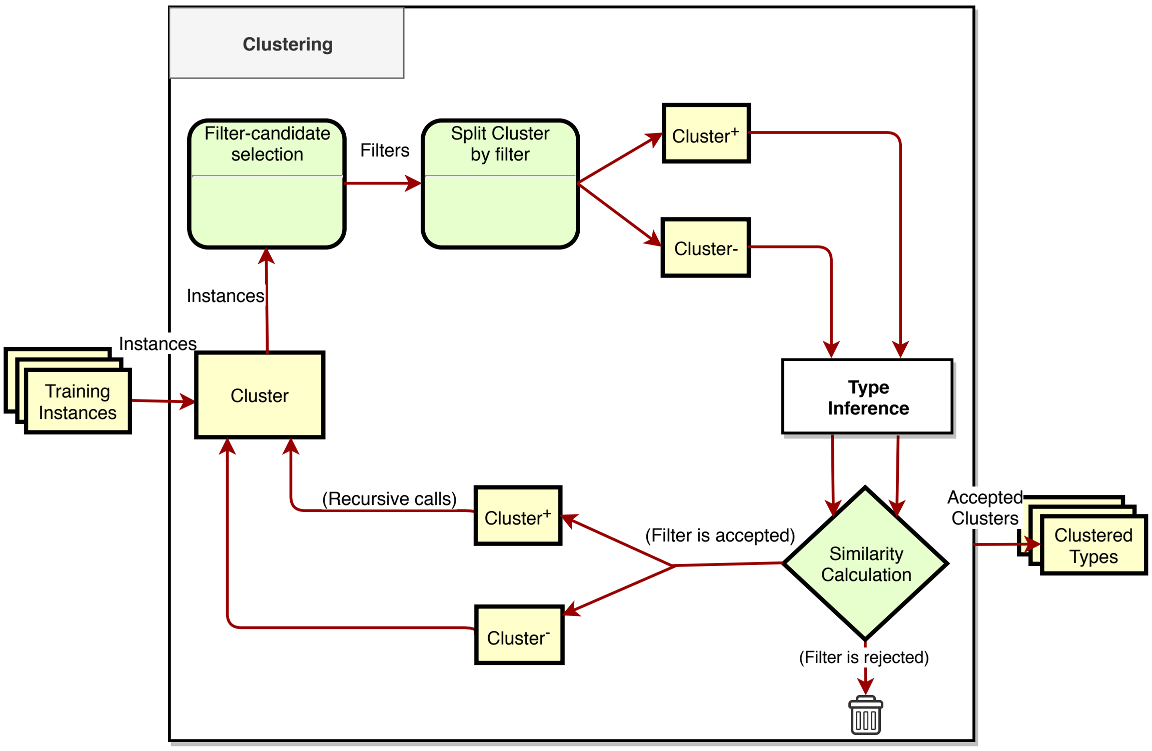


Figure 3.2: Overview of Clustering

1 (completeness) is a hard requirement that is preserved throughout all algorithm. Requirement 6 (user-input) is also generally preserved at runtime, although tweaking can be done, such as configuring the heuristics of the fusion algorithm. For Requirement 2 (semantics), clustering has the biggest effect on unveiling semantics, and minimisation contributes also, albeit less obviously. For Requirement 3 (size), decreasing sizes of types is the largest concern of minimisation, but also clustering. Requirement 4 (generalisation) is achieved in part by a general type system (upcoming Section 3.3, but increased further by fusion. Requirement 5 (specificity) is mostly the concern of clustering, just like requirements 7 (heterogeneity), as it creates specific data groups from the heterogeneous data.

3.3 Type System

This section defines the syntax and semantics of our type system, and give other definitions that are useful for later sections. It serves as a reference for the reader, which is why it already names terminology that we explain later in this chapter.

3.3.1 Definition

The syntax below describes types in our type system, which is used to define our algorithms. Below that, a description and the semantics of the type system are given.

Terminals are written in lowercase, and non-terminals start with a capital letter.

$$\begin{align}
Root &:= Type \tag{3.1}\\
Type &:= Primitive \mid Collection \mid Clustering \tag{3.2}\\
Primitive &:= number \mid text \mid boolean \mid null \tag{3.3}\\
Collection &:= Record \mid Array \mid Sum \mid Map \tag{3.4}\\
Label &:= Key \mid Optionalkey \tag{3.5}\\
Record &:= \{Label_1 : Type_1, ..., Label_n : Type_n\} \tag{3.6}\\
Array &:= [Type] \tag{3.7}\\
Sum &:= (Type_1 + ... + Type_n) \tag{3.8}\\
Map &:= \ \langle Pattern : Type \rangle \tag{3.9}\\
Pattern &:= P(Regex) \tag{3.10}\\
Clustering &:= [Filter_1 \rightarrow Type_1, ..., Filter_n \rightarrow Type_n] \tag{3.11}\\
Filter &:= F(Condition_1 \& ... \& Condition_n) \tag{3.12}\\
Condition &:= String_k : String_v \mid String_k : * \mid !Condition \tag{3.13}
\end{align}$$

The type system can be used to form directed acyclic graphs, similarly to how JSON objects form trees (Section 2.1). The nodes in this graph are formed by the terms in the syntax. Every *Type* is either a Primitive, Collection or a Clustering. *Primitive*s are the simple Types, number, text, boolean and null, which always form leafs in the graph. *Collection*s are Types that can contain child types, and thereby form nodes in the graph. A Collection is either a Record, an Array, a Sum type or a Map type. Broadly speaking, these types relate to JSON as follows: *Record*s and *Map* types describe Objects, *Array* types describe Arrays and *Sum* types can describe a mix of Values. A *Record* contains associations of the form *Label* : $Type_c$, where $Type_c$ is the child of the Record. The *Label* is a string, which corresponds to a String in a JSON Object that is associated with some Value. When a Label is a *Key*, the Label is required to be present in an Object, whereas if it is a Optionalkey is not required that it is present. A *Map* is similar to a Record, but instead of specifying all Labels, a single regular expression (specified by *Pattern*) is used that matches all Strings in an Object. It can be seen as mapping from strings to some child Type, where all these strings match the Pattern. A *Sum* type represents a supertype of its children, that is, it is the type of all Values whose type is given by any of the children of the Sum type. A *Clustering* is a conditional type, which we use to specify that the type of some JSON Value depends on which condition on the Value holds. These *Condition*s are expressed in a *Filter*, for which at this point it is sufficient to know that it expresses some boolean condition on a Value. We also define *Root* as a Type without parents, which therefore only occurs at the root of the graph.

We express semantics of the type system more precisely by the definition of a type checking function, called `check`. This function takes as arguments a *Type* and a *Value*, and returns true if and only if the Value has the given Type, and returns false otherwise. If *check* returns true for some Type T and some Value V, then we say that T is the type of V. Value has been defined by the JSON grammar in Section 2.1.

Let `T` be some Type, and we let `V` be some Value, then the function *check* returns true if any of the following conditions hold:

- `T` is number and `V` is a Number.
- `T` is text and `V` is a String.
- `T` is boolean and `V` is true or false.

- `T` is null and `V` is null.
- `T` is a Record $\{Label_1 : Type_1, ..., Label_n : Type_n\}$ and `V` is an Object, such that for each $Label_i : Type_i$ in `T`:
  - if $Label_i$ is a Key, then there exists a Member in `V`, $s : v$, such that s matches $Label_i$, and the type of v is $Type_i$.
  - if $Label_i$ is a Optionalkey, then there need not exist such a Member $s : v$ in `V` such that s matches $Label_i$, but if it does, then the type of v is $Type_i$.
- `T` is an Array $[Type_c]$ and the `V` is an Array, such that all children in `V` have type $Type_c$.
- `T` is a Sum $(Type_1 + ... + Type_n)$ and `V` is some Value, such that at least one of the children of `T` is the type of `V`.
- `T` is a Map $\langle Pattern : Type_c\rangle$ and `V` is an Object, such that for each Member $s : v$ in `V`, s matches the Pattern and v has type $Type_c$.
- `T` is a Clustering $[Filter_1 \rightarrow Type_1, ..., Filter_n \rightarrow Type_n]$ and `V` is some Value, such that there exists one $Filter_i \rightarrow Type_i$ in `T` such that the request that yielded `V` matches all Conditions of $Filter_i$ and the type of `V` is $Type_i$.

In this thesis we give examples of types by a visualisation of the graph it forms, Such as Figure 3.3 in the next section. Types are visualised such that each node has the form `name : Type`, where the name is a (arbitrary) one-letter identifier for that node, and the Type is one from the syntax. Collection types can have outgoing edges, which signals a parent-child relation. For Records, the Label that points to a child is written on the outgoing edge to that child. If a Label is a Optionalkey, it is written with a question mark `?` at the end. For Maps, the Pattern is written on the outgoing edge to the child. This is a regular expression that starts with the token `^` and ends with the token `$`. Arrays have only one child, and Primitives can only form leafs in the graph. Clusterings are the only Types that we do not visualise in this manner.

3.4 Initial type inference

Initial Type Inference is the first step in type inference, which infers simple types for JSON instances. It generates exactly one root type for each instance it sees. Figure 3.3 already shows an example of a JSON document and its inferred Type.

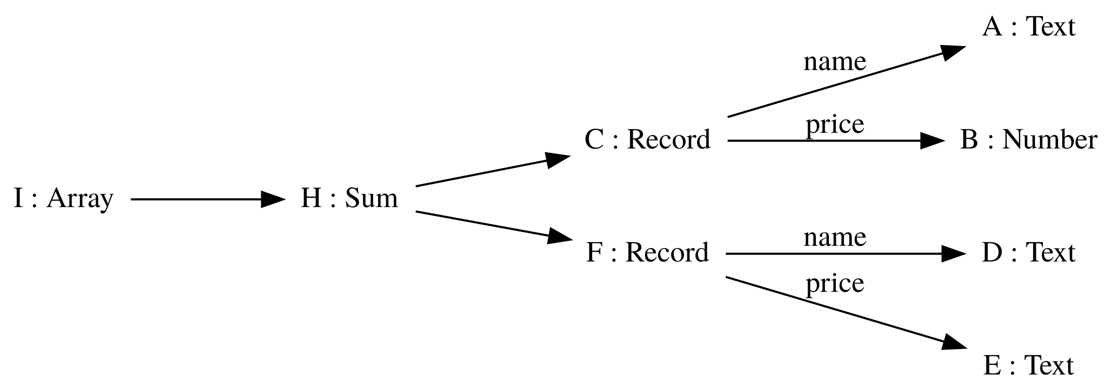
We first state the format of the input to this algorithm. Then, we state how types are inferred from this input.

3.4.1 Instance data model

An *instance* is a data structure that encapsulates one request to an API and its response. A set of instances is used as input for the inference algorithm. The request is encoded as a list of parameter-argument pairs from the query component of a URL, and the response is a JSON Value. The request part of the instance is only used by the clustering algorithm (Section 3.7), not by the type inference algorithms. While the current research only uses the query for the request field, this format could encode other request data, such as path segments for example.

```
[
    {
        "name"  : "Normal Espresso",
        "price" : 0.26
    },
    {
        "name"  : "Decaf Espresso",
        "price" : "Not available"
    }
]
```

(a) Example JSON



(b) Inferred type by *Initial Type Inference*

Figure 3.3: Initial type inference example

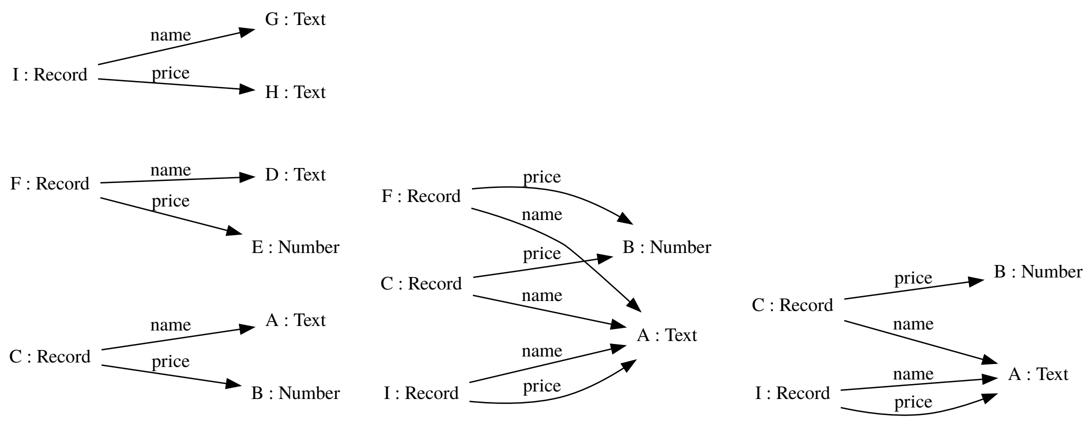
3.4.2 Conversion

Initial Type Inference converts the response field of one instance to one Type in our type system. It traverses the JSON tree and generates a tree in the internal representation of our type system. Each visited JSON Value corresponds to one term of our syntax. We describe the conversion as a function called *initial*. It takes as argument a *Value*, and returns a Root type. If the *Value* is a Number, String, true or false, or null, it returns respectively a number, text, boolean and null Type. If the *Value* is an Object, a Record is returned with equivalent Keys (so that all members are required), and the children are recursively computed by *initial*. If the *Value* is an Array, an Array is returned whose child type is a Sum type, whose children are computed by applying *initial* on the elements of the Array Value. The Root type that it returns is a tree, which is the type of the provided JSON tree.

Note that the only Primitives, Records with required keys, Arrays and Sum types are generated.

3.5 Type Minimisation: Merging

The *merging* algorithm is the first part of type minimisation. One observation is that results from initial type inference contain a lot of redundancy. *Merging* removes redundancy in a set of types, which exists both within root types, as well as between root types. Because the decision to merge Types is based on information of the subtree at a Type, we say that merging is done based on synthesized attributes[1]. As a result, the algorithm first merges redundant leafs, then redundant subgraphs, and finally redundant Root types. The last one only happens if the graphs formed by two root types are completely equivalent. The former two happen when only parts the graph formed by a root type are equivalent. This enables sharing of subgraphs of Types, that is, a node may now have multiple parents, which indicates that some data structure is reused. See Figure 3.4 for an example, where three Root types are merged into two. Initially, Record nodes (I, F and C) are not merged because none of them are equivalent due to not having the same children. After merging the leaf nodes, leaving only A and B for the Primitives, Records C and F have the same labels pointing to the same children, and are thus merged.



(a) Step 0: 9 Types from Initial Type Inference. (b) Step 1: equivalent leafs (G, H and D) and (E and B) are merged. (c) Step 2 : equivalent records (F and C) are merged.

Figure 3.4: Merging example

3.5.1 Algorithm

The merging algorithm removes redundancy in a Type in an iterative manner. We define it as the function *merge* which takes as argument a set of Types and returns another set of Types.

The way it works can be split into two steps. The first step is to identify which types are equivalent. This is done by defining an equivalence relation that groups equivalent types together. A set of equivalent types is called the equivalence class of those types. From each equivalence class we preserve only one element (picked arbitrarily from the class), and remove the others. Because the types that are to be removed may have parents, edges incoming to these types are moved to point to the preserved Type in the equivalence class. The second step is to remove all the redundant Types, and move their incoming edges.

A single application of these two steps removes one or more equivalent Types from the input (assuming there are equivalent types). When equivalent types are removed, it can simultaneously “create” new equivalent Types, because of the way the equivalence relation is chosen. Therefore, removing all redundancy requires iterative application of the two steps. Iteration halts when there are no more equivalent types (that is, all non-empty equivalence classes have size 1).

Which equivalence classes are formed is determined by the following equivalence relation, which mostly follows the Syntax for Types. Firstly, all numbers are equivalent, that is, *(number, number)* is in the relation. Similarly for *text*, *boolean* and *null*. *Record*s are in the relation if the sets of Labels they have are equal (also considering whether they are required or not), and for each of these Labels, the child it points to is equal between both Records. Two *Array*s are in the relation if and only if both have equal children. *Sum*

$$\begin{tikzcd}
 & & \text{A : Text} & \\
\text{I : Array} \arrow[r] & \text{L : Record} \arrow[ur, "\text{name}"] \arrow[dr, "\text{price}"] & & \text{B : Number} \\
 & & \text{J : Sum} \arrow[ur] \arrow[dr] & \\
 & & & \text{E : Text}
\end{tikzcd}$$

Figure 3.5: Fusion applied to the Type of Figure 3.3

types are in the relation if and only if both have an equal set of children (disregarding order). *Map* types are in the relation if and only if the Patterns of both are equal, and their child Types are equal. Any other types are not in the relation.

3.6 Type Minimisation: Fusion

The fusion algorithm supplements the merging algorithm. It achieves two things: First of, it makes Types that contain Sum types more precise by propagating these Sum types downwards. Secondly, syntactically different types which appear to represent the same semantics are fused together by introducing a Sum type for these types, yielding a more general type.

Consider the previous example in Figure 3.3. The two Records (`C` and `F`) represent the same thing semantically (a coffee entry), yet they are two distinct types. Because they are part of a Sum type already suggests they are the same, but the Type is verbose and can be more precise. The Sum type can be propagated downwards, yielding the Type in Figure 3.5. The two Record `C` and `F` are fused into Record `L`, and it is now explicit that the Type of the value at `name` is *Text*, and `price` is either a *Number* or *Text*. Furthermore, the Sum type `I` is removed, because it would only have one entry (`L`).

Sum types are propagated towards its children, whereby inherited attributes are imposed upon the subgraphs of Sum types. Note that determining semantically equivalent types relies on assumptions, and whether or not these assumptions hold differs between APIs. The algorithm captures these assumptions using heuristics, which may be adapted for different APIs.

3.6.1 Heuristics

The algorithm considers a number of heuristics to decide if Types will be fused. A heuristic together with a (user-provided) threshold forms a condition, and when this condition is met, a Sum type is introduced. First, consider the following heuristics:

1. # of keys in a Record
2. # of optional keys in a Record
3. $\frac{\textit{\# of optional keys}}{\textit{\# of keys}}$
4. *children are equal types (boolean)*
5. regularity of the labels (allow specific patterns or broad patterns)

The first heuristic counts the number of members in a Record type. Some instances contain objects with a very large number of members, which yield equally large Record types. The question is whether these Records properly convey the semantics of the object. It seems that if a Record is very large, then neither each individual Label nor its associated value are semantically relevant. These Records instead represent a Map type.

For example, we encountered objects with language-codes as keys and a country name as value. This represents a map from a language-code to country names. Apart from conveying semantics more concisely, a Map type may also form a more proper generalisation, as it does not become incorrect if the set of object were to be extended or shrunk (for example, if more languages are added).

The number of optional keys and the ratio of optional keys are a measure for how specific a type is. If there are several instances with a language object with different subsets of languages, then most members in the Record are optional.

The equality of children is true if all children of a Record point to the same Type node, and therefore depends on the result of the merging algorithm. This is used to form an extra condition on Records that are turned into Map types, namely that the children are equal. This is currently a boolean heuristic, but a numeric value expressing similarity of children would be an interesting heuristics for future work.

The regularity of labels relates to the size of the language that captures all labels. In some cases, a transformation is only sensible if all labels match a specific pattern, for example a date like *dd/mm/yyyy*, or a language code consisting of exactly three letters. In other cases, labels are very irregular but a using a broad pattern is still sensible because all children are semantically equivalent, for example when article titles are used as keys, and the values are objects that represent articles.

Our implementation lists two conditions that allow a Record to be transformed into a Map if any of them is met. The first condition is that a Record has at least 5 keys in a Record and all children are the same Type (that is, there is only one child), and the Pattern must be specific (for example, dates are allowed, but generic patterns such as "any sequence of characters" are not). The second condition is that a Record has at least 12 optional Labels, the fraction of optional Labels must be at least 0.9, the children need not be of equal types, and broad Patterns are allowed. Which heuristics and conditions are used are parameters to the algorithm (and somewhat subjective), and by inspecting the network traffic we found these two conditions capture (mis)usage-patterns of JSON.

3.6.2 Introducing Map types

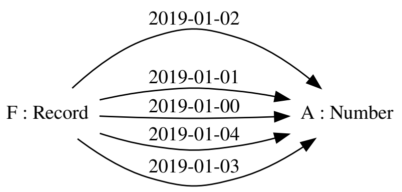
Map types are introduced when a condition is met. We show how a Sum type is introduced for children of a Record. Note that creation of a Sum type with merely one element is always disallowed, as to avoid an infinite loop of creating, followed by removing single-element Sum types. Two examples are shown in Figure 3.6 and Figure 3.7.

As the first step, a Pattern is inferred from the labels of a Record. This is done by selecting the first Pattern from a pre-defined ordered list of Patterns that matches all Labels. By ordering this list from specific to broad, it prefers picking specific patterns over broad patterns. Then, a Sum type is created of all children of the Record. Finally, a Map type is created to replace the Record, using the inferred pattern as label and the Sum type as child.

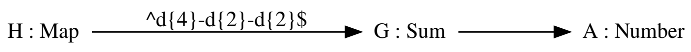
As mentioned earlier, this only happens if certain conditions are met. If, for example, the number of keys does not exceed the threshold, or a pattern can not be inferred from the Labels, then the transformation is not performed.

3.6.3 Propagation

We define a number of transformations that propagate Sum types. The Sum types are propagated in a top-down fashion. Considering that a Sum type is a supertype of each

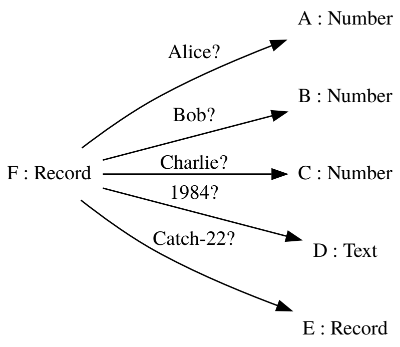


(a) Input

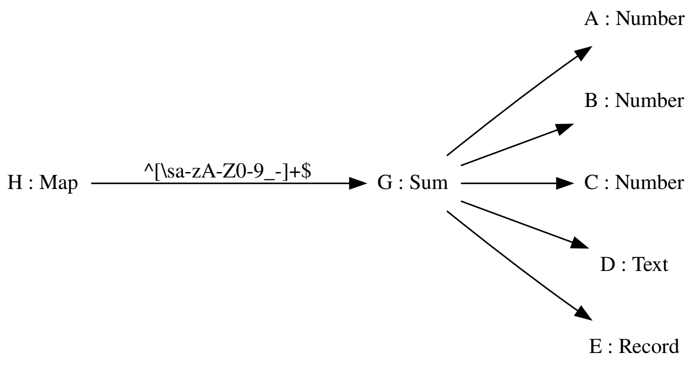


(b) Output

Figure 3.6: Example of introducing a Map type and Sum type



(a) Input



(b) Output

Figure 3.7: Example of introducing a Map type and Sum type

of its children, the overall effect of these transformations is that large sub-types are transformed into smaller sub-types.

Propagation: general

A Sum type with a single child Type is replaced with its child Type. That is, the Sum type `(T)` is replaced with `T`.

Children of nested Sum types are moved to the parent Sum type. That is, a Sum type of the form $((T_1 + ... + T_n) + T_{n+1} + ... + T_m)$ is replaced with $(T_1 + ... + T_m)$.

Propagation: Primitives

Primitive types are simply merged, in the same way as in the merging algorithm of Section 3.5. That is, a Sum type of the form $(Primitive_1 + Primitive_2 + \cdots)$ with equivalent Primitive types (e.g. both are Number) is replaced with $(Primitive_1 + \cdots)$.

Propagation: Arrays

The Sum type of two Arrays becomes an Array that holds the children of both. That is, a Sum type of the form $([Type_1] + [Type_2] + \cdots)$ is replaced with $([(Type_1 + Type_2)] + \cdots)$. See for example Figure 3.8 which shows fusion of two Arrays into a single Array.

Propagation: Maps

A Sum type of Map types becomes a Map type, with a new Pattern and as child a new Sum type. The Pattern is recomputing such that it matches all labels that match with either or both of the original two Patterns. The new child type is the Sum type of the values of both Map types. That is, a Sum type of the form $(\langle Pattern_1 : Type_1 \rangle +$

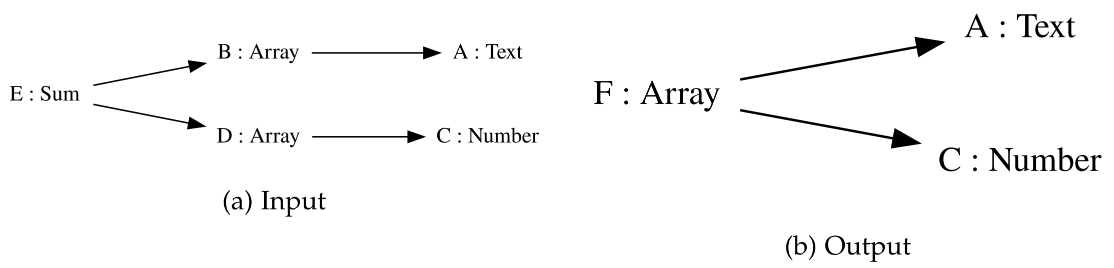


Figure 3.8: Example of fusion of Arrays

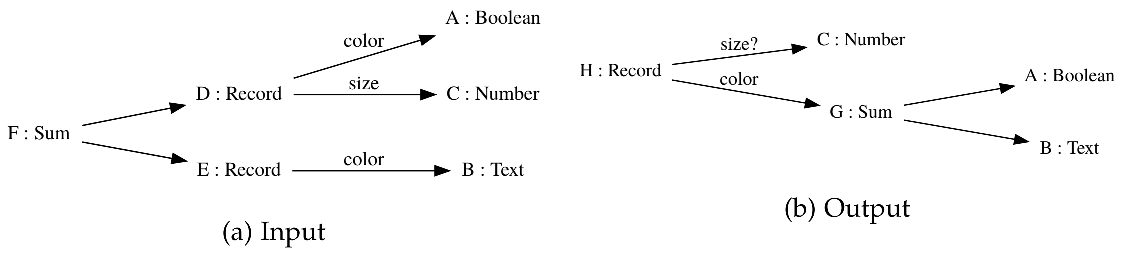


Figure 3.9: Example of fusion of two Records

$\langle Pattern_2 : Type_2 \rangle + \cdots)$ where $Pattern_1$ and $Pattern_2$ were respectively inferred from the set of Labels L_1 and L_2, is replaced with $(\langle Pattern_3 : (Type_1 + Type_2) \rangle + \cdots)$, where $Pattern_3$ is inferred from both sets of Labels, $L_1 \cup L_2$.

Propagation: Records

For a Sum type of two Records becomes one Record with members of both. Children of the new Record are formed by creating Sum types of the child types of both Records. The Label wil be required if it is present in both Record, and it is required in both Records. That is, a Sum type of the form $(Record_1 + Record_2 + \cdots)$ is replaced with $(Record_3 + \cdots)$ where the set of Labels in $Record_3$ is the union of those in $Record_1$ and $Record_2$, which are required if and only if they occur in both, and the child at a Label is either the Sum of the children at the corresponding Labels in $Record_1$ and $Record_2$, or, if only one of the two Records has the Label, the child from that Record. Figure 3.9 shows an example of this transformation.

3.7 Clustering

We apply a clustering algorithm to group related instances together, as to improve inferred types. A *cluster* is a subset of inter-related instances, and a *Clustering* is set of (non-overlapping) clusters. Note that clustering can also be described as finding a partition on a set of types. It makes types smaller, specific for a particular set of instances, and it relates the *request* of an instance with the *response*. An overview has been depicted in Figure 3.2.

As an example, consider the three instances in Figure 3.10 which are used to create a Clustering. These first two requests have in common that both have the `drink` argument to the `get` parameter, while the third request has a different argument to this parameter. Notice that both responses of the first two requests have different types from the third request, namely the top-level Type is a Record in one case and an Array in the other. A Clustering is then created consisting of two Clusters: instances whose requests contain the parameter-argument pair `get=drink` and the subset whose requests do not. Note that while the first and third request also have a commonality, `type=espresso`, but the types of the responses show less similarity. In this case no more Clusters are made because splitting either of these Clusters further does not yield a (sufficiently) different Type compared to if they are not split further.

```
{
    "name"  : "Normal Espresso",
    "price" : 0.26
}
```

(a) Request: get=drink&type=espresso

```
{
    "name"  : "Normal Cappuccino",
    "price" : 0.26
}
```

(b) Request: get=drink&type=cappuccino

```
[
    {
        "name"  : "Normal Espresso",
        "price" : 0.26
    },
    {
        "name"  : "Premium Espresso",
        "price" : 0.52
    },
    {
        "name"  : "Decaf Espresso",
        "price" : "Not available"
    }
]
```

(c) Request: get=variations&type=espresso

Figure 3.10: Three requests and their JSON responses

The issue that clustering deals with is that the data we work with is heterogeneous, because we do not manually segment the traffic logs. Then, the type computed from a heterogeneous set of instances, using Single Type Inference, covers a wide range of semantic concepts. Another issue is that, when semantically unrelated instances are fused together, the assumptions that the heuristics of the fusion algorithm are based on do no longer hold. Compared to a type from Single Type Inference, the number of root types increases from 1 in total to 1 per cluster, the overall size of the type thereby increases, but each individual sub type is smaller. A Cluster prescribes *Conditions* that hold for all instances that it contains. (These conditions are not pre-defined, but determined as we go, which is called refinement of the clusters.) Identifying suitable conditions, which in our case is identifying which API operation an instance belongs to, is the largest challenge for clustering.

The main goal of this section is the explanation of clustering refinement (Section 3.7.4), but that first requires explaining three things, in the upcoming three sections. Firstly, that we consider clustering successful when it is able to identify API operations. Secondly, how similarity of two types is calculated, which we use to determine which types should be clustered together. Thirdly, the clusters must be based on certain keywords that must be present in the query of the request (called Conditions) that yielded the types of the responses, and we use a simple strategy to select potentially good conditions. After that, we explain how *refinement* works, which is the act of turning a Clustering into a more precise Clustering.

3.7.1 Clustering by Operation

For our research, ideally one cluster is formed by instances of one operation of a given API. It then only contains instances of a single operation. We define an operation as a unit of functionality that, from the programmers perspective, does one single thing, but can be applied to different (sets of) arguments. In our previous example, we can say that one operation is to get details on one type of drink, and the other is to list all variations of a type of drink. Parameter-argument pair `get=drink` is that first operation, and `type=espresso` and `type=cappuccino` are arguments to this operation. From a programmer's perspective, an API specification that makes this explicit is semantically meaningful, because one is typically interested in the argument-types and return-types for a given operation. (Additionally, this allows these clustered types to be encoded in the openAPI format, which also segments an API into operations.)

Unfortunately, which part of a request indicates what the operation is that the request represents, is ambiguous. For example, consider the query `?operation=search & title=hamlet`. Through human interpretation we may identify that `operation=search` is the operation, with the argument `title=hamlet`. There is however no standardisation in calling conventions, which makes it hard to identify this algorithmically. As a result, clustering becomes a search algorithm over all possible partitions of the set of instances. Some parts of a URL that may identify an operation are, among others: the path, the existence of a parameter, a specific argument to such a parameter, the presence of a URL parameters. Combination are also possible, such that the combination of two parameters together identify an operation.

We rely on the following assumption to deal with this unknown. We assume that all queries on a given operation return similarly typed responses regardless of the provided arguments, and that queries on a different operation returns significantly differently typed responses. With this idea we differentiate operations from arguments as follows: All instances of one operation have similar types, and the different arguments in these instances merely result in variations of the same type.

3.7.2 Similarity metric

Similarity of two Types is calculated through a similarity metric. We created a metric that depends on our *Type Inference* algorithm and compares the sizes of inferred types. It is used in the refinement step (Section 3.7.4) to assess how to refine a clustering. This section only describes how the metric is computed, and later how it is used to refine clusterings. The similarity of two types is zero for two completely dissimilar types, and it is maximal (we chose 1.0) when two types are exactly the same.

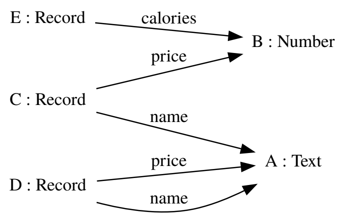
For this section, we say that the size of a Type is the number of nodes, excluding Sum nodes, in the graph *that does not share nodes*. With this we mean that nodes that are shared, will not just be counted once, but counted multiple times. This is the same as making a summation, over every node, of the number of paths from the root to that node. Sum nodes are excluded because they do not represent nodes in JSON instances. We denoted size with vertical bars `|`, such as $|Type_1|$. For example, in Figure 3.11b, $|I| = 4$.

Now, let $Type_1$ and $Type_2$ be two minimised types, and $Type_{1+2}$ the results from applying the Minimisation algorithm on the Sum type $(Type_1 + Type_2)$ (that is, it is the minimised supertype of the two). We can determine the range of the size that this type may have. We know that when $Type_1$ and $Type_2$ have high similarity, minimisation removes more redundancy in $Type_{1+2}$ than when they are dissimilar. The extreme case is when $Type_1$ is a subtype of $Type_2$, then $Type_{1+2}$ equals $Type_2$, because fusing a supertype with its subtype yields the same supertype. Therefore, $Type_{1+2}$ can be no simpler than the most complex of the two, and thus $max(|Type_1|, |Type_2|) \leq |Type_{1+2}|$. On the other hand, if $Type_1$ and $Type_2$ are completely dissimilar, they can not be fused, and $Type_{1+2}$ contains the full graphs of both types. Thus, $Type_{1+2}| \leq |Type_1| + |Type_2|$. The *growth* of the fused type relative to its lower bound is $|Type_{1+2}| - max(|Type_1|, |Type_2|)$. Because $|Type_{1+2}|$ can never exceed $|Type_1| + |Type_2|$, and $max(|Type_1|, |Type_2|)$ is the minimum size, the growth can never be more than the potential *growth* $min(|Type_1|, |Type_2|)$. Dissimilarity is then calculated by dividing the actual growth by the potential growth, that is:

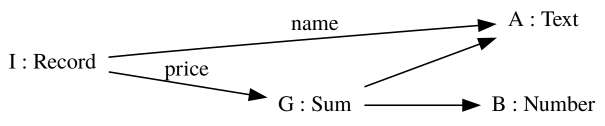
$$dissimilarity = \frac{(|Type_{1+2}| - max(|Type_1|, |Type_2|))}{min(|Type_1|, |Type_2|)}$$

Intuitively speaking, this measures the growth that occurs if two types were fused together. We define *similarity* as 1 minus the dissimilarity.

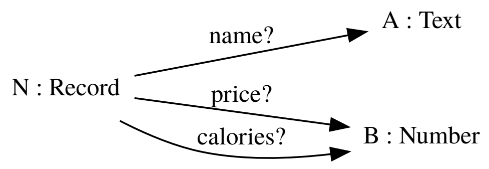
For an example, see Figure 3.11. In Figure 3.11a, the Types labelled `C` and `D` share more similarity than `E` and `D`. Figure 3.11b shows the fused type of `C` and `D`, and Figure 3.11c shows the same for `E` and `D`. Type `E` has size 2, `C` and `D` have size 3. Type `I` has size 4, and `N` has size 4. Then, for the two similar types we have $dissimilarity_{CD} = \frac{4-3}{3} = 0.33$, and for the dissimilar types we have $dissimilarity_{ED} = \frac{4-3}{2} = 0.5$. Note that the goal is to maximise similarity within a cluster, but to minimise similarity between clusters.



(a) Input



(b) Output similar types



(c) Output dissimilar types

Figure 3.11: Similarity Example

3.7.3 Filters and candidate selection

The syntax for a *Filter* indicates that it is a conjunction of a number of *Conditions*. A Filter is evaluated with an Instance, and we say that an instance matches a Filter if the request of the instance matches all Conditions of the Filter. A request matches a Condition if using the following rules. If the condition is of the form $String_{k1} : String_{v1}$, then it matches any request whose query contains $String_{k1} = String_{v1}$. If the condition is of the form $String_{k1} : *$, then it matches any request whose query contains $String_{k1}$ or, for some arbitrary other string, $String_{k1} : string_{v2}$. If the condition is of the form $!Condition_1$, then it matches any request for which $Condition_1$ does not hold.

Examples of a filter that evaluates to true for all coffee machine examples (Figure 3.10) is `Filter(get:* & type:*)` and one that only evaluates to true for two of the examples is `Filter(!get:variations & get:drink)`.

A complete search for a clustering would entail computing every possible clustering. Instead of that, clusterings are only computed for a selection of filters, to mitigate the combinatorial explosion. By picking filter candidates from the request part of an instance (as opposed to the response part), it is prevented that multiple types are generated for a single operation. Our approach for selecting filter candidates is as follows:

- frequently occurring key-value pairs are candidates for conditions of the form $String_k : String_v$ in a filter.[1]
- frequent keys are candidates for conditions of the form $String_k : *$ in a filter.[2]
- If a filter with some key-value pair yielded good clusters (explained further in the upcoming Section 3.7.4), we extend the candidates with this key. For example if `get:drink` was successful, then `get:*` and `get:variations` also become candidates in the next refinement iteration.

In each iteration of clustering (that is, each refinement step), a procedure is executed to find a list of candidate filters, that satisfy these requirements.

3.7.4 Refinement

Finally, the heart of the clustering algorithm is the refinement of a Clustering, which is the process of splitting existing clusters into more, and more specific, clusters. The refinement algorithm takes as input a Clustering that needs refinement, and returns a Clustering that is fully refined (that is when further refinement does not improve the result). For example, say we have some Clustering:

$$[F(get = drink) \rightarrow (Type_1 + Type_2), F(!get = drink) \rightarrow (Type_3 + ... + Type_n)]$$

The second cluster in this clustering may be refined, giving the following:

$$\begin{aligned}[F(get = drink) \rightarrow &(Type_1 + Type_2),\\ &F(get = snack) \rightarrow (Type_3 + ... + Type_m),\\ &F(!get = drink \; \& \; !get = snack) \rightarrow (Type_{m+1} + ... + Type_n)]\end{aligned}$$

This happens if $(Type_3 + ... + Type_m)$ is sufficiently dissimilar from $(Type_{m+1} + ... + Type_n)$.

To perform clustering from a set of instances, we initially make a *trivial* Clustering using all provided instances, consisting of one cluster: $[F_{true} \rightarrow (Type_1 + ... + Type_n)]$ where F_{true} is a filter that always evaluates to true and $(Type_1 + ... + Type_n)$ are the Types of the responses of each individual instance. Refinement works by repeatedly selecting the "best" new Condition from a number of candidates, which is used to split the current Cluster further. The iterative process works as follows.

At the start of each iteration we have a Clustering, of which some clusters are marked *final* and some are not. A cluster has a *final* mark if and only if refining it yields no improvement, and therefore only clusters without this mark are further refined in an iteration. In the first iteration no clusters are marked as final (and the algorithm will halt when they all have a final mark).

[1]limited by the parameter n_{filter}, see Section 4.1

[2]toggled by the parameter `key_filters`, see Section 4.1

An iteration handles one cluster, say $F(Condition_1 \text{ \& .... \& } Condition_m) \rightarrow (Type_1 + ... + Type_n)$. The first step in each iteration determines candidate filters, as described in Section 3.7.3. These are determined from those instances of which the types in the current cluster were inferred. From these candidates, the "best" one is determined as follows. For each candidate, $Condition_{m+1}$, the cluster is split into one with the new filter and one with the negation of that filter . These form new clusters, respectively $F(Condition_1 \text{ \& ... \& } Condition_{m+1}) \rightarrow T^+$ and $F(Condition_1 \text{ \& ... \& } !Condition_{m+1}) \rightarrow T^-$, where T^+ and T^- represent the type of the (non-overlapping) set of instances that match those filters. Both T^+ and T^- are subsets of set $\{Type_1, ..., Type_n\}$. Then, the similarity between T^+ and T^- is calculated. From all candidates, the one that yields the highest dissimilarity is picked. Then, if and only if the dissimilarity is higher than some $threshold_d$, the cluster is accepted, and the two clusters are added to the Clustering, replacing the current one. Otherwise, the only change to the Clustering is that the current cluster is marked final.

Chapter 4

Evaluation

The goal of evaluating our method is to assess how well it works when applied to actual data. Because we want to evaluate its use on actual data, we obtained datasets from existing web services, although we had to generate our own traffic to them. The implementation is evaluated by tests that execute (parts of) our implementation and gather results. We first describe our implementation, then describe the results, and finally discuss these results in the context of our research questions, as a set-up to the conclusion in the next chapter.

4.1 Implementation

4.1.1 Architecture

We made an implementation of the type inference, minimisation, and clustering algorithms in Java[1]. We used the Justify library[2] for creating a serialisation of Types from our type system in JSON Schema, and for schema validation/type-checking. We used the Jackson library[3] for parsing JSON files.

Our type system is implemented as an abstract class called `Type`, with subclasses for specific types. We implemented `Collection` types as having references to other Types, forming a graph of Types. We added a `Universe` class, which holds a closed set of Types, and the Root Types, as to make looking-up specific types easier than traversing the graph. The methods that implement the algorithms of our method then take a (mutable) Universe object as argument, to which they apply the transformations.

One important difference between the described method and the implementation is that the method states that an Array has a single child Type, but the implementation allows Arrays to have multiple children. In the implementation of Fusion, heterogeneous Arrays are transformed into Arrays with a single Sum type that contains the children of the original heterogeneous Array. This allows the Fusion of the children of Arrays, just like described in our method. This difference does not change which types can be expressed, because an Array with multiple children has the same semantics as an Array with a single Sum type with these children. Therefore, when Type Inference and Clustering are executed as a whole, the schemas that the implementation outputs still conform to the description of the method. Still, it causes a discrepancy in our "intermediate" results, when only Initial Type Inference, or only Initial Type Inference together with Merging, are applied, because in the implementation these still allow Arrays to have multiple children, while addition of Fusion removes Arrays of that form.

[1]`https://github.com/willemsiers/apispec_learner`
[2]`https://github.com/leadpony/justify` (accessed 24th July 2019)
[3]`https://github.com/FasterXML/jackson` (accessed 24th July 2019)

4.1.2 Parameters

The implementation is configured by a number of parameters. These have significant effect on the execution of the algorithms, and therefore on the results as well. They give the implementation flexibility so that it can be adapted for learning specifications for different APIs. Below we list important ones we implemented.

- In filter candidate selection, n_{filter} is the maximum number of conditions of the form `key:value` are considered per clustering refinement iteration.
- In filter candidate selection, the boolean `key_filters` determines whether or not conditions of the form `key:*` are also considered. In our experiments this is always true.
- For clustering refinement, the `threshold` between 0 and 1 indicates how dissimilar types must be in order for the new clustering to be accepted. Generally, a higher threshold creates fewer clusters.
- The heuristics that determine whether types are fused (as described in Section 3.6.1) give rise to the following parameters :
  - The threshold for # of keys in a Record.
  - The threshold for # of optional keys in a Record .
  - The threshold for $\frac{\textit{\# of optional keys}}{\textit{\# of keys}}$.
  - A boolean to indicate whether or not children must be equal types.
  - The set of patterns that are allowed for Map keys.

When we do not mention the values of all parameters in the upcoming results, we use some defaults that were chosen by trying out several values, and settling on those that gave decent results. These default values are $n_{filter} = 10$, *key_filters=true*, *threshold=0.85*. The values we used for the parameters of the heuristics were already given in Section 3.6.1.

4.2 Input datasets

This section describes our two datasets and our method of obtaining them. We have a dataset based on the OMDb API, and a dataset based on the Wikipedia API.

4.2.1 Datamining method

We intended to obtain datasets from network traffic logs. Although we assume that these are easily obtainable for owners of a server, we did not have access to one. Therefore, we used a different method. Our approach relies on doing requests from the client side and receiving the response from the server. Each request-response pair becomes one item in our dataset.

Gathering request URLs

We obtained an initial set of possible requests by looking at documentation.

For OMDb, we looked at the API specification, which lists two methods and their parameters, for which we manually created some request URLs.

For Wikipedia, the documentation page[4] was scraped for all examples of API request URLs. This yielded 413 URLs, of the "Main Module" of the API. These do not cover the complete Wikipedia API.

Fuzzing

Because the gathered set of URLs is very incomplete, most combinations of parameters are used only once. For example, the request string `?action=query &prop=templates &titles=Main Page` is contained in the set of URLs. It is a single example of how the `query` method is used, while in practice this method will be called with many different values for prop and titles.

To mitigate this problem, we expanded the set of URLs with fuzzing. To generate our full set of requests, we started with a initial set of requests and applied a simple fuzzing technique. This fuzzing technique analyzes the URL parameters of the provided requests, and adds extra requests for combinations of these parameters, as well as some additional values we provided. Note that we do not alter the response, but rely on the server to provide it for us based on the query. Therefore, illegal requests will still yield a valid entry in our dataset.

Fuzzing altered the requests sufficiently to provide us with responses that were structurally different from those we saw in our initial dataset.

4.2.2 Description of datasets

Here we briefly describe our datasets. The OMDb dataset consists of 258 instances, and the Wikipedia dataset consists of 17676 instances.

OMDb

The Open Movie Database[5] (OMDb) REST API provides access to a user-contributed database of movies information. The page list two methods, one retrieving information by either movie title or movie ID, the other for searching movies based on keywords. There are a total of 8 parameters, and for each parameters we provided some possible arguments such as movie titles, ids, and years, to enable fuzzing. The API is small and many JSON responses are structurally similar to each other. Notable is that primitive values are presented as strings, such as `"True"` and `"1999"`, instead of `true`, and `1999`.

Our OMDb dataset has 258 instances and provides a rather complete view of the possibilities of the API.

Wikipedia

Wikipedia is an online encyclopedia running on the MediaWiki server software. The API of MediaWiki[6] provides a large set of operations, most of which are informally described in their documentation. Unfortunately, the large number of options of the API are not well-structured and responses also vary wildly in structure for small changes in options.

We started with an initial set of 413 request URLs that where scraped from examples on their documentation page. For a number of parameters we manually provided extra possible arguments which were used for fuzzing. The final Wikipedia dataset has 17676 instances, but does probably not provide a complete view of the API.

[4]`https://en.wikipedia.org/w/api.php?action=help&submodules=true&recursivesubmodules=true&toc=true` (Last accessed 14th August 2019)

[5]`http://www.omdbapi.com` (Last accessed 14th August 2019)

[6]`https://www.mediawiki.org/wiki/API:Main_page/en` (accessed 14th August 2019)

4.3 Results

4.3.1 Sizes of the output of Type Inference

Tables 4.1 and 4.3 list the sizes of the inferred types for Multi Type Inference. The sizes are listed separately as the number of nodes as well as the number of edges. Note that it counts unique nodes: `total` is the total of unique nodes that all root types have, `avg` is the average number of unique nodes per root type, and `max` is the highest number of unique nodes in one of the root types. Therefore, `avg` is not calculated by dividing the total by the number of root types. We differentiate three phases, which illustrates how the progression of our algorithm progressively changes the size of types. The output of each phase (the set of types) becomes the input of the next phase.

The output of the first phase is the set of types after executing *Initial Type Inference*. Note that after the initial phase, the number of root types equals the number of instances in the dataset because every instance gets a root type. The second phase (repeatedly) applies the bottom-up merging algorithm (Section 3.5) so that all equivalent types are merged. This decreases the number of nodes and edges. The number of root types also decreases, if and only if two root types are completely equivalent. The third phase applies the full type minimisation algorithm repeatedly, which consists of the bottom-up merging algorithm and the top-down fusion algorithm (Section 3.6). For the Wikipedia dataset, this decrease the size. Note that in this case the maximum size decreases considerably more than the average size. For the OMDb dataset, this does not decrease the sizes, because its instances are structured very uniformly.

Tables 4.2 and 4.4 list the sizes for Single Type Inference. It is computed by creating a Sum type of all root types in phase 3 of the preceding tables, and then applying the minimisation algorithm.

		# unique nodes	# edges	# root types
1) Initial	total	7567	7309	258
	avg.	29.329	28.329	
	max.	64	63	
2) After Merging	total	9	77	5
	avg.	3.0	15.4	
	max.	4	30	
3) After Merging + Fusion	total	9	64	5
	avg.	3.0	12.8	
	max.	4	28	

Table 4.1: OMDB Sizes of types from Multi Type Inference

		# unique nodes	# edges	# root types
4) Single type	total	4	9	1

Table 4.2: OMDB Sizes for Single Type Inference

.

		# unique nodes	# edges	# root types
1) Initial	total	5829617	5811941	17676
	avg.	329.80	328.80	
	max.	95193	95192	
2) After Merging	total	11727	138399	3555
	avg.	9.72	38.93	
	max.	212	4866	
3) After Merging + Fusion	total	9080	67097	2808
	avg.	9.45	23.89	
	max.	108	526	

Table 4.3: Wikipedia Sizes of types from Multi Type Inference

		# unique nodes	# edges	# root types
4) Single type	total	54	94	1

Table 4.4: Wikipedia Sizes for Single Type Inference Wikipedia

.

4.3.2 Sizes and Filters of Clustering

Table 4.5 lists the clusters that were computed for the OMDb dataset, and Table 4.6 shows those for the Wikipedia dataset. In this test we provided our implementation with the OMDb dataset and the full Wikipedia dataset It determined 3 clusters for OMDb and it determined 14 clusters for Wikipedia. Each row in the table corresponds to a cluster, showing the `filter`, then the size of the cluster type, and the number of instances that match this cluster, and finally the `dissimilarity` that this filter achieved. Because our algorithm selects the filters with the highest dissimilarity, the table only lists those filters that had the highest dissimilarity. Note that there is no overlap in the filters, and therefore the filter in a given row has the implicit condition that all filters in other rows are excluded. For OMDb, the value of the clustering parameter `threshold` was lowered, to 0.7. For the rest, the default values for parameters, as given in Section 4.1.2, were used.

Filter	#Instances	# Nodes	#Edges	Dissimilarity
`F(r:xml)`	5	1	0	1
`F(!r:xml & s:*)`	76	4	10	0.7272
`F(!r:xml & !s:*)`	146	4	30	0.7272
Totals	227	9	40	

Table 4.5: Clusters for OMDB

Filter	#Instances	# Nodes	#Edges	Dissimilarly
`F(action:query & formatversion:2 & prop:pageassessments)`	66	12	21	0.94
`F(action:query & formatversion:2 & !prop:pageassessments)`	228	20	61	0.94
`F(action:query & prop:info & generator:categories)`	83	9	21	0.98
`F(action:query & prop:info & generator:transcludedin)`	83	9	20	0.97
`F(action:query & prop:info & generator:templates)`	84	9	22	0.96
`F(action:query & prop:info & generator:redirects)`	83	9	22	0.97
`F(action:query & prop:info & generator:links)`	85	9	24	0.96
`F(action:query & prop:info & generator:linkshere)`	83	9	22	0.96
`F(action:query & prop:info & generator:*)`	640	10	30	0.96
`F(action:query & prop:info & !generator:*)`	164	27	120	0.96
`F(action:query & !prop:info)`	12242	43	104	0.88
`F(action:templatedata & titles:Template:Stub|Template:Example)`	4	8	12	0.86
`F(action:templatedata & !titles:Template:Stub|Template:Example)`	130	8	11	0.86
`F(!action:templatedata)`	2885	40	95	0.99
Totals	16860	222	585	

Table 4.6: Clusters for Wikipedia

4.3.3 Schemas of Type Inference

Types for OMDb

The results of applying Type Inference on the OMDb dataset results in 5 Root Types, of which the schemas are shown in Figures 4.1 and 4.2. Schema 1 represents an Object with only a `Response` and an `Error` field, Schema 2 represents an Object that has movie details, Schema 3 represents search results which are contained in the Array at the `Search` label, Schema 4 represents responses consisting of only a single string (in this case, when the response is a string in XML format), and Schema 5 is the same as Schema 2 except that it has 5 extra properties (such as `DVD` and `Website`). Parts of the types are shortened, indicated by ellipses (...). In Schemas 2 and 5, there are many more *properties* that have string as their type (which can still be found in the *required* field), and therefore we shortened the *properties* object. Array types are also shortened, because in this case they were homogeneous, but the same type occurred more than once (for example, in the *Ratings* property in Schema 2). The fact that homogeneous arrays have multiple entries is addressed when beside Merging, Fusion is also applied. The only difference in the OMDb schemas that adding Fusion makes is that the arrays only have single values, which causes a decrease in the number of edges, while the number of nodes stays constant.

```
{
  "$id": "C",
  "type": "object",
  "properties": {
    "Response": {
      "$id": "BBA",
      "type": "string"
    },
    "Error": {
      "$id": "BBA",
      "type": "string"
    }
  },
  "required": [
    "Response",
    "Error"
  ]
}
```

(a) OMDb Schema 1

```
{
  "$id": "BBA",
  "type": "string"
}
```

(b) OMDb Schema 4

```
{ "$id": "DGE",
  "type": "object",
  "properties": {
    "imdbID": {
      "$id": "BBA",
      "type": "string"
    },
    "Response": {
      "$id": "BBA",
      "type": "string"
    }, ... ,
    "Ratings": {
      "$id": "BDF",
      "type": "array",
      "items": [ {
          "$id": "BCY",
          "type": "object",
          "properties": { ... },
          "required":["Value","Source"]
        }, ...  ]
    }
  },
  "required": [
    "Released", "totalSeasons",
    "Metascore", "imdbID",
    ... ]
}
```

(c) OMDb Schema 2

Figure 4.1: OMDb Schemas Multi Type Inference (part 1/2)

```
{
  "$id": "BDQ",
  "type": "object",
  "properties": {
    "imdbID": {
      "$id": "BBA",
      "type": "string"
    },
    "Response": {
      "$id": "BBA",
      "type": "string"
    }.
    "Ratings": {
      "$id": "BDF",
      "type": "array",
      "items": [
        {
          "$id": "BCY",
          "type": "object",
          "properties": {
            "Value": {
              "$id": "BBA",
              "type": "string"
            },
            "Source": {
              "$id": "BBA",
              "type": "string"
            }
          },
          "required": ["Value","
              Source"]
        }, ...  ]
    }
  },
  "required": [
    "Metascore", "BoxOffice",
    "imdbVotes", "Ratings",
    "Runtime", "Language",
    "DVD", "Website",
    ... ]
}
```

(a) OMDb Schema 5

```
{
  "$id": "BCH",
  "type": "object",
  "properties": {
    "Response": {
      "$id": "BBA",
      "type": "string"
    },
    "totalResults": {
      "$id": "BBA",
      "type": "string"
    },
    "Search": {
      "$id": "BCE",
      "type": "array",
      "items": [
        {
          "$id": "BBF",
          "type": "object",
          "properties": {
            "Type": {
              "$id": "BBA",
              "type": "string"
            },
            "Year": {
              "$id": "BBA",
              "type": "string"
            },
            "imdbID": {
              "$id": "BBA",
              "type": "string"
            },
            "Poster": {
              "$id": "BBA",
              "type": "string"
            },
            "Title": {
              "$id": "BBA",
              "type": "string"
            }
          },
          "required": [
            "Type",
            "Year",
            "imdbID",
            "Poster",
            "Title"
          ]
        }, ... ]
    }
  },
  "required": [ "Response", "
      totalResults", "Search" ]
}
```

(b) OMDb Schema 3

Figure 4.2: OMDb Schemas Multi Type Inference (part 2/2)

Additionally, Figure 4.3 shows the schema resulting from Single Type Inference. This single schema represents the supertype of all previous schemas, 1 to 5.

```
{ "$id": "KFH",
  "anyOf": [
    { "$id": "KFL",
      "type": "object",
      "required": [],
      "patternProperties": {
        "(?=^[a-zA-Z]+$)": {
          "$id": "KFN",
          "anyOf": [ {
              "$id": "BBA",
              "type": "string"
            },
            { "$id": "KFS",
              "type": "array",
              "items": [ {
                  "$id": "KFU",
                  "type": "object",
                  "properties": {
                    "Type":   { "$id": "BBA", "type": "string" },
                    "Year":   { "$id": "BBA", "type": "string" },
                    "imdbID": { "$id": "BBA", "type": "string" },
                    "Poster": { "$id": "BBA", "type": "string" },
                    "Value":  { "$id": "BBA", "type": "string" },
                    "Title":  { "$id": "BBA", "type": "string" },
                    "Source": { "$id": "BBA", "type": "string" }
                  },
                  "required": []
                } ]
            } ]
        } }
    },
    {
      "$id": "BBA",
      "type": "string"
    }
  ]
}
```

Figure 4.3: OMDb Schema Single Type Inference

Excerpts of Types for Wikipedia

Figure 4.4 shows (a shortened version of) two schemas that result from Multi Type Inference without Fusion. They are nearly identical schemas as both contain a *pages* Object, which lists pages that are the response to a query. In both excerpts, the page Object has the same type (with id *FPVEK*), but the parents have different Types due to this page Object being referred to by different keys, 14640471 and 27680 respectively. This causes a large number of seemingly duplicate and equivalent schemas, in which only the key in the page Object differs.

The merging algorithm does not merge the types further because they have different keys, but, as Figure 4.5 shows, the Fusion algorithm identifies that the pages objects can be more accurately described as a Map type by using a patternProperties field.

```
{
  "$id": "FQCIU",
  "type": "object",
  "properties": {
    "query": {
      "$id": "FQCIT",
      "type": "object",
      "properties": {
        "pages": {
          "$id": "FQCIS",
          "type": "object",
          "properties": {
            "14640471": {
              "$id": "FPVEK",
              ...
            }
          },
          "required": [ "14640471" ]
        }
      }
    }, ...
  }
}
```

(a) Wikipedia Schema 1

```
{
  "$id": "JEZLN",
  "type": "object",
  "properties": {
    "query": {
      "$id": "JEZLM",
      "type": "object",
      "properties": {
        "pages": {
          "$id": "JEZLL",
          "type": "object",
          "properties": {
            "27680": {
              "$id": "FPVEK",
              ...
            }
          },
          "required": [ "27680" ]
        }
      }
    }, ...
  }
}
```

(b) Wikipedia Schema 26

Figure 4.4: Wikipedia Schemas Multi Type Inference, without Fusion

```
{
  "$id": "GUHLY.schema.json",
  "type": "object",
  "properties": {
    "query": {
      "$id": "FPVEM.schema.json",
      "type": "object",
      "properties": {
        "pages": {
          "$id": "LTDHW.schema.json",
          "type": "object",
          "patternProperties": {
            "(?=^-?\\d+$)": {
              "$id": "FPVEK.schema.json",
              ...
            }
          }
        }
      }
    }, ...
  }
}
```

Figure 4.5: Wikipedia Schema 172, Multi Type Inference, with Fusion

4.3.4 Schemas of Clustering

In Section 4.3.2 we have shown the sizes of the clusters that were computed, as well as the filters that correspond to each of the clusters. Now, we give the corresponding schemas that were produced. We only show the clustering for OMDb, and not for Wikipedia, due to its large size. We therefore show all schemas of the clustering of this dataset.

Figures 4.7, 4.8 and 4.6 show the JSON schemas corresponding to the clusters. The captions of each subfigure indicates the Filter that the schema belongs to. For additional illustration, Figure 4.8b shows one of the instances that is contained in the cluster of Figure 4.8.

The first cluster corresponds to requesting data in an XML format, as can be seen from the filter. The type of the response is then a string (in XML format). The second cluster corresponds to the operation of performing a search query, as indicated by the presence of the `s`(earch) parameter in the Filter. This operation returns an array of movie results. Lastly, the third cluster corresponds to requesting movie details for one movie, which is indicated by the lack of the `s` parameter, which according to the API reference implies the presence of an `i`(d) or `t`(itle) parameter. This operation returns an object that represents a movie, with many optional fields.

```
{
  "$id" : "AMMB.schema.json",
  "title" : "AMMB",
  "type" : "object",
  "properties" : {
    "Title" :    { "type" : "string" },
    "Response" : { "type" : "string" },
    "Error" :    { "type" : "string" },
    ...
    ,"Ratings" : {
      "$id" : "AIIE.schema.json",
      "type" : "array",
      "items" : [ {
        "$id" : "SDJ.schema.json",
        "type" : "object",
        "properties" : {
          "Value" : {  "type" : "string" },
          "Source" : { "type" : "string" }
        },
        "additionalProperties" : false,
        "required" : [ "Value", "Source" ]
      } ]
    }
  },
  "additionalProperties" : false,
  "required" : [ "Response" ]
}
```

Figure 4.6: OMDb Schema of cluster 3 (`!r:xml & !s:*`)

```
{
  "$id" : "SCU.schema.json",
  "title" : "SCU",
  "type" : "string",
  "additionalProperties" : false
}
```

Figure 4.7: OMDb Schema of cluster 1 (r:xml)

```
{
  "$id" : "AMLD.schema.json",
  "title" : "AMLD",
  "type" : "object",
  "properties" : {
    "Response" : {
      "type" : "string"
    },
    "totalResults" : {
      "type" : "string"
    },
    "Search" : {
      "$id" : "AIIG.schema.json",
      "type" : "array",
      "items" : [ {
        "$id" : "BGG.schema.json",
        "type" : "object",
        "properties" : {
          "Type" :
            { "type" : "string"},
          "Year" :
            { "type" : "string"},
          "imdbID" :
            { "type" : "string"},
          "Poster" :
            { "type" : "string"},
          "Title" :
            { "type" : "string"}
        },
        "additionalProperties" :
           false,
        "required" : [ "Type", "Year"
           , "imdbID", "Poster", "
           Title" ]
      } ]
    },
    "Error" : {
      "type" : "string"
    }
  },
  "additionalProperties" : false,
  "required" : [ "Response" ]
}
```

(a) OMDb Schema of cluster 2, with example of a matching instance

```
[ {
  "input" : {
    "s" : "pirates",
    "apikey" : "113dd685",
    "plot" : "short",
    "page" : "1"
  },
  "output" : {
    "Search" : [ {
      "Title" : "Pirates of the
          Caribbean",
      "Year" : "2003",
      "imdbID" : "tt0325980",
      "Type" : "movie",
      "Poster" : "https://..."
    },
    . . .
    {
      "Title" : "Pirates of Silicon
          Valley",
      "Year" : "1999",
      "imdbID" : "tt0168122",
      "Type" : "movie",
      "Poster" : "https://..."
    } ],
    "totalResults" : "308",
    "Response" : "True"
  }
}, {
.   .   .
},
. . .
]
```

(b) Example of one of the instance in cluster 2 (shortened)

Figure 4.8: OMDb Cluster 2 (!r:xml & s:*)

4.3.5 Cross-Validation and Generalisation

The following results illustrate to which extent generated types contain values that were not contained in the learning dataset. This indicates how well generated types generalise over JSON values. To quantify this, we have used an existing JSON Schema checker (see 4.1) to count the number of instances that match a generated specification. These results are obtained through *repeated sub-sampling cross validation* (also known as Monte-Carlo cross validation). Hereby, a type is inferred from different, random, subsets of instances, and validated using a different set of instances. These training and the validation sets do not overlap. Table 4.7 and 4.8 show the results for respectively the OMDb dataset and the reduced Wikipedia dataset[7]. The same test has been run for another schema generator, called Schema Guru (further described in Chapter 5), for sake of comparison. We compared with this tool because its type inference is similar to our research, and it also takes multiple instances as inputs. Furthermore, the tool is published, easily available, and a fully automated approach.

Column `f` shows the ratio of training data to validation data, on a logarithmic scale (i.e. they range from 0.1^4 to 1.0^4). We chose to sample logarithmically because we noticed that the values fluctuate most for small values (< 0.1) and less for larger values. The calculation is repeated 100 times for each ratio, each time picking different subsets. For Schema Guru it is repeated 10 times due to longer runtimes.

The column *Correct % avg.* shows the average percentage of instances (from the validation set) that correctly type check with the generated specification (made from the training set). This column shows an increasing trend as the ratio (`f`) increases. The number of instances that correctly type checks is not strictly increasing because the size of the validation set changes as the size of the training set changes, but the percentage is increasing.

Note that for the rows whose values are marked with a star (*), the training set equals the validation set, both containing of 100% of the dataset. This serves as to validation that the specifications are complete, and thus will at least match all instances that it was generated from.

[7]We reduced the Wikipedia dataset size to prevent Schema Guru from running out of memory on our system, for very large inputs. The reduced dataset contains 3555 instances. To this goal, a pre-processing step was applied to remove instances that had duplicate types according to our Type Inference algorithm. This seemed a better method than simply taking some subset of the data, because that would could increase skewness of the dataset.

OMDB	f log_4	f %	Correct % avg	Correct % stddev	Training #	Validation #
Current work	0.1	0.01	37.1	39.17	1	257
	0.2	0.16	37.1	39.17	1	257
	0.3	0.81	75.3	40.60	3	255
	0.4	2.56	93.2	20.68	7	251
	0.5	6.25	97.2	7.83	17	241
	0.6	12.96	98.4	2.60	34	224
	0.7	24.01	98.9	2.25	62	196
	0.8	40.96	99.5	1.25	106	152
	0.9	65.61	99.8	0.47	170	88
	1.0⋆	100⋆	100⋆	0⋆	258⋆	258⋆
Schema Guru	0.1	0.01	2.8	6.94	1	257
	0.2	0.16	2.8	6.94	1	257
	0.3	0.81	37.7	33.65	3	255
	0.4	2.56	54.0	23.71	7	251
	0.5	6.25	80.7	18.03	17	241
	0.6	12.96	86.7	8.44	34	224
	0.7	24.01	92.8	9.05	62	196
	0.8	40.96	94.8	5.06	106	152
	0.9	65.61	97.9	1.30	170	88
	1⋆	100⋆	98.0⋆	0⋆	258⋆	258⋆

Table 4.7: Generalisation results. Current work 100 repetitions, Schema Guru 10 repetitions

Wikipedia (Reduced)	f log_4	f %	Correct % avg.	Correct % stddev.	Training #	Validation #
Current work	0.1	0.01	0.006	1.42	1	3554
	0.2	0.16	0.14	7.35	6	3549
	0.3	0.81	12.04	537.83	29	3526
	0.4	2.56	61.25	113.84	92	3463
	0.5	6.25	76.23	51.55	223	3332
	0.6	12.96	82.00	35.21	461	3094
	0.7	24.01	85.77	20.85	854	2701
	0.8	40.96	88.65	15.16	1457	2098
	0.9	65.61	90.85	9.66	2333	1222
	1⋆	100⋆	100⋆	0⋆	3555⋆	3555⋆
Schema Guru	0.1	0.01	0.04	0.89	1	3554
	0.2	0.16	0.25	3.83	6	3549
	0.3	0.81	1.63	8.41	29	3526
	0.4	2.56	5.07	8.73	92	3463
	0.5	6.25	8.95	16.56	223	3332
	0.6	12.96	13.80	18.91	461	3094
	0.7	24.01	18.67	19.83	854	2701
	0.8	40.96	23.91	15.44	1457	2098
	0.9	65.61	29.18	16.99	2333	1222
	1⋆	100⋆	98.79⋆	0⋆	3555⋆	3555⋆

Table 4.8: Generalisation results. Current work 100 repetitions, Schema Guru 10 repetitions

4.3.6 Runtimes

The following figures show the runtime performance of our implementation, which is the time it takes to run the Type Inference and the Clustering algorithms. Figure 4.9 and Figure 4.10 illustrate the relation between the runtime of Type Inference and the size of the input, split into Initial Type Inference and Minimisation, such that the sum of these two runtimes is the runtime of Type Inference for that input size. Figure 4.11 illustrates the relation between the runtime of Clustering and the size of its input. Figures 4.12 and Figures 4.13 illustrates the effect of two parameters on Clustering, respectively the number of filters per iteration of clustering refinement ($n_{filters}$), and the threshold for dissimilarity that is used to accept or discard new clusters.

The defaults for clustering performance tests were different than those in the other results, as follows: $n_{filters} = 8$, threshold= 0.9, use_keyfilters=true, inputsize=1024. For clustering, 10 repetitions were used for each setting. For type inference, 100 repetitions were used for each setting. We used average times, as the deviations between minimum and maximum runtimes were small. In the Appendix, these runtimes are given in Tables A.2, Figure A.3 and Figure A.1.

For the tests that compare runtimes for different input sizes (Figure 4.11, 4.9 and 4.10), we performed linear regression and power regression and found that results indicate quadratic relation between size and runtime, rather than a linear relation. In these figures, the functions that approximates the runtimes are shown by the dashed line.

The benchmarks were performed on a Ubuntu 16.04 system with an Intel 7700HQ processor, 16GB RAM and an SSD, using 64-Bit OpenJDK 12.0.1, ran with JVM parameter `-Xmx10g` to limit the heap size to 10GB.

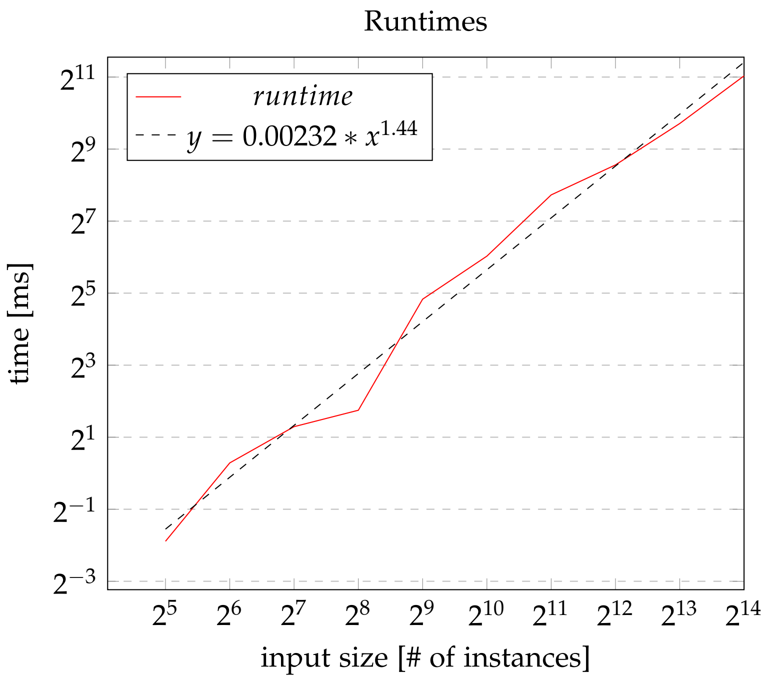


Figure 4.9: Runtimes of Initial Type Inference for several input sizes

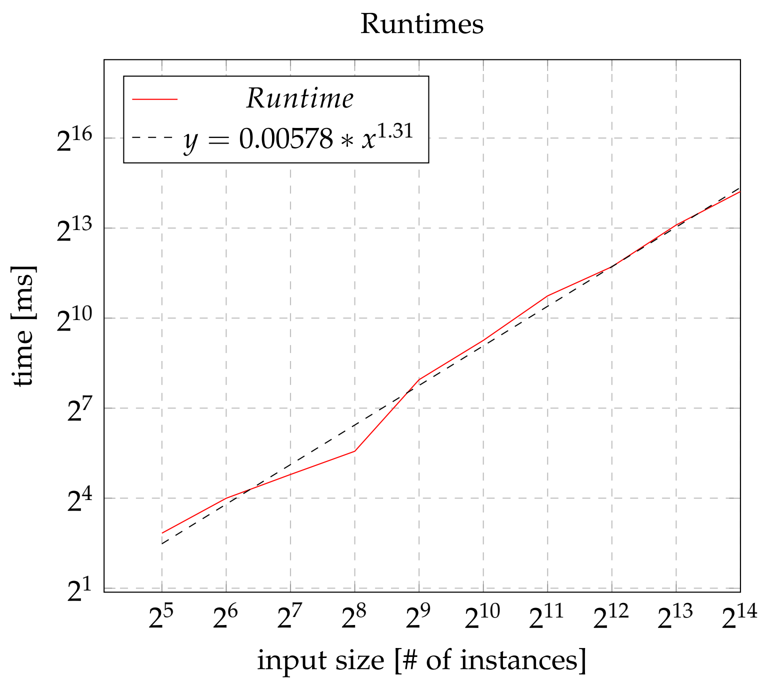


Figure 4.10: Runtimes of Minimisation for several input sizes

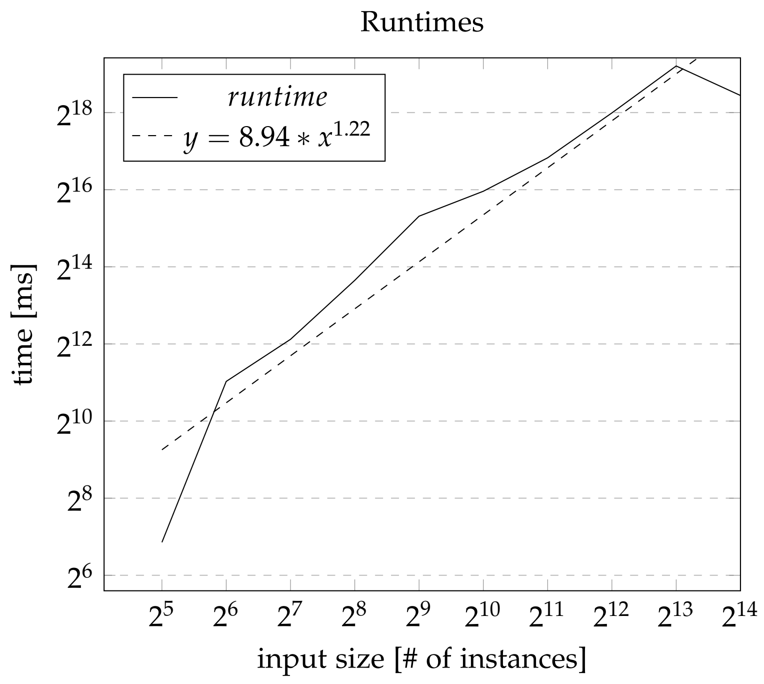


Figure 4.11: Runtimes of Clustering for several input sizes

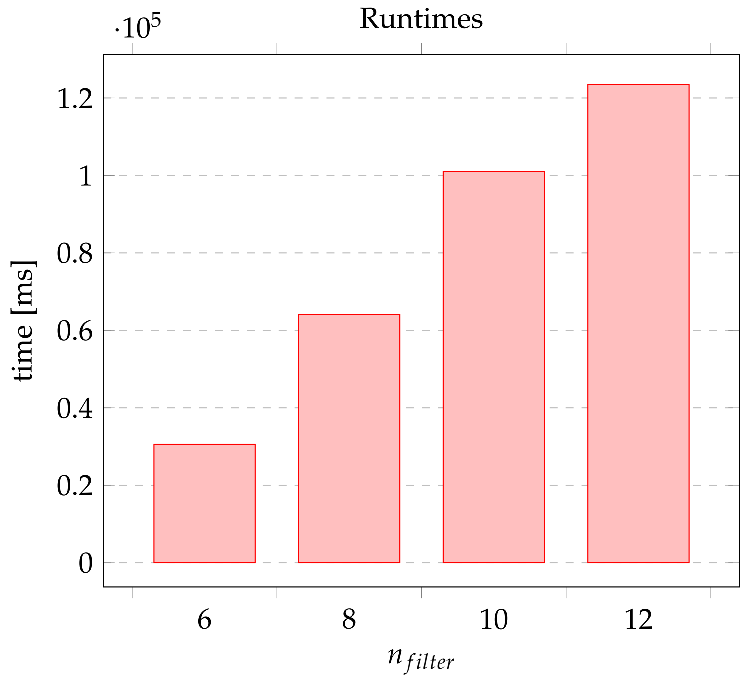


Figure 4.12: Runtimes of Clustering for several n_{filter}

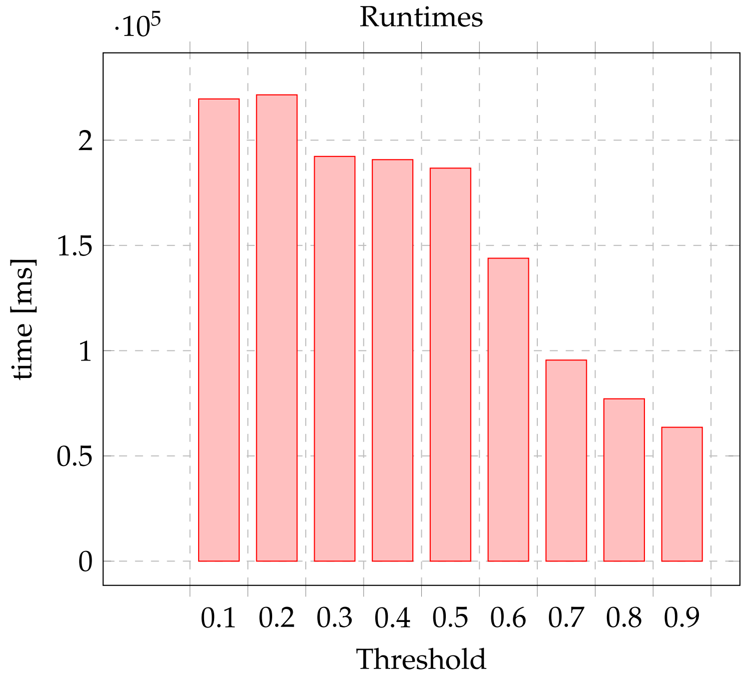


Figure 4.13: Runtimes of Clustering for several thresholds

4.4 Discussion

This section discusses the results as a way of answering research questions 2 and 3. Research Question 2 (*How useful are these automatically learned API specifications for formal purposes?*) is answered through the discussion points in Sections 4.4.1, 4.4.2 and 4.4.6. Research Question 3 (*How useful are these automatically learned API specifications for semantic purposes?*) is answered through the discussion points in Sections 4.4.6, 4.4.5, 4.4.4 and

4.4.3. Not belonging to any particular research question is the discussion of the runtime performance, in Section 4.4.7.

4.4.1 Completeness

To reiterate, for some input set, the method creates a complete specification for this input, if and only it flags all inputs as correct. The validation results in Section 4.3.5, specifically the rows marked with a star (*) (of the current work) in Tables 4.7 and 4.8, indicate that the generated specifications are complete. For these rows, the whole dataset was the training set as well as the validation set. The specifications learned from the complete dataset correctly check all instances in that dataset. It can not be concluded that specifications are complete with respect to an API, because there may be valid inputs that are not contained in the dataset. We do however conclude from this data that for a complete dataset, it learns a complete specification.

4.4.2 Lack of Soundness

A type specification is sound if and only if all instances that match it are valid inputs for the API. The following example illustrates that our type system is not sound. The Type specification *Number* allows the decimal value 1.5, which is an invalid value for a system that only supports integers. Because such an invalid input would be flagged as correct by the specification, the type system is not sound.

For applications that only receive JSON data this does not form a large problem, except that its programmers may have to consider a broader range of values than what will be received in practice. On the other hand, it does hamper its usefulness for type checking an application that sends JSON data, because type checking will then not ensure that it sends valid data.

4.4.3 Effects of different parts of Type Inference

Applying Merging and Fusion decreases sizes of types significantly, based on the results on the sizes of type inference (Section 4.3.1). Let us call the ratio between the size before minimisation and after minimisation *compression*. This is calculated using the *average* sizes, because the *total* sizes are largely influenced by the number of instances that were used, and thus, the compression in terms of total size could be inflated to be arbitrarily large by inserting duplicate instances into the dataset. The compression that merging achieves on the average number of nodes is 29.329/3 = 9.8 for OMDb and 329.804/9.721 = 33.9 for Wikipedia. The compression on the average number of edges is 28.329/15.4 = 1.8 for OMDb and 328.804/38.930 = 8.4 for Wikipedia.

For the results of Phase 3, when both merging and fusion are applied, the average number of nodes in both datasets remains mostly equal, compared to Phase 2, but the average number of edges still significantly decreases. For OMDb, adding Fusion increases the compression of the number of edges to 29.329/12.8 = 2.3, and for Wikipedia, it increases to 328.80/23.89 = 13.8. This decrease is caused by representing Record with many outgoing edges as Map types, and by the propagation of Sum types. It may also be influenced significantly by the fact that in our implementation Arrays can have multiple children, but these are removed in Fusion.

For Fusion it is also interesting to look at the maximum sizes. In Phase 3, the maximum number of nodes per type is reduced from 212 to 110, and the maximum number of edges is reduced from 4866 to 531. The maximum decreases more than the average decreases, and thus it seems that the largest minimisation occurs in types that are already large. This is because map-like structures in types inflate the size a lot when they are encoded as a Record.

4.4.4 The effect of clustering on sizes of types

We look at the results about the Type Inference Sizes and Clustering Sizes (Sections 4.3.1 and 4.3.2) to assess the effect of clustering on the sizes of the generated types, compared to Single Type Inference. For this, we look at the sizes of clusters in Tables 4.5 and 4.6, and compare them to the size of single types in Tables 4.2 and 4.4 where clustering is not applied . The single type describes the same instances as all types of the clusters combined.

Our hypothesis was, firstly, that a clustering would always be larger than the single, minimised, type because they contain more redundancy due to increased precision, and secondly that individual sizes of clusters would always be smaller than the single type because they only describe a limited set of instances. However, the results disprove the latter. The single type of OMDb has 4 nodes and 9 edges. The maximum number of nodes among the clusters is 4, and the maximum number of edges is 30. Sizes of individual clusters here are thus larger than the single type. On the other hand, the total size of the clustering, 9 nodes and 40 edges, is indeed larger than the single type.

A slightly different observation holds for the Wikipedia dataset. The single type of Wikipedia has 54 nodes and 94 edges, while the maximum of the clusters is 43 nodes and 120 edges. In this case, all cluster types have fewer nodes, and most have fewer edges, although some exceed the single type size. The sum of the cluster sizes is again larger than the size of the single type, at 222 nodes and 585 edges.

The reason that our hypothesis is not supported by the results is a deficiency in the current implementation of Fusion, namely that the heuristics are evaluated on individual cluster types only instead of the whole clustering. This is because we had to disable Fusion during the clustering refinement as a workaround to a programming error whereby we substituted Map types computed in the unrefined cluster into the refined clusters. This caused the results to become extremely general, as the Map types are already initially computed for the whole dataset and these are copied into the refined clusters. The reason we substituted Map types from unrefined clusters into refined clusters in the first place is that we need to preserve the requirement refined cluster types are smaller than their unrefined type, which would otherwise be broken because the heuristics could infer some Patterns in the unrefined cluster than could not be inferred its smaller, refined clusters. The solution would be to only compute the Pattern for the Map types in the unrefined cluster, and instead of fully substituting it, to compute the actual Map type in the refined clusters, based on only its own instances.

From the results from Wikipedia we have some support to suggest that clustering generates smaller individual types, at the cost of increasing the total size. So we can conclude that clustering generates smaller individual types in some cases, but does generally not hold for the current implementation.

4.4.5 The effect of clustering

This section considers the clustering results in Section 4.3.2. For OMDb, the three clusters that were found were:

- *parameter `r` has value `xml`.* According to the API reference[8], this parameter accepts the values `json` and `xml`, which changes the response format accordingly.

[8]`http://www.omdbapi.com`

- *parameter `s` exists*. The argument to this parameter is the search query for a movie. Search is one of the two operations of the API.
- *parameter `s` does not exist (this filter matches the remainder of the data)*. This matches the other operation of the API that retrieves details of a single movie. This operation is formed which is formed by the absense of the search parameter `s`, which implies the presence of either a `i` or `t` parameter.

The API reference[9] lists that there are two operations, *search* and *get by ID or Title*. Clustering thus identified the two methods that are documented in the API reference. The former is listed explicitly while the latter is only implied by the absence of the `s`(earch) parameter. A third operation was identified, which is not listed in the API reference, namely the one formed by the format parameter `r`. This is caused by the fact that a response in XML format gets a very different type than a response in JSON format, namely a string type. All operations of the API were thus found, with one additional operation. Therefore, the generated specification is not equavalent to the reference, but it could be argued that the format parameter has such a significant effect on usage of the API, that explicitly listing it as a separate operation may be a welcome addition.

14 clusters were found for Wikipedia. While for this API it is less trivial to determine the number of supported operations, a glance at the documentation reveals that this clustering does not provide a good view of the whole API. For example, our clustering suggests that the `action` parameter takes as argument either `query` or `templatedata`, and leaves the rest unspecified. The reference lists 78 possible arguments for this parameter alone, while, intuitively speaking, the argument to this parameter determines the operation. The cause for not finding many operations is the used configuration. Decreasing the dissimilarity threshold would yield more clusters, but when we inspected those results it would have the opposite problem of finding too many filters. It would then create separate clusters for each individual title of a Wikipedia article that a request was done for, because the responses are considered different types because the title of the article is used as keys in the objects. The Fusion algorithm is in fact what is supposed to resolve this issue, but because we disabled Fusion during the clustering iteself (as was explained in Section 4.4.4), it was unable to detect that the responses of an operation have the same type when the argument to the `title` argument varies.

4.4.6 Generalisation of specifications

Results of cross-validation were shown in Section 4.3.5. Looking at the relation between training ratio `f` and the ratio of correctly checked instances in Tables 4.7 and 4.8 we notice that a high ratio of correctness is achieved for rather low training ratios. For example, in the OMDb table, a higher than 90% correctness is achieved for a training percentage of only 2.56%. In the Wikipedia table, 90% correctness requires a training ratio of 65.61%.

However, for no training ratio that is not 100% did it reach 100% correctness. This is due to the many repetitions of repeated random sub-sampling, in which it is likely that in some repetition the training set misses an instance that has a unique type. This suggests that from incomplete data our method does not create specifications that capture an entire API. We can speculate however that in practical application it may still generate specifications that capture the entire API because a dataset will likely contain at least one instance that is representative for some type. The limitation in terms of generalisation can thus be attributed to missing edge cases, but our results suggest that it does generalise strongly over instances.

[9]`http://www.omdbapi.com`

Based on Tables 4.7 and 4.8 we compare our method with Schema Guru in terms of generalisation. The correctness ratio for Schema Guru are structurally lower than those for our approach, for a given training ratio. For the OMDb dataset it achieves a correctness ratio that is close to the that of our method, when using a large enough training ratio of 65.61%. The values are much lower for the Wikipedia dataset, where at a training ratio of 65.61%, it only correctly matched 29% of the instances. Also notable, is that Schema Guru does not achieve a 100% correctness when trained with the same set as the validation set (i.e. the last row of the tables). This means that this tool does not produce *complete* results. The cause for this is that Schema Guru does not handle strings as top level values, despite strings being valid JSON according to the specification. These values are ignored, and therefore the schema does not describe these values. As a result, it fails on those instances where the server responded with a string.

4.4.7 Runtime performance

From runtime results in Section 4.3.6, we already saw that there is a quadratic relation between input size and runtimes. An interesting artifcact in Figure 4.11 shows that it is not monotonic, considering that the runtime decreases between input size 2^{13} and 2^{14}. We think this is because of the combination of that different Filter candidates will be considered for different input data, and that the input data is only randomised initially but not between repetitions. To clarify, remember that only a selection of the most common key-value pairs become Filter candidates. The most common key-value pairs may change when input size is increased, and therefore a different *best* filter can be found, resulting in a different clustering and a different runtime.

Our implementation was not optimised for performance, and clustering tests ran within minutes and type inference tests within seconds. We do not expect that the runtime complexity will be an issue in practice, even for reasonably complex APIs. However, if the size of a network log used as input data increases much beyond the sizes we used, it is advisable to use either a selection of the log, or optimise the current implementation so that equivalent entries in the network log have limited effect on the runtime.

Chapter 5

Related Work

This chapter discusses previous work related to our research, to place the current research into the context of others. The first section provides a global outline, and subsequent sections give more details on specific works. Finally, we provide a comparison between previous works about schema inference and ours.

5.1 Outline

First, we discuss three JSON schema inference tools that are practical in use. While these are limited in terms of handling multiple instances, they provide useful ideas for improvement of other inference tools, and an idea of which user-friendly tools exist.

Next, the work by Ed-douibi et al. [11] seems most similar to ours in terms of its goal, as it aims to generate full openAPI specifications of APIs. Their contribution of a meta-model of openAPI, as well as a structured approach to populate these models, are valuable.

Then there is an approach by DiScala and Abadi [10], which is based on functional dependencies, and has a different goal. They generate normalised relational schemas, for use in relational databases. Their goal was to improve over the work by Chasseur et al. [9]. The technique by DiScala and Abadi [10], for finding relations between fields in JSON data can be valuable for our clustering algorithm. Concerning their inferred schemas, they are not suitable for our goal, because the schema does not capture the structure of a JSON instance. Therefore it can not be used to, for example, decode or type check a JSON response that an API returned.

In terms of methodology, the approach based on structural information, by Baazizi et al. [2] resembles our work. It also has a fusion algorithm, which operates on array types that are inferred in the initial schema inference phase. They show that their approach successfully generates succinct types for 2 out of 4 datasets.

5.2 JSON Schema inference tools

There exist user-friendly web applications that determine a schema for a JSON document. We first discuss two tools that use a single JSON instance to infer a JSON schema, which limits these tools in their ability to infer a type that generalises over multiple documents.

5.2.1 Single instance inference tools

With the tool called Jsonschema.net[1] a user creates a schema as follows. The user provides a JSON file for which a schema must be inferred. Then, they can specify options such as the minimum and maximum of *number* types and whether or not to set the *uniqueItems* flag for arrays. Note that these user-specified options are simply copied into the schema, not inferred from the input. In the generated schema, the tool also populates the *examples* field of properties with the value of the field in the given JSON, and also populates the *title* field for each subschema with the label that is used to reference it. As the tool provides multiple options for how the schema should be built from the given JSON, a good use-case for the tool is building a schema with some user-provided knowledge of the data.

Quicktype[2] is more advanced, but has a similar user-interface, where the user supplies a JSON document. It additionally has (partial) support for features such as detecting UUIDs, dates, and times in text values, as well as and enums. They also have an option that infers maps, which, when all children of an object have equal types and the property names appear to be map keys, replaces the *properties* field in the schema with an *additionalProperties* field. They use a Markov model of property names in objects to statistically distinguish map keys from object property names [15].

5.2.2 Schema Guru

Schema Guru is a tool that handles a set of JSON instances to create a single schema. It is assumed that all instances are of the same type, however it has the option to segment instances on a user-provided identifier. For example, the user can specify that instances with the property `"event" : "A"` are separated from `"event" : "B"`, thereby producing two schemas. It uses the multiple instances to infer useful properties such as lengths of strings, ranges of numbers, and recognises values that are formatted as, for example, dates, IP addresses, and URLs. Furthermore, it detects enums. These are text properties in JSON of which the value is always from a specified set of strings.

5.3 Relational schemas for semi-structured data

There is also work that creates relational schemas from semi-structured data (such as JSON).

DiScala and Abadi [10] noted the shift from structured data towards semi-structured data. This shift occurs due to developers shying away from the rigidity of the relational model used in relational databases (RDBMS). Their method infers a relational schema from semi-structured data, so that an RDBMS can be used to querying this data. In the relational model, one field in the data can reference another piece of data. This allows a database to be normalised. One advantage is that this de-duplicates data. Furthermore, it imposes constraints on the data, because relations between data are made explicit. This is in contrast with semi-structured such as JSON, which inlines all fields and allows fields can be added or removed on a per-instance basis due to a lack of schemas.

Preceding their work, work by Chasseur et al. [9] yielded a proof-of-concept tool to already create relational schemas from semi-structured data, called Argo. Their motivation was that database systems that store JSON documents, lack a good query language and ACID safety guarantees (measures that ensure, among others, consistency

[1]`https://jsonschema.net` and `https://github.com/jackwootton/json-schema` (accessed 24th July 2019)

[2]`https://quicktype.io` (accessed 24th July 2019)

of a database), which are present in many RDBMSs. Their tool enables storing JSON documents into a RDBMS, and it can be queried with the Argo/SQL language. This is done by encoding JSON objects as 3-tuples (object-id, field-name, field-value), such that each value in a JSON object becomes a row in the database. They show that relational schemas can indeed be made from semi-structured data.

However, according to DiScala and Abadi [10], Argo does not facilitate data exploration. They, similarly to the current research, then aimed to create a semantic schema. They state that it is better not to rely on structural information to determine a schema, because structure may be meaningless if structure can be changed arbitrarily (which is the case in JSON). Instead, their encoding relies on finding functional dependencies between fields. For two fields A and B, A has a functional dependency on B if the value of A sufficiently predicts the value of B. That is, if the following condition holds for some threshold α: $\frac{\#\text{ of unique values of } A}{\#\text{ of unique } (A,B)\text{pairs}} > \alpha$. Such fields together form a semantic entity, and, because of their coherence, are put together in a separate table.

Their results are normalised relational schemas, which, according to their evaluation, caught many meaningful attributes (which they determined manually). Counter to their approach, our approach is based on structure. They already noted that a hybrid approach that uses values as well as structural information will yield higher quality schemas.

5.4 Structural schema inference

Baazizi et al. [2] also noted the lack of schemas for JSON datasets, and laments that queries can not be statically checked and that the user can not benefit from structural information to figure out the structure of a dataset. Their schemas do not aim to describe relations in the data (for usage in an RDBMS), but instead describe the data structurally. They created a type inference algorithm that captures irregularities in a set of JSON instances, and creates a single type for it. They assume heterogeneous data, with at most small variations. Important evaluation metrics they use are *precision* and *succinctness* of the inferred schemas. These two metrics are a trade-off, but their target for succinctness is: "small enough to enable a user to consult it in a reasonable amount of time, to get knowledge on the structural and type properties of the JSON collection". For assessing succinctness, they measure number of distinct types, minimum, maximum and average size of types (i.e. number of nodes in the AST), and size of the fused type. Their evaluation used 4 datasets, and found succinct schemas for 2 of them. Similarly to our research, their approach uses multiple examples of JSON instances to create a single schema. Their approach consists of two parts, initial type inference and application of a *fuse* operation.

5.5 Generating API specifications

The work of Ed-douibi et al. [11] focusses on a similar goal as us, that is discovering a specification for a REST API from usage examples. They note the availability of Swagger and openAPI but the lack of use of it. Instead, API documentation, if it exists, often consists of informal documentation. Therefore, developers need to manually discover how to call the API and how to encode a query and decode the response. Their solution to make formal specification more widespread is to make it easier to create them. Their approach consists of two parts, both accompanied by a meta-model: behavioral discovery and structural discovery. The behaviour model lists operations on the API by their path and parameters, and describes its response. The model is populated by extracting the information from examples. Parameters are detected in the path if it is a UID (i.e. an integer or hexadecimal string), and query parameters are always added to the model instance. Conflicts in parameter types are resolved by taking the most generic form, e.g. if it can be an integer and a string, it becomes a string. If and only if the response is JSON, the structural discovery is then started. The structural model is updated with every example of a type that it sees.

5.6 Comparison

Table 5.1 compares features of different methods and tools. It compares the following selected features:

Multiple	It supports multiple instances as input
Heterogeneous	It supports and handles heterogeneous input data
Semantics	It attempts to generate semantically meaningful schemas
Structural	The schemas describe the original structural
Subschemas	Subschemas are identified
Patterns	Support for detection of enums or patterns in values
Use case	One potential use-case (among others)

	Multiple	Heterogeneous	Semantics	Structural	Patterns	Subschemas	Use case
JsonSchema.net	×	×	✓	✓	×	×	Building JSON Schema
QuickType	×	×	✓	✓	✓	✓	Building JSON Schema
Schema Guru	✓	✓[1]	✓	✓	✓	×	Advanced Schema inference
Argo [9]	✓	×	×	×	×	×	Storage in RDBMS
DiScala and Abadi [10]	✓	✓[2]	✓	×	×	✓	Semantic storage in RDBMS
Baazizi et al. [2]	✓	✓[2]	✓	✓	×	✓	Schema inference on Big Data
Ed-douibi et al. [11]	✓	✓[2]	✓	✓	×	✓	Building API specifications
Current work	✓	✓	✓	✓	×	✓	API specification from heterogeneous data

Table 5.1: Feature comparison of related work

[1]If user provides segments
[2]Limited/if user-guided

Chapter 6

Conclusion

Firstly, the current research found a method to automatically generate type specifications for JSON based APIs. We have shown the type system of our specifications, the algorithms to create them, and previous work on this topic. We did not find previous work that both preserved (our definition of) completeness and applied a form of clustering to produce specifications.

Secondly, we assessed the usefulness of these specifications when employed in formal applications such as static type checking. A limitation for formal application is the lack of the soundness property, which means that using these specifications for type checking may yield false positives. On the other hand, they do hold the completeness property, which means that type checking will at least not yield false negatives. Experimental evaluation showed that it is complete with respect to its learning dataset, and therefrom we conclude that if the dataset covers the whole API, that the specification is also complete with respect to the API. However, an incomplete dataset may still produce useable specifications because of the generalising ability of the specifications. Experimental evaluation of our *type inference* algorithms found that the *merging* and the *fusion* algorithms decreases the size of types significantly, which increases consiseness, and thereby, readability of types. For the *fusion* algorithm, the increased consiseness comes at the cost of precision. The *clustering* algorithm increases the overal size of a type specification (because it consists of multiple constituent types), but its constituent types are in most cases smaller than the non-clustered type. Thereby, *clustering* increases precision of types at the cost of conciseness.

Thirdly, we assessed whether the specifications convey semantics of the API to a programmer. One positive for the semantic aspect is that the *merging* algorithm identifies recurring structures in the data and makes them explicit, by identifying equivalent subgraphs within types, and making them shared by replacing them with references. Another positive is that the *fusion* algorithm identifies semantically equivalent types (according to a heuristic) and makes this fact explicit, by creating a supertype of these semantically equivalent types and fusing common parts. The latter adds the risk of creating a type that is too general if the heuristics do not match the learning dataset, and using it requires more care than the former. Another positive is that *clustering* makes explicit which specific types specific API operations return, although our evaluation found that for only one of the two datasets, the result conforms to the reference specification, but for the other it did not produce a great result. Clustering only works well if there is a strong correlation between a request and the type of its response.

We conclude that our automated method can be of use for formal purposes, because completeness of specifications implies they can reliably detect typing errors in a program, even though it should be taken as a serious consideration that the lack of soundness prevents it from proving a program correct. The extent to which the generated specifications convey semantics of APIs varies, but the techniques we present in this thesis increase usefulness of specifications for programmers.

6.1 Limitations and Future work

Subtyping of Primitives

Schema Guru and Quicktype infer ranges in number values, lengths of strings, and enums from string values. These are subtypes of primitive types. The current work supports explicit subtyping through Sum types and implicit subtyping occurs for Collection types, but primitives are not there exists no subtyping for primitive values. One could define subclasses for primitives, such as dates and ip-addresses, or integers and natural numbers. A possibility is to apply automata learning as well as on primitive values, to detect patterns in strings.

Heuristics and Map detection

The heuristics for introducing Sum types that we implemented were added based on cases we encountered, where it seemed that the semantically meaningful type was not inferred due to syntactical oddities. One could experiment with different heuristics. One that could use some research is a heuristic that only introduces a Sum type or a Map type if all children bare enough similarity. We currently only implemented this as a boolean toggle (either children must be equal, or they do not need to be).

An alternative for Map detection is the method based on Markov model of property names, as seen in QuickType. Although, unlike our method, this does not provide the pattern for the property names, it is more flexible.

Extend instance model and support more aspects of requests

In the current model instance model we use, the availability of request data is limited. Only the query parameters and arguments are available. It does for example not consider *paths* and the *Hypertext Transfer Protocol method* Currently, our clustering implementation does not support APIs whose operations are formed by the path of a request. The meta model made by Ed-douibi et al. [11] could be used to capture all elements of an OpenAPI specification.

Application on more APIs

Applying the current research on more APIs will yield more interesting results. A limitation of our research was that obtaining API traffic data was more difficult than expected. We had no access to suitable server-side request logs. Reasons include confidentiality and lack of logging facilities. Another obstacle is that most traffic is end-to-end encrypted, making it harder to intercept at the client-side. Also, we did not use API requests that require authentication, and implementing support for this enables application on more APIs.

Comparison with handmade specifications

It is interesting to compare results of our method to specifications that a human would make. An experienced API designer or specification writer may make a specification, and this is then to be compared to the results of our method. We expect that results would differ significantly, but it may yield useful insights into limitations of our method that are valued by programmers.

Clustering

In terms of improving our current clustering approach, one can look at the following.

Calculate similarity of types using a different metric. For example, *Tree edit distance* is used for comparison of trees [17]. It is the minimal number of tree-edit operations (adding, removing and renaming nodes) required to transform one tree into another. An alternative, type specific edit-distance needs to be devised, because tree edit distance does not seem to match with similarity of types. That is, for example, renaming of a node in a type tree can result in very different types, while still only having an edit distance of 1.

Another possibility is to calculate a score for the clustering on a complete clustering. Currently, the score is only computed for each refinement step, which may be limiting compared to computing it for the total clustering.

One could also look into an approach in machine learning. We currently can not do K-means (or X-means) clustering because we have no way to compute a number to represent the position of a Type on a plane. Important features of types would need to be devised so that such traditional clustering methods can be used.

Bibliography

[1] Alblas, H. (1991). Introduction to attribute grammars. In *International Summer School on Attribute Grammars, Applications, and Systems*, volume 545 LNCS. Springer.

[2] Baazizi, M.-A., Lahmar, H. B., Colazzo, D., Ghelli, G., and Sartiani, C. (2017). Schema Inference for Massive JSON Datasets. In *Extending Database Technology (EDBT)*, pages 222–233.

[3] Bechhofer, S., Volz, R., and Lord, P. (2003). Cooking the Semantic Web with the OWL API. In *International Semantic Web Conference*, pages 659–675. Springer.

[4] Berners-Lee, T., Fielding, R., and Masinter, L. (2005). Rfc 3986, uniform resource identifier (uri): Generic syntax, 2005. `http://www.faqs.org/rfcs/rfc3986.html`. Accessed: 25-07-2019.

[5] Blanchette, J. (2008). The Little Manual of API Design. *Trolltech, a Nokia company*.

[6] Bloch, J. (2006). How to design a good api and why it matters. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 506–507. ACM.

[7] Bray, T. (2014). The JavaScript Object Notation (JSON) Data Interchange Format. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`.

[8] Burke, D. and Johannisson, K. (2005). Translating Formal Software Specifications to Natural Language. In *International Conference on Logical Aspects of Computational Linguistics*, pages 51–66. Springer.

[9] Chasseur, C., Li, Y., and Patel, J. M. (2013). Enabling JSON Document Stores in Relational Systems. In *WebDB*, volume 13, pages 14–15.

[10] DiScala, M. and Abadi, D. J. (2016). Automatic Generation of Normalized Relational Schemas from Nested Key-Value Data. In *Proceedings of the 2016 International Conference on Management of Data*, pages 295–310. ACM.

[11] Ed-douibi, H., Izquierdo, J. L. C., and Cabot, J. (2017). Example-driven web API specification discovery. In *European Conference on Modelling Foundations and Applications*, volume 10376 LNCS, pages 267–284. Springer.

[12] Myers, B. A. and Stylos, J. (2016). Improving API usability. *Communications of the ACM*, 59(6):62–69.

[13] Pautasso, C. (2014). RESTful Web Services: Principles, Patterns, Emerging Technologies. In *Web Services Foundations*, pages 31–51. Springer.

[14] Pezoa, F., Reutter, J. L., Suarez, F., Ugarte, M., and Vrgoč, D. (2016). Foundations

of JSON Schema. In *Proceedings of the 25th International Conference on World Wide Web*, pages 263–273. International World Wide Web Conferences Steering Committee.

[15] Probst, M. (2018). Detecting Maps. *https://blog.quicktype.io/markov/*. Accessed: 05-07-2019.

[16] SmartBear Software (2019). State of API 2019. *https://smartbear.com/SmartBearBrand/media/pdf/SmartBear_State_of_API_2019.pdf*. Accessed: 25-07-2019.

[17] Zhang, K. and Shasha, D. (1989). Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. *SIAM Journal on Computing*, 18(6):1245–1262.

Appendix A

Tables of Runtime Performance

	Initial Type Inference avg. [ms]	Minimisation avg. [ms]	Clustering avg. [ms]
n=32	0.27	7.13	115.7
n=64	1.22	15.97	2085.7
n=128	2.45	27.65	4444.6
n=256	3.36	47.26	12838.8
n=512	28.48	246.84	40654.9
n=1024	65.22	611.97	63658.9
n=2048	211.82	1710.1	116052.4
n=4096	377.25	3351.29	260167.2
n=8192	836.85	8780.9	604805
n=16384	2092.08	19028.28	355776.4

Table A.1: Runtimes of Type Inference and Clustering for several input sizes.

$n_{filters}$	Runtime avg. [ms]
6	30589
8	64149
10	100998
12	123434

Table A.2: Runtimes of Clustering for several values of $n_{filters}$

`threshold`	Runtime avg. [ms]
0.1	144
0.2	139
0.3	114
0.4	111
0.5	108
0.6	85
0.7	50
0.8	36
0.9	23

Table A.3: Runtimes of Clustering for several thresholds