Statically Checking REST API Consumers

Nuno Burnay, Antónia Lopes, and Vasco T. Vasconcelos

LASIGE, Faculdade de Ciências, Universidade de Lisboa, Lisbon, Portugal

Abstract. Consumption of REST services has become a popular means of invoking code provided by third parties, particularly in web applications. Nowadays programmers of web applications can choose TypeScript over JavaScript to benefit from static type checking that enables validating calls to local functions or to those provided by libraries. Errors in calls to REST services, however, can only be found at runtime. In this paper, we present SRS, a language that extends the support of static analysis to calls to REST services, with the ability to statically find common errors such as missing or invalid data in REST calls and misuse of the results from such calls. SRS features a syntax similar to JavaScript and is equipped with a rich collection of types and primitives to natively support REST calls that are statically validated against specifications of the corresponding APIs written in the HeadREST language.

1 Introduction

During the last decades web services have become an important building block in the construction of distributed applications. REST web services in particular have become quite popular [16,30]. These services, through specific application programming interfaces, allow consumers to access and manipulate representations of web resources, identified by Unique Resource Identifiers, by using the operations offered by HTTP. Nowadays a very large number of APIs are interfaces of REST services [24] and many software companies expose REST APIs for their services.

Since so many applications are designed to offer REST APIs, the consumption of REST services has become a popular means of invoking code provided by third parties. However, the support available to programmers for writing code that consumes these services is extremely limited when compared to the sort of support offered when invoking external libraries provided by third parties. The practical impact of this problem is attested by a study on a large-scale payment company which concluded that errors in invocations of REST services, related to invalid or missing data, cause most of the failures in API consumer code [2].

The fact that programmers have no way of knowing whether their calls to REST APIs are correct until runtime was identified as one of the four major research challenges for the consumption of web APIs [38]. This state of affairs led to an inter-procedural string analysis proposal to statically check REST calls in JavaScript [37]. The solution checks whether a request to a service conforms

to a given API specification written in OpenAPI[1], and involves checking whether the endpoint targeted by the request is valid and the request has the expected data. Since OpenAPI has severe limitations on what can be expressed about the exchanged data, there are many errors related to invalid or missing data in requests that cannot be addressed by this approach. Moreover, since the models of response data are not taken into consideration, misuse of the result to REST calls cannot be addressed.

This paper presents an approach to API consumer code development based on two new languages: HeadREST—a specification language for REST APIs with a rich type system that supports the specification of semantic aspects of REST APIs—and SRS (short for SafeRESTScript)—a subset of JavaScript equipped with (i) types and strong static analysis and (ii) primitives to natively support REST calls that are statically validated against HeadREST specifications of the corresponding APIs. The validation system of SRS is based on a general-purpose verification tool (Boogie). SRS compiler generates JavaScript code for valid SRS programs, making it easy to use the two languages together and providing a solution for the execution of SRS programs in many different execution environments. The SRS compiler comes in the form of an Eclipse plugin which is publicly available for download. Alternatively, HeadREST and its validator can be exercised directly from a browser [8].

The main contributions of this work are an approach to statically check REST API consumption, and SRS, a type-safe JavaScript-like language, together with its compiler.

The paper starts with a tour through our approach (Section 2) and a brief introduction to THE HeadREST specification language (Section 3). Then, we present the SRS programming language and its validation system (Section 4). The validation system guarantees the detection of errors in REST calls as well as common runtime errors (including null dereference, division by zero and accesses outside arrays bounds). Section 5 presents the evaluation of SRS, Section 6 discusses related work, and Section 7 concludes the paper by pointing towards future work.

2 Overview of the Approach

This section presents the motivation for SRS and walks through our approach by means of an example. First we show how HeadREST allows us to specify REST APIs and capture properties that are important and cannot be expressed in currently available Interface Description Languages (IDL) such as OpenAPI. Then we show how SRS allows programming clients of REST services and how to rely on the compiler to check whether (i) REST calls conform to the specified service interface and (ii) the response data is correctly used, thus avoiding runtime errors.

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

```
1 // Get Location ID from lat/long coordinates
2 function getLocation() {
3     var client_id = '813efa314de94e618290bc8bfcbbb1ac';
4     // URL for API request to find locations
5     var locationUrl =
6         'https://api.instagram.com/v1/locations/search?lat='+LAT
7             +'&lng='+LNG+'&distance=5000&client_id='+client_id;
8     var result = $.ajax({
9         url: locationUrl,
10        dataType: "jsonp",
11        type: "GET",
12    })
13    .done(function(result){
14        //grab first ID from results
15        var locationId = result.data[0].id;
16        return locationId
17    })
18 }
```

Fig. 1. Excerpt of JavaScript code with a call to an endpoint of the Instagram API

2.1 Background

Applications that consume REST APIs communicate with the service provider through calls to the API endpoints, that is to say, by sending requests for the execution of a HTTP method over an URL. The URL of the request identifies a web resource and additionally can provide values for some optional parameters; additional data can be sent in the request body. The service provider sends back a response that carries, among other data, a response status code indicating whether the request has been successfully completed.

Figure 1 shows an excerpt of a JavaScript application [20] that performs a call to an endpoint of the Instagram API to search for locations by geographic coordinates. According to the API documentation [19], this endpoint has several optional parameters, including `lat`, `lng`, and `distance`. The center of the search must be defined and there are three different ways of doing it. Although `lat` and `lng` are optional, if one is used, the other is also required. The distance is optional and its maximum value is 5000. In the success case—signalled by response code 200—the response body is an object with field `data` containing an array of objects with field `id`, among others.

Code that consumes this endpoint may contain different sorts of errors. Calls may not conform to the specified interface: for instance the request may contain a value for `lat` but not for `lng`, or it may contain a value for `distance` that exceeds the maximum value or simply that is not an integer. Moreover, the response data may not be correctly used. This is the case in the example: if the call succeeds, then line 15 accesses a possibly non-existent element of the array in field `data`.

The fact that the model of the response data might depend on the provided input is an additional source of errors. For example, the endpoint in the GitLab API to get all wiki pages for a given project [17] features an optional boolean parameter `with-content` to indicate whether the pages' content must be included

```
specification Instagram

type SearchLocation = (o: {
    ?distance: (x: Integer where x>0 && x<=5000),
    ?lat: Integer,
    ?lng: Integer,
    ?facebook_places_id: String,
    ?foursquare_id: String
} where
    isdefined(lat) <=> isdefined(lng) &&
    (isdefined(lat) || isdefined(facebook_places_id) || isdefined(foursquare_id))
)

type Location = {id: String, name: String, lat: Integer, lng: Integer}

{ !(request in {template: SearchLocation}) }
    get `/locations/search{?distance,lat,lng,facebook_places_id,foursquare_id,client_id}`
{ response.code != 200 }

{ true }
    get `/locations/search{?distance,lat,lng,facebook_places_id,foursquare_id,client_id}`
{ response.code == 200 ==> response in {body: {data: Location[]}} }
```

Fig. 2. A HeadREST specification for an Instagram API endpoint

in the response. Hence, the response body is an array of objects that contains field `content` only when the request has value **`true`** for field `with-content`.

In order to avoid such errors programmers must carefully read the API documentation. The situation is worsened as this sort of documentation tends to be vague and imprecise, even when available in a formal document. Limitations in the expressiveness of existing IDLs—and in particular of OpenAPI, the de facto standard for specifying REST APIs—make programmers resort to natural language for conveying extra information. In the case of the two endpoints considered here this is in fact what happens since most of the properties under discussion are not expressible in the IDLs used for the documentation.

Such state of affairs lead us to develop an approach to support the detection of common errors at compile-time by statically checking that calls match APIs' requirements and that data obtained in the response is correctly used.

2.2 SRS in Action

Our approach builds on two pillars: HeadREST, a specification language for REST APIs, and SRS, a language with an expressive type system for programming the code that consumes REST APIs.

HeadREST resorts to types to express properties of states and of data exchanged in interactions and pairs of pre and post-conditions to express the relationship between data sent in requests and those obtained in responses, as well as the resulting state changes. Two type primitives account for its expressiveness: refinement types, `(x:T where e)`, consisting of values of type `T` that satisfy expression `e`; and a predicate, `e in T`, for checking whether the value of `e` is of type `T`.

```
1  specification "./specs/Instagram.hrest" of "https://api.instagram.com/v1"
2
3  var string client_id = "813efa314de94e618290bc8bfcbbb1ac";
4
5  type Error = {error: string};
6
7  async string|Error getLocation(int lat, int lng) {
8      SearchLocation searchLocation = {lat= lat, lng= lng, distance= 5000, client_id= client_id};
9      Request request = {template = searchLocation};
10     Response result = await get
11             "/locations/search{?distance,lat,lng,facebook_places_id,foursquare_id,client_id}"
12              request;
13     if (result.code != 200) {
14             return {error = "No locations found!"};
15     }
16     return result.body.data[0].id;
17 }

7  async string|Error getLocation(int lat, int lng) {
8      SearchLocation searchLocation = {lat= lat, lng= lng, distance= 50000, client_id= client_id};
9      Request request = {template = searchLocation};
```

Index out of bounds

Expression does not match declared type

Fig. 3. Example of SRS code consuming an Instagram API endpoint

The endpoints exposed by an API, together with their behaviour, are specified by assertions of the form $\{\phi\}\ m\ u\ \{\psi\}$ where ϕ is the pre-condition, m is the HTTP method; u is a URI template and ψ is the post-condition.

Figure 2 shows a specification of the endpoint discussed before. It starts with the declaration of type `SearchLocation` that represents the search data. Note how refinement types capture the endpoint requirements for search data; e.g., line 10 says that fields `lat` and `lng` must be both present or absent; the question marks in front of these two fields indicate that they are optional.

The behaviour of the endpoint is specified by two assertions (lines 16–22). The first says that, if the requirements for the search data sent in the request are not met, then the call does not succeed (the response code is different from `200`). The second assertion says that, if the request is successful, then the response body consists of an array of `Location`, a type defined in line 14.

HeadREST also supports the specification of conditions concerning resources, their representations and their identifiers (see [34] for details). For instance, in the second triple (lines 20–22), we can specify that each `Location` in the response is the representation of a resource that can be individually obtained through the endpoint. Since these properties do not help in avoiding errors in consumer code (individually, clients have no control over the state of the resources), in this paper we limit our presentation to a resource-less version of HeadREST.

Figure 3 shows an SRS program similar in spirit to the JavaScript code in Figure 1. SRS adopts HeadREST types, while featuring direct support for REST operations. We can see that the type checker spots an error in the use of the response data in line 16. The specification ensures that the effective type of `result` in that execution point is `{body:{data:Location[]}}` (Figure 2, line 22). This means that it is safe to access `result.body.data[i].id` only if `i < length(result.body.data)` and, hence, line 16 is incorrect. Would the specification be stronger and, in Figure 2 line 22, read instead `response.code == 200`

Scalar types	G	::=	Integer | String | Boolean | {} | Regexp | URITemplate
Types	T	::=	Any | G | $\{l\colon T\}$ | T[] | $(x\colon T$ where $e)$
Constants	c	::=	n | s | true | false | {} | u | r | null
Expressions	e	::=	x | c | $f(e_1,\ldots,e_n)$ | $e\,?\,e:e$ | e in T | $\{l_1=e_1,\ldots,l_n=e_n\}$
			$\mid e.l$ | $[e_1,\ldots,e_n]$ | $e[e]$ | forall $x\colon T.e$ | exists $x\colon T.e$
Verbs	m	::=	get | put | post | delete
Declarations	D	::=	$\{e\}m\,u\{e\};D$ | ϵ

Fig. 4. The syntax of HeadREST

`==> response in body:{data:(v: Location[] where length(v) > 0)}`, then the program would be valid. Note that the use of type `SearchLocation` in line 8 makes sure that the data sent in the request meets the stated necessary conditions for the request be successful (Figure 2, line 16). The figure also shows the type checker signaling an error if, in line 8, the value given for distance exceeds the maximum value allowed. SRS further supports assert statements that are statically validated. They are useful to check, immediately before a call to an endpoint, that a necessary condition for the request to be successful holds. In the example, we could add `assert(request in {template: SearchLocation})` immediately before line 10.

3 The HeadREST Specification Language

HeadREST was designed to support the specification of REST APIs and to capture important properties that cannot be expressed in currently available interface description languages. This section briefly introduces the resourceless version of HeadREST [34].

The syntax and validation system of HeadREST are influenced by the Dminor language [7]. Extensions and adaptations to Dminor types, expressions and their respective validation rules were adapted to address the specific needs of REST. The syntax of HeadREST is in Figure 4. It assumes a countable set of identifiers (denoted by f or x, y, z), a set of constants (c), a set of labels ($l, l_1, l_2, \ldots$), integer literals (n), string literals (s), a set of URI template literals (u), and a set of regular expression literals (r).

Scalar types include standard Integer, String, Boolean, the REST-specific URITemplate to represent a service endpoint or a group of URI resources and Regexp for regular expressions. Any is the top type. For *types*, we additionally have arrays, refinement types, and the singleton object type $\{l\colon T\}$.

Constants include integer, string, and boolean literals, to which null was added. The null value is of type Any but not of object types. The empty object type, {}, describes empty objects and constitutes the super type of all objects. To inhabit Regexp and URITemplate types, two sorts of literals were added: regular

expressions and URI templates values. Regular expressions form a subset of those in JavaScript. The syntax of URI Templates is conform to RFC-6570 [18].

Expressions include variables and constants, (primitive) function calls, a conditional, arrays and object operations, quantification, and the e in T operator that allows checking whether a given expression e belongs to type T. Useful derived expressions include $\mathsf{isdefined}(e.l) \triangleq e \text{ in } \{l : \mathsf{Any}\}$ and $e \mathbin{\&\&} f \triangleq e \mathbin{?} f : \mathsf{false}$.

Although HeadREST features a small core of types, the type language is quite expressive due to the interplay between refinement types and the in predicate. A few examples of derived types follow, where x is a variable taken freshly.

$$T \mathbin{\&} U \triangleq x\colon \mathsf{Any} \text{ where } (x \text{ in } T \mathbin{\&} x \text{ in } U) \qquad !T \triangleq x\colon \mathsf{Any} \text{ where } !(x \text{ in } T)$$

$$\{?l\colon T\} \triangleq x\colon \{\} \text{ where } x \text{ in } \{l\colon \mathsf{Any}\} \Rightarrow x \text{ in } \{l\colon T\} \qquad \mathsf{Natural} \triangleq x\colon \mathsf{Integer} \text{ where } x \geq 0$$

The operator e in T is essential for the expressiveness of the type system. The intersection, union and negation types are derived using this operator, and these types are the basis for many other derived types. The important multi-field object type can be derived thanks to the intersection type; e.g., $\{l\colon \{\}, m\colon \mathsf{String}\}$ abbreviates $(x\colon \mathsf{Any} \text{ where } (x \text{ in } \{l\colon \{\}\} \mathbin{\&} x \text{ in } \{m\colon \mathsf{String}\}))$ which only uses core types. An important derived type is the optional field type, $\{?l\colon T\}$, asserting that if an object has a field l then its type is T. For example, if e is an expression of type $\{?l\colon \mathsf{Boolean}\}$, then expression e in $\{l\colon \mathsf{Any}\}$ && $e.l$ is valid since, according to its type, if e has field l its type is Boolean and the good formation of $e.l$ is only ensured in this case.

Specifications consist of a collection of assertions (triples), each of which describe part of the behavior of an endpoint. Currently HeadREST supports the four main HTTP verbs: get, post, put and delete. For the specification of pre- and post-conditions three variables are added: request and response that correspond to the call and the reply, and root, the absolute URL of the entry point of the service. The types of the request and response variables are as follows.

$$\mathsf{Request} \triangleq \{\text{location}\colon \mathsf{String}, ?\text{template}\colon \{\}, \text{header}\colon \{\}, ?\text{body}\colon \mathsf{Any}\}$$

$$\mathsf{Response} \triangleq \{\text{code}\colon \mathsf{Integer}, \text{header}\colon \{\}, ?\text{body}\colon \mathsf{Any}\}$$

Algorithmic type checking is based on a bidirectional system, composed of two main relations: one that synthesizes the type of a given expression and one that checks whether an expression is of a given type [13,15,29]. At the intersection of these two relations lies *semantic subtyping*, a relation that establishes that a type T is subtype of a type U when all values that belong to T also belong to U. Types and contexts are translated into first-order logic (FOL) formulae. The thus obtained FOL formulae are then evaluated using an SMT solver. Our implementation uses Z3 [26].

$$
\begin{array}{lrcl}
\text{Constants} & c & ::= & \ldots \mid \mathsf{undefined} \\
\text{Expressions} & e & ::= & \ldots \mid \mathsf{await}^{?}\, m\, u\, e \\
\text{Locations} & w & ::= & x \mid w.l \mid w[e] \\
\text{Statements} & S & ::= & w = e \mid \mathsf{if}\,(e)\,\mathsf{then}\,S\,\mathsf{else}\,S \mid \mathsf{while}\,(e)\,\mathsf{inv}\,e\,S \mid \mathsf{return}\,e \mid S; S \mid \epsilon \\
\text{Declarations} & D & ::= & \mathsf{specification}\,s\,\mathsf{of}\,u \mid \mathsf{var}\,T\,x = e \mid \mathsf{async}^{?}\,T\,x\,(\overline{T\,x})\,\{\overline{T\,x = e}; S\} \\
\text{Programs} & P & ::= & D; P \mid \epsilon
\end{array}
$$

Fig. 5. The syntax of SRS (extends Figure 4)

4 The SRS Programming Language

The SRS language (a shorthand for SafeRestScript) is a type-safe variant of JavaScript with direct support for REST calls. It was designed to be, at the syntactic level, as close as possible to JavaScript. It transpiles to JavaScript, making it easy to integrate REST API consumer code written in SRS with JavaScript code, namely code of web applications for manipulating the DOM.

Compared with other typed extensions of JavaScript, such as TypeScript [6], the main novelty of SRS is the incorporation of refinement types, the in-type predicate and, most importantly, REST endpoints as external functions. More precisely, a REST endpoint is seen as an impure, external function that receives a value of type Request, possibly changes a global resource set state, and then returns a result of type Response. REST calls are then just calls to such functions. Additional properties of these endpoints-as-functions, namely their specific return type, are inferred from the HeadREST specification of the REST API endpoints. Each triple in the specification specifies a relation between the input (the request) and the output (the response) of an endpoint: if the request meets the pre-condition, then the response meets the pos-condition. From triple $\{\phi\}\, m\, u\, \{\psi\}$, the return type of endpoint-as-function $m\, u$ is extracted as $\{r\colon \mathsf{Request}\ \ \mathsf{where}\ \ \phi \Rightarrow \psi\}$. Note that endpoints-as-functions are, hence, total: they accept any input of type Request, even those that do not meet the pre-condition of any of their triples (in the vein of Hoare Logic [21] and as opposed to that of Design by Contract [25]). JavaScript is single threaded and, hence, function calls that take time to execute should ideally be executed asynchronously. REST calls fall into this category; SRS supports asynchronous in addition to synchronous REST calls.

SRS adopts the HeadREST type system, not only for its support for REST operations, but also to provide precise static type checking. In SRS, each variable is declared with a type that restricts the values that can be assigned to the variable. Each variable also features an *effective type* that corresponds to the set of values the variable may have at a given point in a program. The effective type changes with program flow, but is necessarily a subtype of the declared type.

4.1 Syntax

The syntax of SRS, presented in Figure 5, extends that of HeadREST in Figure 4. The language includes a new *constant* undefined. Functions that return undefined are of type void, an abbreviation of (x: Any where x == undefined).

At the level of *expressions*, SRS introduces REST calls $m\,u\,e$, composed of an HTTP method m (see Figure 4), an URI template literal u describing the relative URL of the target resource, and an expression e that should evaluate to a value of type Request. The endpoint needs to be specified in the SRS specification imported by the program. Functions can be declared with the async keyword; calls to these functions are asynchronous while REST calls are asynchronous if they are preceded by keyword await.

Statements include variable assignment. The left hand side w of an assignment statement (a *location*) is a variable x, an object field $w.l$, or a position in an array $w[e]$. An assignment can thus update a specific element of an object or an array. Moreover, statements include conditional statements, while loops, and return statements. Loops may declare an invariant, i.e., an expression that is true at loop entry and after each loop iteration. Invariants are sometimes necessary to prove that certain expressions have the right type, for instance, whether the effective type of the expression used in a return statement matches the return type of the function. Statement return abbreviates return undefined.

An SRS *program* is a sequence of *declarations*: import clauses, global variable and function declarations. The implementation of SRS further supports type abbreviations in the form of type $x = T$. Function definitions are composed of a return type T, the function name f, a comma-separated list of parameters with their respective types $\overline{U\,x}$, and the function body. In order to simplify variable scope validation, the body opens with the declaration and initialization of all local variables: $\overline{V\,y = e}$ is a semi-colon-separated list of variable declarations. The initialization is mandatory since some types, such as refinement types, may not have a default value. The function's body consists of a statement S that defines the control flow and the return value.

4.2 Type Checking

Statically type checking SRS programs is a major challenge given the rich type system of SRS and global imperative variables. It requires flow-sensitive typing (the effective type of an expression depends on its position in the program).

SRS programs are translated into verification conditions, i.e., logical formulae whose validity entails the correctness of the program. Following a popular approach initiated by Spec# [4], these conditions are not generated directly but instead obtained through a translation into Boogie [5], an intermediate language for program verification. Once a SRS program is translated into a Boogie program, it is up to the Boogie validator to generate the verification conditions and, resorting to an SMT solver, verify whether they hold.

At the basis of the translation is an axiomatization of the typing relation that is inspired by Whiley [28]. Values and types are modelled as sets. All SRS values,

independently of their type, belong to the Boogie type `Value`. For each type, we introduce functions and axioms that define its subset of values. More concretely, given a type `X` (for example, **`Integer`**) and its internal representation `Y` in Boogie (**`int`**, in the example), the base functions and axioms are the following.

```
function isX(Value) returns (bool);
function toX(Value) returns (Y);
function fromX(Y) returns (Value);
axiom (forall y: Y :: isX(fromX(y)));
axiom (forall y: Y :: toX(fromX(y)) == y);
axiom (forall v: Value :: isX(v) ==> fromX(toX(v)) == v);
```

Function `isX` checks whether a value belongs to type `X` and returns a Boogie boolean. Function `toX` converts the Boogie value to its internal representation `Y`, and `fromX` performs the inverse operation. The axioms define the properties of the functions. The first asserts that all values constructed from type `Y` belong to type `X`. The second and third axioms assert that `toX` and `fromX` are inverse functions. More complex types, such as arrays and objects, are represented by Boogie maps and require the introduction of additional functions and axioms.

Functions `isX`, `toX` and `fromX` are used for defining the translation of expressions and the predicate that checks whether the value of an expression is of a given type. This is illustrated below in simple cases: the translation of an SRS integer literal and an array access, and the predicate for the integer type.

$$\mathbf{V}[\![n]\!] = \mathsf{fromInt}(n) \qquad \mathbf{F}[\![\mathsf{int}]\!](e) = \mathsf{isInt}(e)$$
$$\mathbf{V}[\![e_1\,[e_2]]\!] = \mathsf{getIndexValue}(\mathbf{V}[\![e_1]\!], \mathsf{toInt}(\mathbf{V}[\![e_2]\!]))$$

The translation of SRS to Boogie is based on the collection of functions presented below. We discuss some cases that convey the main ideas of how the translation works. The full set of rules is available in the extended version of this paper [9].

$$\begin{aligned}
\mathbf{V}[\![e]\!] &\equiv \text{Boogie expression of type } \texttt{Value} \text{ that represents expression } e \\
\mathbf{F}[\![T]\!] &\equiv \text{Boogie predicate that checks whether an expression is of type } T \\
\mathbf{V}^*[\![e]\!]_x &\equiv \text{Sequence of Boogie statements that validates expression } e \\
&\quad\ \text{and places the corresponding Boogie expression in variable } x \\
\mathbf{W}[\![T]\!] &\equiv \text{Sequence of Boogie statements that validates type } T \\
\mathbf{B}[\![S]\!] &\equiv \text{Boogie statement that represents statement } S \\
\mathbf{B}[\![D]\!] &\equiv \text{Boogie declaration that represents declaration } D
\end{aligned}$$

The translation of REST calls and of specification triples to Boogie are the most interesting elements of the translation, as they accomplish the view of endpoints-as-functions discussed before. REST calls are translated to Boogie using a function, named restCall, that receives as parameters the REST method, the translation of a string u' representing the URI template relative path u, and the request object, and returns the response object. Each specification triple is

translated into an axiom relating the return value of restCall with the request call argument as follows.

$$\begin{aligned}\mathbf{B}[\![\{e_1\}m\,u\{e_2\}]\!] = \mathbf{axiom}\ (\mathbf{forall}\ \mathsf{request}\colon \mathsf{Value},\ \mathsf{response}\colon \mathsf{Value} ::\\ \mathsf{restCall}(m, \mathbf{V}[\![u']\!], \mathsf{request}) == \mathsf{response} \wedge\\ \mathbf{V}[\![e_1]\!] == \mathbf{V}[\![\mathsf{true}]\!] \Rightarrow \mathbf{V}[\![e_2]\!] == \mathbf{V}[\![\mathsf{true}]\!])\end{aligned}$$

The translation of REST calls is defined by the following rules:

$$\begin{aligned}\mathbf{V}^*[\![m\,u\,e]\!]_x &= \mathbf{V}^*[\![e]\!]_y;\ \mathbf{assert}\,\mathbf{F}[\![\mathsf{Request}]\!](y);\\ &\quad x := \mathsf{restCall}(m, \mathbf{V}[\![u]\!], y);\ \mathbf{assume}\,\mathbf{F}[\![\mathsf{Response}]\!](x)\\ \mathbf{V}^*[\![\mathsf{await}\,m\,u\,e]\!]_x &= \mathbf{V}^*[\![m\,u\,e]\!]_x;\ \mathbf{havoc}\,g_1,\ldots,g_p\end{aligned}$$

Expression e in a synchronous REST call is validated and placed in a fresh variable y. Then an **assert** checks whether y is of type Request. Function restCall is called and its response is stored in variable x. The response is assumed to be of type Response. Note that when the request does not meet the pre-condition of any triple for the target endpoint, the axiomatization of restCall does not ensure anything about the response; it is only known that it belongs to type Response. In asynchronous REST calls, the execution is suspended and, when resumed, the global variables may have changed. This is captured by the **havoc** statement, which assigns arbitrary values to variables (while respecting their declared types).

The type validation takes into account that types may contain expressions by descending the abstract syntax tree of types. The most important rule is the rule for where types (y, z are variables taken freshly).

$$\mathbf{W}[\![(x\colon T\ \mathsf{where}\ e)]\!] = \mathbf{W}[\![T]\!]; \mathbf{assume}\,\mathbf{F}[\![T]\!](y); \mathbf{V}^*[\![e[y/x]]\!]_z; \mathbf{assert}\,\mathbf{F}[\![\mathsf{Boolean}]\!](z)$$

We complete this brief presentation by addressing the translation of global variable declaration and functions.

$$\begin{aligned}&\mathbf{B}[\![T\ x = e]\!] = \mathbf{var}\ x : \mathsf{Value}\ \mathbf{where}\ \mathbf{F}[\![T]\!](x); \mathbf{V}[\![T\ f()\ \{\,\mathsf{return}\ e\,\}]\!]\\ &\mathbf{B}[\![T\ f\ (\,T_1\,x_1, \ldots, T_n\,x_n\,)\ \{\,U_1\ y_1 = e_1\,;\ \ldots\ ;\ U_m\ y_m = e_m\,;\ S\,\}]\!] =\\ &\mathbf{procedure}\ f(x_1 : \mathsf{Value}, \ldots, x_n : \mathsf{Value})\ \mathbf{returns}\ (\mathsf{result} : \mathsf{Value})\\ &\quad \mathbf{requires}\ \mathbf{F}[\![T_1]\!](x_1) \wedge \cdots \wedge \mathbf{F}[\![T_n]\!](x_n);\ \ \mathbf{ensures}\ \mathbf{F}[\![T]\!](\mathsf{result});\\ &\quad \mathbf{modifies}\ g_1, \ldots, g_p;\\ &\{\\ &\quad \mathbf{var}\ y_1, \ldots, y_m,\ w_1, \ldots, w_n : \mathsf{Value}; w_1 := x_1; \ldots; w_n := x_n;\\ &\quad \mathbf{W}[\![T_1]\!]; \ldots; \mathbf{W}[\![T_n]\!]; \mathbf{W}[\![U_1]\!]; \ldots; \mathbf{W}[\![U_m]\!]; \mathbf{W}[\![T]\!]\\ &\quad \mathbf{B}[\![y_1 = e_1\,;\ \ldots\ ;\ y_m = e_m\,; S[w_1/x_1] \ldots [w_n/x_n]; \mathsf{return}]\!]\\ &\}\end{aligned}$$

In the first rule, the declared type is captured by a Boogie **where** clause while the initialization is ignored as it is not relevant: whenever a procedure is called,

nothing can be assumed about any global variable besides its declared type. The validation of T and e is achieved via an additional, dummy, procedure f.

In the second rule, the immutability of procedure parameters in Boogie requires the declaration of new variables to use instead of the parameters in the function body. The **requires** clause checks whether the arguments belong to the parameters types and the **ensures** clause checks whether in all returning points of the procedure the result of the function matches the function type. The **modifies** clause asserts that all global variables can be modified by the procedure. The body of the procedure makes the validation of parameters types T_i, local variables types U_k and the return type T. The validation order allows that the validity of T and U_k depend on T_i and the validity of T_i depend on the T_j, for $j < i$, as in $\{x\colon \mathsf{int\ where}\ x > b/a\}\ f(\{x\colon \mathsf{int\ where}\ x! = 0\}\ a, \mathsf{int}\ b)\{...\}$.

4.3 Transpiling to JavaScript

Valid programs are transpiled to JavaScript. The translation of REST calls is achieved by calling auxiliary functions, one for synchronous and another by asynchronous calls. The URL to the call is the expansion of the URI template; its parameters are defined by the field template of the request object. The expansion follows the RFC 6570 [18], only for the level of URI templates supported by SRS. The content-type JSON is added to the request headers, so objects sent and received in the body are ensured to be of JSON format, and therefore having a direct translation to JavaScript objects. The calls use `XMLHttpRequest`, an object that is supported by all browsers and devices.

5 Evaluation

This section addresses the evaluation of our approach. Ideally, we would like to compare the bug finding efficacy of our approach in "real code" with that of Wittern at al. [37], the unique approach to statically checking REST calls that we are aware of. However, this turned out not feasible, since translating JavaScript code into SRS requires annotating all the libraries used and/or write adaptors that monitor the interface with libraries.

In this way, to evaluate our approach, we used HeadREST to specify a variety of REST APIs and SRS to write and validate programs that exercise the different elements of the language while consuming REST APIs. The goal is to evaluate *to what degree can SRS be used in examples which include complex REST calls that can be found in real examples.*

We used SRS to write programs that consume publicly available APIs and do not require authentication, including PetStore[2] and DummyAPI[3] as well as programs that consume real-world off-the-shelf services such as Instagram, GitHub and GitLab. Since API calls in SRS are checked against HeadREST

[2] https://petstore.swagger.io
[3] http://dummy.restapiexample.com

specifications, we also developed HeadREST specifications for the chosen APIs describing the behaviour of the relevant endpoints. In what follows we provide details about three of these case studies. The complete examples are available in the supplementary material [8].

Instagram We developed in SRS a solution alternative to the JavaScript function in Section 2. The application allows users to find Instagram photos by tag or location and calls the different endpoints of the Instagram API which supports search for (i) locations by geographic coordinate, (ii) photos by location and (iii) photos by tag. The solution is based on a SRS program defining asynchronous functions for calling the API, similar to that presented in Figure 3. These functions are available in the generated JavaScript code and used by the program that manipulates the DOM. We additionally developed a program for showing the recent comments on media for a user, given its identifier, which requires to call three other endpoints: one to get the ids of recent media, another to get the comments for each of them and a third one to get information about the user. Both programs use the same specification with the behaviour of the six endpoints.

GitHub We developed an SRS program that offers a function `getUserById(int id)` to obtain a GitHub user given its `id` with return type `(u: User where u.id == id)|NotFoundError`. Since the GitHub REST API does not have an endpoint that supports this operation (to get the representation of individual users, one needs to provide the username), our program sends a `GET` request to `/users?since=id-1` if `id` is a positive integer. According to the API documentation, this endpoint lists all users, in the order that they signed up on GitHub. Retrieval is by pagination: each call retrieves a sublist of all users. The start of the sublist is defined by the optional parameter `since`. If case the parameter is not present, then its value is assumed to be zero. In the HeadREST specification of GitHub we were able to precisely express this behaviour. One of the assertions included in the specification states that if the request provides a value for `since` that is a natural number, then the array of users provided in the response body starts with a user whose `id` is equal to `since+1`. This assertion is essential to prove that if a user is obtained, it has the id provided in the function argument and, hence, that the return type of the function is valid.

The endpoint `/get users{?since}` can also be used for searching for an user with certain characteristics. We used it to define a function `getSiteAdmin()` with return type `(u: User where u.site_admin)|AdminNotFound` that searches over the GitHub users to find an administrator. The search code gets the various pages of users and stops when one of them contains an user with admin privileges, or when no admin was found on all pages, in which case `AdminNotFound` is returned. The fact that the function type checks ensures that the returned user representation (if any) is an indeed an administrator.

In GitHub each user has a set of repositories, and each repository has a set of collaborators and a list of commits, each with its author. We programmed a function that gets the collaborators of a repository that did not contribute

	HeadREST				SRS			
	#EndP	#Types	LOC	Check (s)	#EndP	#Func	LOC	Check (s)
Instagram#app1	6	9	225	1.3	3	4	82	1.5
Instagram#app2	6	9	225	1.3	3	3	65	1.8
GitHub	5	9	93	0.8	5	3	86	1.3
GitLab	10	20	435	1.7	8	10	250	50.5

Table 1. Case studies of consuming REST APIs with SRS

to a project, i.e., did not make a commit. The function crosses the information obtained in two different endpoints: one for retrieving the collaborators of a given repository for a given user and another for retrieving the list of commits of the repository. As the repository may be private, the function receives a key that must give authorization to access the repository information, and that is added to the request header.

GitLab is the Git manager used by our students to develop their course projects. We wrote functions that automate tasks we recurrently perform manually. For instance, we programmed a function to remove a user from all projects owned by another user. This function uses three endpoints, one of them involving request and response types particularly large—the request has more than 10 optional parameters and the response body is an array of an object type with more than 30 fields, several of them also objects. The type of the response is used, for instance, to validate the expression `response.body[i].namespace.name` that occurs in the body of the function. Another interesting example is the function `getWikisFromProject(string token, int|string id, boolean withContent)` we defined in SRS to get the wikis from a project, identified by its integer id or a string that is the URL-encoded path of the project (and, hence, of type `int|string`). The function uses an endpoint that features an optional parameter to indicate whether the response should contain the content of the wikis. The behaviour of GitLab at this endpoint was specified in HeadREST as shown below and allows the SRS validator to find errors in accesses to `response.body[i].content`, such as when the SRS code does not guarantee that accesses are performed only if the value sent in `request.template.with_content` is true.

```
{ request in {template: {id: String|Integer, ?with_content:
    Boolean}} }
       get '/projects/{id}/wikis{?private_token,with_content}'
{ (response.code == 200 ==> response in {body: Wiki[]}) &&
  (response.code == 200 && request.template in {with_content:
    Boolean}
    && request.template.with_content ==>
        response.body in (Wiki & {content:String})[]) }
```

Table 1 presents additional information about the three case studies. The first group of columns addresses HeadREST specifications and shows the number

of endpoints that were specified, the number of types that were defined, the number of lines and the validation time, in seconds. The second group of columns, which addresses SRS client programs, shows the number of endpoints that were consumed, the number of functions that were defined, the number of lines of code and the validation time, in seconds. The validation the time of SRS programs presented in the table does not consider the validation time of the specification. Benchmarking was performed on a machine with an Intel Core i7-7700HQ CPU, with 2.80 GHz and 16 GB of RAM memory, under Windows 10. The times reported are the average of three runs.

Overall, these examples demonstrate that HeadREST supports the specification of a variety of API endpoints found in real examples and is able to capture important properties of these endpoints that were previously available only in natural language. During the development of the client programs we could witness that the formalisation of properties allowed SRS to find all sort of errors in our code, in particular, errors in the invocations of the underlying services (invalid or missing data in the requests or use of incorrect URLs) and errors in the use of the data received in the response. We also noted that were we programming the same client code in JavaScript, most of the errors we made would not be found by Wittern at al. [37]. On the one hand, errors caused by invalid data in the requests were often caused by restrictions on data that are simply not expressible in OpenAPI. On the other hand, several errors lied in the usage of the data received in the response, a type of error that is not addressed by the analysis performed by the tool.

In terms of performance, we witness what is also evident in the results in Table 1: the complexity of the types involved in REST calls significantly slows the validation process when the correctness of the code strongly depends on these types. This problem can be alleviated by placing functions whose validation is too demanding in separated source files. Because the validator ignores files that have not changed, these functions do not need to be validated again if they have not changed.

6 Related Work

Static verification of JavaScript code has been the main research topic for client-side coding in the last few years [31]. Nevertheless, research concerning the verification of consumer code of REST APIs for JavaScript-like client-side languages is slim and the solutions proposed tend to be quite limited.

Solutions for helping finding bugs in scripts come in the form of a varied set of languages and tools. JSHint [22] scans JavaScript code for suspicious usage; Thiemann [32] and Anderson et al. [1] propose type system for subsets of JavaScript; TypeScript [6], Dart [11] and Flow [14] are languages that were developed with the goal of statically detecting type-related errors in JavaScript-like languages. Languages such as Dependent JavaScript [10] and Refined TypeScript [35] incorporate sophisticated type systems, but the power of the e in T predicate and

semantic subtyping (supported by SRS) seems to be particularly suited for programming REST clients. Whiley [28] is a programming language that features a rich type system and flow typing; it uses Boogie only to check the verification conditions [33]. Contrary to SRS, neither of these solutions specifically addresses REST calls.

TypeScript$_{\text{IPC}}$ [27] extends TypeScript with the ability to describe the presence or absence of properties in objects, a feature that HeadREST and SRS can easily describe and for which a derived predicate isdefined was introduced (cf. Section 3). Like all the languages discussed above, TypeScript$_{\text{IPC}}$ does not provide explicit support for REST calls.

The tool by Wittern et al. [37], discussed in the introduction, statically checks web API requests in JavaScript code, focusing on ajax requests made via jQuery[4]. The tool uses a field-based call graph to make the necessary string analyses on the JavaScript method calls and is able to check whether calls to endpoints match a valid URI template in the API specification and the request has the expected data. Such errors are easier to check in SRS since the construction of URIs is limited to URI template instantiation (thus ruling out the construction of new URIs via string operations such as concatenation). In contrast, the verification supported by SRS that the request has the expected data is beyond reach of Wittern et al. for the rich, non-OpenAPI, data definitions. RESTyped Axios [12] is a client-side tool that verifies REST calls in TypeScript against RESTyped specifications, with requests made via the Axios framework [3]. RESTyped allows to define strongly-typed routes and Axios checks at compile time whether the URLs are valid and whether the types of the members passed on requests and accessed on responses correspond to the ones declared in the specification. These two approaches fail to detect many of the defects at the reach of SRS, including those related to complex restrictions on input data of REST calls (not expressible in the adopted specification languages) or the misuse of the return data.

Whip [36] is a contract system for services that uses a dependent type system to monitor services at runtime and check whether they respect their advertised interfaces. Whip offers a high-order contract language that, similarly to HeadREST, addresses the lack of expressiveness of IDLs to capture non-trivial properties that can be found in the documentation of popular services. Whip focus is on the specification of properties that cross-cut more than one service (e.g., properties that describe how a client of one system should use a reply to interact with another) and, by using contracts, addresses the specification of the expectations and promises of a service to other services.

7 Conclusion

We present a framework for statically checking code that consumes APIs. Relevant aspects of APIs are described with HeadREST, a specification language

[4] https://api.jquery.com

featuring refinement types and semantic subtyping. The consumer code itself is written in SRS, a variant of JavaScript with explicit primitives for synchronous and asynchronous REST calls. HeadREST specifications are validated by resorting to an SMT solver to discharge semantic subtyping goals. API consumer code is checked via a translation to Boogie. We validate our approach by writing in SRS various benchmarks from the literature. We further report on three case studies of consumer code for popular APIs (Instagram, GitHub, and GitLab).

Much remains to be done; we sketch a few ideas for future work. The lack of references is the most relevant difference between SRS and JavaScript. Introducing references in objects and arrays is not trivial and adds additional complexity to the Boogie translation. Dafny [23] devised a clever solution using object references in its translation to Boogie, but the technique does not carry straightforwardly to refinement types. A preliminary experience showed that this extension substantially increases the validation time.

Since SRS compiles to JavaScript, programmers may take advantage of its standard libraries. To use JavaScript functions in SRS code, their signatures are required. We plan to address this issue, possibly by following the TypeScript approach, that is, by introducing declaration files where external JavaScript can be declared and annotated with the SRS types so they can be used in SRS code.

HeadREST specifications may feature inconsistent triples. This aspect does not influence the validation of HeadREST specifications, since each triple is validated independently, but it can affect the validation of SRS programs. Specifications featuring inconsistent triples induce inconsistent Boogie axiomatizations, allowing programs with typing errors to be validated. It is therefore important to detect inconsistent HeadREST specifications.

Acknowledgements This work was supported by FCT through the LASIGE Research Unit, ref. UIDB/00408/2020, and by project Confident ref. PTDC/EEI-CTP/4503/2014.

References

1. Anderson, C., Giannini, P., Drossopoulou, S.: Towards type inference for JavaScript. In: ECOOP 2005 - Object-Oriented Programming, 19th European Conference. Lecture Notes in Computer Science, vol. 3586, pp. 428–452. Springer (2005). https://doi.org/10.1007/11531142_19
2. Aué, J., Aniche, M.F., Lobbezoo, M., van Deursen, A.: An exploratory study on faults in web API integration in a large-scale payment company. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE. pp. 13–22. ACM (2018). https://doi.org/10.1145/3183519.3183537
3. Axios: Promise based HTTP client for the browser and node.js, https://github.com/axios/axios
4. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: The Spec# experience. Communications of the ACM **54**(6), 81–91 (June 2011)

5. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Formal Methods for Components and Objects, 4th International Symposium, FMCO. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (2005). https://doi.org/10.1007/11804192_17
6. Bierman, G.M., Abadi, M., Torgersen, M.: Understanding TypeScript. In: ECOOP 2014 - Object-Oriented Programming. Lecture Notes in Computer Science, vol. 8586, pp. 257–281. Springer (2014). https://doi.org/10.1007/978-3-662-44202-9_11
7. Bierman, G.M., Gordon, A.D., Hritcu, C., Langworthy, D.E.: Semantic subtyping with an SMT solver. J. Funct. Program. **22**(1), 31–105 (2012). https://doi.org/10.1017/S0956796812000032
8. Burnay, N., Ferreira, F., Lopes, A., Martins, F., Medeiros, F., Santos, T., Vasconcelos, V.T.: Communication contracts for distributed systems development, http://rss.di.fc.ul.pt/confident
9. Burnay, N., Lopes, A., Vasconcelos, V.T.: SafeRESTScript: Statically checking REST API consumers. arXiv:2007.08048 (2020), http://arxiv.org/abs/2007.08048
10. Chugh, R., Herman, D., Jhala, R.: Dependent types for JavaScript. In: Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA. pp. 587–606. ACM (2012). https://doi.org/10.1145/2384616.2384659
11. Dart: The Dart programming language, https://www.dartlang.org/
12. Dezfuli-Arjomandi, A.: Introducing RESTyped: End-to-end typing for REST APIs with TypeScript (2017), https://blog.falcross.com/introducing-restyped-end-to-end-typing-for-rest-apis-with-typescript/
13. Dunfield, J., Krishnaswami, N.R.: Complete and easy bidirectional typechecking for higher-rank polymorphism. In: ACM SIGPLAN International Conference on Functional Programming, ICFP. pp. 429–442. ACM (2013). https://doi.org/10.1145/2500365.2500582
14. Facebook: Flow: A static type checker for JavaScript, https://flow.org/
15. Ferreira, F., Pientka, B.: Bidirectional elaboration of dependently typed programs. In: Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming. pp. 161–174. ACM (2014). https://doi.org/10.1145/2643135.2643153
16. Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture. ACM Trans. Internet Techn. **2**(2), 115–150 (2002). https://doi.org/10.1145/514183.514185
17. GitLab: GitLab OpenAPI documentation, https://gitlab.com/gitlab-org/gitlab-foss/blob/swagger-api/doc/api/wikis.md
18. Gregorio, J., Fielding, R.T., Hadley, M., Nottingham, M., Orchard, D.: URI template. RFC **6570**, 1–34 (2012). https://doi.org/10.17487/RFC6570
19. Harmony, A.: Instagram API, https://apiharmony-open.mybluemix.net/public/apis/instagram#get_locations_search
20. Herman, M.: Instagram search app, https://github.com/mjhea0/thinkful-mentor/blob/master/frontend/instagram-search/app.js
21. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259
22. JSHint: JSHint, a static code analysis tool for JavaScript, https://jshint.com/about/

23. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20
24. Levin, G.: The rise of REST API (2015), https://blog.restcase.com/the-rise-of-rest-api/
25. Meyer, B.: Object-Oriented Software Construction, 2nd Edition. Prentice-Hall (1997)
26. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
27. Oostvogels, N., Koster, J.D., Meuter, W.D.: Static typing of complex presence constraints in interfaces. In: 32nd European Conference on Object-Oriented Programming, ECOOP. LIPIcs, vol. 109, pp. 14:1–14:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.ECOOP.2018.14
28. Pearce, D.J., Groves, L.: Whiley: A platform for research in software verification. In: Software Language Engineering - 6th International Conference, SLE. Lecture Notes in Computer Science, vol. 8225, pp. 238–248. Springer (2013). https://doi.org/10.1007/978-3-319-02654-1_13
29. Pierce, B.C., Turner, D.N.: Local type inference. In: POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998. pp. 252–265. ACM (1998). https://doi.org/10.1145/268946.268967
30. Richardson, L., Ruby, S.: RESTful web services - web services for the real world. O'Reilly (2007)
31. Sun, K., Ryu, S.: Analysis of JavaScript programs: Challenges and research trends. ACM Comput. Surv. **50**(4), 59:1–59:34 (2017). https://doi.org/10.1145/3106741
32. Thiemann, P.: Towards a type system for analyzing JavaScript programs. In: Programming Languages and Systems, 14th European Symposium on Programming, ESOP. Lecture Notes in Computer Science, vol. 3444, pp. 408–422. Springer (2005). https://doi.org/10.1007/978-3-540-31987-0_28
33. Utting, M., Pearce, D.J., Groves, L.: Making Whiley Boogie! In: Integrated Formal Methods - 13th International Conference, IFM. Lecture Notes in Computer Science, vol. 10510, pp. 69–84. Springer (2017). https://doi.org/10.1007/978-3-319-66845-1_5
34. Vasconcelos, V.T., Martins, F., Lopes, A., Burnay, N.: HeadREST: A specification language for RESTful APIs. In: Models, Languages, and Tools for Concurrent and Distributed Programming - Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday. Lecture Notes in Computer Science, vol. 11665, pp. 428–434. Springer (2019). https://doi.org/10.1007/978-3-030-21485-2_23
35. Vekris, P., Cosman, B., Jhala, R.: Refinement types for TypeScript. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI. pp. 310–325. ACM (2016). https://doi.org/10.1145/2908080.2908110
36. Waye, L., Chong, S., Dimoulas, C.: Whip: higher-order contracts for modern services. PACMPL **1**(ICFP), 36:1–36:28 (2017). https://doi.org/10.1145/3110280
37. Wittern, E., Ying, A.T.T., Zheng, Y., Dolby, J., Laredo, J.A.: Statically checking web API requests in JavaScript. In: Proceedings of the 39th International

Conference on Software Engineering, ICSE. pp. 244–254. IEEE / ACM (2017). https://doi.org/10.1109/ICSE.2017.30
38. Wittern, E., Ying, A.T.T., Zheng, Y., Laredo, J.A., Dolby, J., Young, C.C., Slominski, A.: Opportunities in software engineering research for web API consumption. In: 1st IEEE/ACM International Workshop on API Usage and Evolution, WAPI@ICSE. pp. 7–10. IEEE Computer Society (2017). https://doi.org/10.1109/WAPI.2017.1