UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Types to the Rescue: Verification of REST APIs Consumer Code

Nuno Miguel Pereira Burnay

Mestrado em Engenharia Informática
Especialização em Engenharia de Software

Dissertação orientada por:
Prof. Doutor Vasco Manuel Thudichum de Serpa Vasconcelos
Prof. Doutor Maria Antónia Bacelar da Costa Lopes

2019

Agradecimentos

Os maiores agradecimentos vão para quem mais contribuiu para o sucesso desta dissertação: o prof. Vasco Vasconcelos e a prof. Antónia Lopes. Em primeiro lugar, por terem-me proposto este tema e aceitado trabalhar comigo. De seguida, por terem-me orientado maravilhosamente ao longo de quase um ano, corrigindo os meus erros e dando-me liberdade para errar. Por fim, por terem-me ajudado a escrever esta tese, atenuando as minhas lacunas na escrita. Sem os meus orientadores teria sido um ano muito entediante!

Agradeço também a todos os colegas do LASIGE que tive de aturar durante o ano inteiro. Quase conseguiram que eu não entregasse a tese a tempo. Sem vocês isto teria sido muito menos emocionante.

Estendo este agradecimento a todos os colegas e professores que me auxiliaram ao longo do meu percurso académico. Também aos meus (poucos mas bons) amigos que estiveram ao meu lado antes de ingressar na faculdade. Foram certamente quem mais contribuiu indiretamente para esta tese.

Por fim, um agradecimento especial a todos aqueles que colocaram pedras no meu caminho. Guardeias todas e fiz-me Mestre.

Aos amigos que me ajudaram

Aos inimigos que me motivaram

Resumo

As arquiteturas de software são fundamentais para o desenvolvimento de um software fiável, escalável e com uma fácil manutenção. Com a criação e crescimento da internet, surgiu a necessidade de criar padrões de software que permitam trocar informação neste novo ambiente. O protocolo SOAP e a arquitetura REST são, dos padrões que emergiram, os que mais se destacaram ao nível da utilização. Durante as últimas décadas, e devido ao grande crescimento da World Wide Web, a arquitetura REST tem se destacado como a mais importante e utilizada pela comunidade.

REST (*Representational State Transfer*) retira partido das características do protocolo HTTP para descrever as mensagens trocadas entre clientes e servidores. Os dados na arquitetura REST são representados por recursos, que são identificados por um identificador único (p.e. URI) e que podem ter várias representações (em vários formatos), que são os dados concretos de um recurso. A interação com os recursos é feita usando os métodos HTTP: *get* para obter um recurso, *post* para adicionar um novo recurso, *put* para fazer uma atualização de um recurso, *delete* para remover um recurso; entre outros, sendo estes os principais para aplicações CRUD.

As aplicações RESTful, isto é, aplicações que fornecem os seus serviços através da arquitetura REST, devem ser claras na especificação dos seus serviços de forma a que os seus clientes possam utilizá-las sem erros. Para tal, existem várias linguagens de especificação de APIs REST, como a Open API Specification ou a API Blueprint, no qual é possível descrever formalmente as várias operações fornecidas pelo serviço, como o formato dos pedidos de cada operação e as respetivas respostas. No entanto, estas linguagens apresentam uma limitação nas condições formais que se pode colocar nos parâmetros dos pedidos e no impacto que estes têm no formato e conteúdo da resposta.

Deste modo, foi introduzida uma nova linguagem de especificação de aplicações REST, HeadREST, onde é adicionada a expressividade necessária para cobrir as lacunas das outras linguagens. Esta expressividade é introduzida com a utilização de tipos refinados, que permitem restringir os valores de um determinado tipo. Adicionalmente, é introduzida também uma operação que permite verificar se uma determinada expressão pertence a um determinado tipo. Em HeadREST, cada operação é especificada usando uma ou mais asserções. Cada asserção é composta por um método HTTP, um URI template da operação, uma pré-condição que define as condições onde esta operação é aceite,

e uma pós-condição que estabelece os resultados da operação se a pré-condição for comprida. Deste modo, estas condições permitem expressar os dados enviados nos pedidos e a receber na resposta, assim como expressar o estado do conjunto de recursos antes e depois do pedido REST.

Devido à utilização de tipos refinados não é possível resolver sintaticamente a relação de subtipos na validação de uma especificação HeadREST. Deste modo, é necessária uma abordagem semântica: a relação de subtipos é transformada em fórmulas de lógica de primeira ordem, e depois é utilizado um SMT *solver* para resolver a formula e, consecutivamente, resolver a relação de subtipos.

Por outro lado, é também importante garantir que as chamadas às APIs REST cumprem as especificações das mesmas. As linguagens de programação comuns não conseguem garantir que as chamadas a um serviço REST estão de acordo com a especificação do serviço, nomeadamente se o URL da chamada é válido e se o pedido e resposta estão bem formados ao nível dos valores enviados. Assim, um cliente só percebe se as chamadas estão bem feitas em tempo de execução. Existem poucas soluções para análise estática deste tipo de chamadas (RESType é um raro exemplo) e tendem a ser limitadas e a depender de um único tipo de linguagem de especificação. Para além disso, os clientes de serviços REST tendem a ser maioritariamente desenvolvidos em JavaScript, que possui uma fraca análise estática, o que potencializa ainda mais o problema identificado.

Numa primeiro passo para tentar resolver este problema desenvolveu-se a linguagem SafeScript, que se caracteriza por ser um subconjunto do JavaScript equipado com um forte sistema de tipos. O sistema de tipos é muito expressivo graças à adição de tipos refinados e também de um operador que verifica se uma expressão pertence a um tipo. SafeScript apresenta *flow typing*, isto é, o tipo de uma expressão depende da sua localização no fluxo de controlo do programa. Tal como no HeadREST, não é possível realizar uma simples análise sintática para a validação de tipos. No entanto, neste caso trata-se de uma linguagem imperativa com *flow typing*, logo uma abordagem igual de tradução direta para um SMT *solver* não é trivial. Deste modo, a validação de tipos é feita traduzido o código SafeScript para a linguagem intermédia Boogie, onde as necessárias validações são traduzidas como asserções, sendo que o Boogie utiliza internamente o Z3 SMT *solver* para resolver semanticamente as asserções. Devido à validação semântica, o compilador de SafeScript consegue detetar estaticamente diversos erros de execução comuns, como divisão por zero ou acesso a um *array* fora dos seus limites, e que não conseguem ser detetados por linguagens similares, como o TypeScript. SafeScript compila para JavaScript, com o intuito de poder ser utilizado em conjunto com este.

Graças ao seu expressivo sistema de tipos, o validador de programas SafeScript é também um verificador estático. A partir deste é possível provar que um programa cumpre uma determinada especificação, que pode ser descrita usando os tipos refinados. Neste trabalho destacou-se a capacidade de prova do validador de SafeScript, concretamente

resolvendo alguns desafios propostos pelo *Verification Benchmarks Challange*.

A partir do SafeScript desenvolveu-se a extensão SafeRESTScript, que adiciona pedidos REST à sintaxe do SafeScript e valida-os estaticamente de encontro a uma especificação HeadREST. Para cada chamada REST são feitas principalmente duas validações. Em primeiro lugar, é verificado se o URL é um endereço válido do serviço para o método HTTP do pedido, isto é, se existe algum triplo na especificação com o par método e URL do pedido. De seguida, e com a tradução da especificação HeadREST importada para Boogie, é verificado se as chamadas REST cumprem os triplos da especificação, nomeadamente, se as pré-condições são cumpridas então as pós-condições também se devem verificar. Por exemplo, se uma pós-condição, cuja respetiva pré-condição é verdadeira para uma determinada chamada, asserta que no corpo da resposta existe um objeto com o campo id, então um acesso a este campo no corpo da resposta é validado. Neste trabalho, como exemplo ilustrativo das capacidades da linguagem, desenvolveu-se um cliente SafeRESTScript da API REST do conhecido repositório GitHub.

Ambas as linguagens possuem um compilador e editor que estão disponíveis como *plug-in* para o IDE Eclipse, para além de uma versão terminal. As duas linguagens possuem várias limitações, e por isso muito trabalho ainda existe pela frente. No entanto, SafeScript e SafeRESTScript não têm ambição de ser linguagens de produção, mas sim contribuir para um melhoramento da análise estática de programas e mostrar que é possível auxiliar o desenvolvimento fiável de código cliente de serviços REST.

Palavras-chave: REST, análise estática, JavaScript, tipos refinados

Abstract

REST is the architectural sytle most used in the web to exchange data. RESTful applications must be well documented so clients can use its services without doubts and errors. There are several specification languages for describing REST APIs, e.g. Open API Specification, but they lack on expressiveness to describe the exchanged data. HeadREST specification language was introduced to address this gap, containing an expressive type system that allows to describe rigorously the request and response formats of a service endpoint.

On the other hand, it is also important to ensure that REST calls in client code meet the service specification. This challenge is even more important taking in account that most REST clients are made in JavaScript, a weakly typed language.

To aim this problem, we firstly developed SafeScript, a subset of JavaScript equipped with a strong type system. SafeScript has a expressive type system thanks to refinement types and to an operator that checks if an expression belongs to a type. A semantic subtyping analysis is necessary; the typing validation in done by translating the code to Boogie intermediate language which uses the Z3 SMT solver for the semantic evaluation. SafeScript compiles directly to JavaScript.

SafeRESTScript is an extension of SafeScript that adds REST calls, being a client-side language for consuming REST services. It uses HeadREST specifications to verify REST calls: whether the URL of the call is a valid endpoint and whether the data exchanged match the pre and post-conditions declared in the specification.

With the creation of this new languages, we dot not intend in having them as production languages, but to show that it is possible to contribute with a better verification and correction in area where software reliability is weak.

Keywords: REST, static analysis, JavaScript, refinement types

Contents

List of Figures

List of Tables

Acronyms

ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
AST	Abstract Syntax Tree
ATP	Automated Theorem Proving
CoAP	Constrained Application Protocol
CRUD	Create, read, update and delete
DLS	Domain-Specific Language
ECMA	European Computer Manufacturers Association
FOL	First-order logic
IDE	Integrated Development Environment
IDL	Interface Definition Language
JSON	JavaScript Object Notation
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
LHS	Left-hand side
MSON	Markdown Syntax for Object Notation
OWL	Ontology Web Language
RAML	RESTful API Modeling Language
REST	Representational State Transfer
RFC	Request for Comments
SMT	Satisfiability modulo theories
SOA	Service-oriented architecture
SOAP	Simple Object Access Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WADL	Web Application Description Language
WSDL	Web Services Description Language
WWW	World Wide Web
XML	Extensible Markup Language

YAML Ain’t Markup Language

Chapter 1

Introduction

1.1 Motivation

Architectural styles are the basis for a good software system. Richard Taylor says that they "are a key technical ingredient in the creation and sustainment of a successful software ecosystem" [68], which evidence the importance of have a good, suitable and maintainable architecture in the development of a software product or service, specially the ones who intend to be scalable or integrable with others systems.

With the need of exchange information in web services, some web architectures and protocols emerged with the service-oriented architecture (SOA) paradigm. The two that are more distinguished, recognized and used by the community are REST architecture and SOAP protocol. In the last decade, particularly due to the grow of the World Wide Web [23], REST has become more and more prominent. Nowadays, 69% of the APIs are handed on REST services [47]. Also, RESTful services are seen more flexible and with more control, when directly compared with SOAP services [55]. Big software companies like Facebook, Google and Microsoft have REST APIs endpoints to their more relevant services [20, 30, 49], and some of them already retired their SOAP APIs [72].

With this in mind, we can anticipate the importance of REST in the current and future days. With the current permanent grow of the web applications, the usage of the REST architecture will scale as well. So, the necessity of reliability on RESTful applications is an important and actual topic.

During the development of a RESTful application, it is mandatory to have a good documentation about the API, so clients can use it without errors or doubts. Joop Aué et al. reported that 74% of API consumers consult documentation *often* or *very often* [2]. There are tools that help developers to build strong documentation. For example, Open API specification (swagger) [67] is one of the most popular. However, current tools have several limitations regarding the description of data exchanged in REST calls. They do not allow, for example, to describe some data as being a prime number, or establish a relation between two parameters (except in the natural language). Microsoft Outlook Calendar

API [49] has a practical and common example: a time range input must have an end time not before the respective start time.

On other hand, it is also important that the client side meets the web service specification. A study in a large-scale payment company revealed that request data (for being invalid or missing) causes the most faults for the respective API consumer [2].

In a general-purpose programming language, when a call is made to an external library, the compiler or interpreter checks whether the parameters have the right type. So, why can't we do have the same for REST calls? In the general case, it is not possible because there is not a standardization in the API specifications languages. There are some tools that perform some verification in REST calls inside popular languages, however, as expected, they required that the API specification must be done in their way. Wittern et al. suggested very recently four important challenges for web API consumption, one of them being this exact problem: web API clients don't know if their calls are right until runtime [80].

In the begin of this century, Tony Hoare proposed the creation of a *Verifying Compiler* as a grand challenge for the computing research, reinforcing that the "correctness of computer programs is the fundamental concern of the theory of programming and of its application in large-scale software engineering" [37]. Shortly, the major motivation of this work is to contribute with verification and correction in an area where there are several gaps in software reliability.

1.2 Context

This work was conducted at Large-Scale Informatics Systems Laboratory (LASIGE), a research unit at the the Department of Informatics, Faculty of Sciences, University of Lisboa, in the context of project Communication Contracts for Distributed Systems Development (CONFIDENT), supported by the *Fundação para a Ciência e Tecnologia* (FCT) through the Project UID/CEC/00408/2013.

CONFIDENT [75] is a toolchain for effective construction and evolution of REST APIs. In the context of this project was developed HeadREST, an expressive specification language of REST APIs [74]. From that language two tools were developed: HeadREST-TestTool [21], a tool to automatically test REST APIs from its HeadREST description; and HeadREST-CodeGen [62], a tool to generate server and client-side code from a HeadREST specification.

1.3 Objectives and Contributions

The first contribution is the reconstruction of the HeadREST language validator. Since its implementation several major bugs were found, being some of them related with the lan-

guage core typing rules. The original language validator source code had a high complexity, so it was decided to rebuild the validator from scratch, fixing errors and performing some improvements, culminating in a new version of the language.

The second goal and contribution is the design and development of SafeScript language, a JavaScript syntax based language equipped with types and a strongly type analyses. It was developed a declarative type system and a translation to Boogie, which is responsible for the semantic evaluation of the type system. SafeScript compiles to JavaScript, so it can be used together with this.

The third contribution and main objective of this thesis work is the development of SafeRESTScript, a client-side language for programming clients of REST applications. The language is an extension of SafeScript and uses HeadREST specifications to verify REST calls: whether the URL of the call is a valid endpoint and whether the data type exchanged and respective relations match the pre and post-conditions declared in the specification.

The artifacts produced in this work are available at CONFIDENT web page [75], and includes an Eclipse IDE plugin and a terminal version of SafeScript and SafeRESTScript languages compiler and editor. Also, a summarize version of this work is published in [9].

1.4 Structure of the document

This document is organised as follows:

Chapter 2 Background & Related Work - Introduces fundamental concepts to the understand of the problem. Reports similar languages, tools, and recent research related to the topic of this thesis.

Chapter 3 HeadREST Specification Language - Presents the HeadREST specification language, namely its syntax and validation rules. Includes some examples of specifications.

Chapter 4 SafeScript Language - Introduces the SafeScript language with its main ideas, syntax, and some simple examples. Describes the validation phase, namely the declarative type system and the translation to Boogie. Describes also the translation to JavaScript and presents the language operational semantics. Finalizes with an overview of SafeScript implementation.

Chapter 5 SafeRESTScript Language - Introduces the REST extension of SafeScript, its additional syntax and semantic, and a simple example of a SafeRESTScript client. Describes the novelties of the validation and translation phases, relatively to SafeScript. Ends with the integration of SafeRESTScript in SafeScript implementation, and with some editor IDE features.

Chapter 6 Evaluation - Evaluates the designed languages, comparing them with similar propose languages and tools. Presents a set of more complex examples of SafeScript and SafeRESTScript programmes. Enumerates the limitations of the languages, and from these suggests a plan for the future work.

Chapter 7 Conclusion - Summarizes the thesis, presenting the main contributions and conclusions of this work.

Chapter 2

Background & Related Work

2.1 REST - Representational State Transfer

Roy Fielding introduced the concept of REST in his PhD thesis, describing it as an architectural style for distributed hypermedia systems [24]. REST uses the most known hypermedia system, the World Wide Web (WWW), to represent and exchange data, but does not necessarily depend of it. REST was developed in parallel and together with the Hypertext Transfer Protocol (HTTP), the WWW protocol, with the goal of using HTTP as the main and most syntax appropriate protocol to make REST communications.

REST defines a few terms concerning its data elements. A *resource* is the abstraction of the data or information in REST. More formally, a resource R is a function, $M_R(t)$, that maps a time t to a set of entities, which can include *resource identifiers* and *resource representations*. The former is an identifier that references unequivocally a particular resource. A representation is basically a piece of information that captures a concrete state of a resource. This can be accompanied with the representation metadata that describes the representation itself. A resource can also have metadata with similar purposes and may have multiples representations in different formats.

The main principles and constraints in the design of REST architecture are the following [54]:

Stateless Interactions All REST communications must be stateless, therefore no state must be saved and used from a REST call to another. Clients do not establish permanent sessions when connecting to REST services.

Uniform Interface The interaction to all resources must be made through a uniform interface and each available method of the interface must have a well-defined semantics. This is the key feature that distinguishes the REST architectural from other network styles; the next constraints are a consequence of this decision.

Addressability All resources must have an unique and stable identifier (e.g., an URL in WWW).

Self-Describing Messages Services interact using request and response messages that contain the data exchanged and the metadata, i.e., the messages hold the information about their own data.

Hypermedia Resources can have relations to each other. For example, a resource representation can have an hyperlink to another resource, representing some relation between the two resources.

An application that fulfills these constraints is called RESTful. Applications that use the HTTP as the communication protocol take advantage of its syntax to make the exchanged messages more concise. The request message is composed by the URL of the resource, the HTTP method, a body (that may not be present) and a set of header fields, that must contain the media type of the body, if present. The response message contains a three digit response code with the correspondent text message, header fields and a body (also with the media type in the respective header).

HTTP includes several request methods, but for REST the more relevant ones are the equivalent to the four CRUD operations (create, read, update, delete), the base operations for persistence storage. They are formally described in RFC7231 [22]:

GET is the method for retrieving representations of one or more resources. It is a safe method since it is not supposed to change the resources set, i.e., after a GET call the data must not change. It is the CRUD equivalent to read operation.

POST is the target method for creating a new resource. The request should have a body with the representation associated with the resource to be created. It is the only method that is not idempotent, i.e., multiple equals sequential POST requests can create multiple resources. It is associated with the CRUD create operation.

PUT is the method for creating or changing an already existent resource(s). In creation utilization, the main different to the POST method (that also creates resources) is the fact that this operation is idempotent, so multiple equal creation requests are equivalent to a single request. In both cases the data is passed in the body. This method is used for the update operation.

DELETE is the available method for removing a resource. The request should have URI of the resource to be deleted in the header and no body. It corresponds to the CRUD delete operation.

Table 2.1 shows a summary of the various methods. Note that the definitions introduced may not be followed: REST is an architecture style and not a protocol or a standard. So, the above definitions can be seen as the REST best practices and not as a *must do*.

Figure 2.1 shows the Dummy API [19], a simple API that applies the REST principles to a CRUD application. Next, we present an example of a request to the Dummy API, with the creation of a new employee.

HTTP method	CRUD operation	Body in request	Safe	Idempotent
GET	read	Optional	Yes	Yes
POST	create	Yes	No	No
PUT	update	Yes	No	Yes
DELETE	delete	No	No	Yes

Table 2.1: Summary of the REST operations

#	Route	Method	Type	Full route	Description
1	/employee	GET	JSON	http://dummy.restapiexample.com/api/v1/employees	Get all employee data
2	/employee/{id}	GET	JSON	http://dummy.restapiexample.com/api/v1/employee/1	Get a single employee data
3	/create	POST	JSON	http://dummy.restapiexample.com/api/v1/create	Create new record in database
4	/update/{id}	PUT	JSON	http://dummy.restapiexample.com/api/v1/update/21	Update an employee record
5	/delete/{id}	DELETE	JSON	http://dummy.restapiexample.com/api/v1/update/2	Delete an employee record

Figure 2.1: Dummy API

```
POST /api/v1/create HTTP/1.1
Host: dummy.restapiexample.com
Content-Type: application/json

{"name": "test", "salary": "123", "age": "23"}
```

The first line contains the HTTP method, the relative path of the resource, and the version of the protocol. The following two lines introduce the header fields: the host (the first part of the resource URL), and the media type of the body. The last line contains the body (necessary in this case because we are in the presence of a POST operation) with the necessary operation data for the creation of an employee. One possible response to this request is the following:

```
HTTP/1.1 200 OK
Content-Type: application/json

{"name": "test", "salary": "123", "age": "23", "id": "5407"}
```

It begins with the response status code (200), accompanied with its description (`OK`), and with the protocol version. The second line contains the media type of the body, which is the only response header field. After a blank line, the response contains the body with a representation of the resource created.

From this simple example it becomes evident that the REST architecture matches very well the HTTP protocol. This contrasts with SOAP: if a SOAP application uses HTTP protocol for communication, it must only use POST method (not idempotent), which encapsulates any action that must be described in the body [65].

As already explained, REST naturally uses the HTTP protocol, but being an architecture style, it can be used with other protocols, specially for specific and more optimized

applications. For example, the Constrained Application Protocol (CoAP) [64], a protocol design specially for constrained networks (e.g., for low-power use), can be used to publish RESTfull APIs in embedded systems [61].

2.2 Interface Definition Languages for RESTful Applications

We now present the most relevant existing Interface Definition Languages (IDLs) for describing REST APIs.

RAML [59] is a language definition for RESTful APIs based on YAML version 1.2 [52], a human-friendly data serialization language. The goal of this language is to provide API documentation for users, but it may also be used to create client or server source code from specifications. It is equipped with different tools, including one for automatic testing a REST application from its RAML specification.

API Blueprint [8, 70] uses markdown [32], a markup language design to simplify HTML, to create a user-friendly documentation of RESTful APIs. It groups the actions by each resource, and uses MSON, a markdown specially design for data structures, to describe separately the data structures of the data exchanged. As in RAML, many tools were developed to help build servers and clients from specifications.

Web Application Description Language (WADL) [78] is a language to describe HTTP-based services, particularly oriented to REST applications. Based on XML, WADL is particularly designed to machines, so less adequate to humans. WADL allows the description of the set of resources, the relations among them, the HTTP methods that can be applied to the resources (with expected inputs and outputs and their supported formats) and also supports media types and data schemes. It is currently one of the languages recommended by the W3C [26].

Web Services Description Language (WSDL) [77] is another XML based language, originally developed for SOAP APIs. The latest version (2.0) added the semantics necessary to describe RESTful APIs. However, WSDL keeps a syntax specially suited for SOAP calls, so, together with the fact that there is another XML based language dedicated to REST APIs, WSDL is not recommend for REST.

SERIN [11] is an IDL to describe RESTful services using OWL semantics. The goal of the language is to define abstract models using interfaces, so that, when a host wants to implement a REST service, it must implement the interface described by the specification. SERIN uses annotated ontologies to provide information about the resources and their possible operations, using the OWL classes and properties for that characterization.

Open API specification [67] (formerly know as Swagger) is one of the most used languages for describing REST APIs. It may be written in JSON or YAML notations, providing good readability both for humans and machines. It can describe the available endpoints and their HTTP operations, the input and output for each operation (parameters schema, description, etc), methods of authentication and other important information for the user client (contact information, terms of use, etc).

RESTyped [14, 16] is a specification language for REST services made from TypeScript [50] annotations, more precisely from TypeScript description files. The notation is very similar to JSON. It can describe the available operations for each path, the type of the members of request (query, parameters, body, etc) and response. The types available are those of JavaScript, including intersection and union types, allowing a precise description of the API.

2.3 Static Verification in JavaScript Code

JavaScript is an interpreted language created by Netscape in 1996 with the goal of providing a more dynamic web [57]. It is the most popular web language for the development of client-side applications, being the most used language in the GitHub software repository [39].

There are languages and tools that help in statically detecting type-related errors related with types. Although these tools and languages do not directly involve REST calls, they provide some verification that can be useful in coding REST clients. Some of them are shortly described below.

TypeScript [50] is a superset of JavaScript created by Microsoft. Any valid JavaScript code is a valid TypeScript code, and this code is always compiled to JavaScript, thus enabling the use of other JavaScript frameworks. TypeScript introduces type annotations in field declarations, which help the compiler to understand some possible type errors. TypeScript also allows the creation of interfaces and classes, something that was not possible in JavaScript until the ECMAScript 6 definition [18]. In short, TypeScript helps in reducing the number of typing bugs in JavaScript code [28].

Dart [12] is an object-oriented class-based language that compiles directly to JavaScript. Unlike TypeScript, Dart is a completely new language, so it is not possible to write JavaScript code inside Dart (making code migration more difficult). It was design for web applications, mainly at the syntax level, which tends to facilitate the design of REST clients. It also has a good support for mobile applications.

Flow [38] is a static type checker for JavaScript developed by Facebook. It is very similar to TypeScript, in terms of annotations and the kind of errors detected. Although not

sold like that, flow is a new language, because the JavaScript interpreter doesn't support Flow annotations. The main difference when compared with TypeScript is that Flow depends on external tools to remove the needed annotations [63]. Flow was developed to be fast in the checking, thanks to its modularity and parallelization in the code analysis [10]. It was reported that Flow detects the same amount of bugs as TypeScript in JavaScript code [28].

JSHint [42] is a tool to detect errors and potential problems in JavaScript code. It statically analyses the code looking for syntax errors, leaking variables, invalid type conversions, and more possible bugs. It does not require specific annotations for verification, since it works with *pure* JavaScript code.

Parallel to these open-source projects, several researchers have proposed new solutions to deal with the dynamic typing of JavaScript programs.

Peter Thiemann identified a subset of JavaScript, added a type system, and called the result Core JavaScript [71]. This language predates those described above, but has similar goals: detect typing errors. However, Thiemann proved the soundness of the type system, a result that is not available for TypeScript.

Christopher Anderson et al. developed a similar system: a new language that is a subset of JavaScript (with few additions) and features a type system [1]. However, the goal of this language is to allow type inference, so type annotations may be omitted by the programmer and inferred by the compiler. The type system was also proved sound.

Nordio et al. created Javanni, a verifier for JavaScript [51]. This tool simple parses the JavaScript code to the Boogie language and then calls the Boogie verifier. Boogie [3] is an intermediate verification language developed by Microsoft. It is design to accommodate the encoding of verification conditions for imperative object-oriented programs. To discharge verification condition, Boogie uses Z3 [13], an efficient SMT solver, also developed by Microsoft Research. So, discarding possible Boogie heuristics, the typing bugs are *caught* by the SMT, and relayed to Javanni that alerts the error directly in the JavaScript code. The major advantage of using Javanni as opposed to previous tools is the fact that it deals directly with JavaScript code and does not require code annotation.

Vekris et al. developed Refined TypeScript, a lightweight refinement type system for TypeScript [76]. It can statically detect common runtime problems, e.g., in the areas of array safety, reflection, and down-casts. Regarding the validation, refinements are reduced to verification conditions, so the subtyping validation can be achieved with the help of an SMT solver.

2.4 Static Verification of REST Calls

The previous examples are a small set of the many works on static verification of JavaScript code. It is the principal research topic for this client-side language in the last years [66]. When compared to research concerning the verification of consumer code of REST APIs in JavaScript (or in similar client-side languages) the number goes down dramatically, and the solutions proposed tend to be very limited.

Wittern et al. propose an approach to statically check web API requests from JavaScript code [81]. They focus on ajax requests made using the jQuery library [41], although JavaScript supports different forms of REST APIs requests [33]. The tool aims at detecting bugs in two ways: checking whether the endpoints URLs calls match a valid URI template defined in a swagger specification, and checking whether the request data (both payload data and query parameters) are as expected. The tool uses a field-based call graph to make the necessary string analyses on the JavaScript method calls. Figure 2.2 shows an example of data flow in a context of a jQuery REST request. The URL is formed from variables defined outside the function scope, so a flow analyses is needed to build the URL. Note that no type verification is performed. The tool was tested with more than 6000 request from GitHub code, and had a general precision above 90%.

Figure 2.2: JQuery REST call example, from [81]

Dezfuli-Arjomandi developed RESTyped Axios, a client-side tool that verifies REST calls against a RESTyped specification [15, 16]. The code must be written in TypeScript and the requests must be made using the Axios framework [82]. It checks at compile time whether the URLs are valid and if the types of the members passed on the request and accessed on response correspond to the ones declared in the specification. An example of the specification of the Dummy API in RESTyped is the following.

```
1 interface Employee {
```

```
2    name: string
3    salary: string
4    age: string
5  }
6
7  interface EmployeeWithId {
8    name: string
9    salary: string
10   age: string
11   id: string
12 }
13
14 export interface DummyAPI {
15   '/employee/:id': {
16     GET: {
17       params: {
18         id: string
19       }
20       response: EmployeeWithId
21     }
22   }
23   '/create': {
24     POST: {
25       body: Employee
26       response: EmployeeWithId
27     }
28   }
29   //...
30 }
```

An example of a client of this API using the RESTyped Axios is presented next. The field `name` is declared as a string (line 8 above). If, for example, line 8 below is replaced by `name: 789`, then a typing error is thrown during the compilation process.

```
1  import axios from 'restyped-axios'
2  import {DummyAPI} from './dummyAPI'
3
4  const client = axios.create<DummyAPI>(
5      {baseURL: 'http://dummy.restapiexample.com/api/v1'})
6
7  client.post('/create', {
8      name: "test",
9      salary: "123",
10     age: "23"
11 })
```

In short, the lack of available research in this area and solutions to the problem, to-

gether with the fact that this is a topic of utmost importance in the software development process, suggest the difficulty of finding solutions to statically analyze REST clients.

Chapter 3

HeadREST Specification Language

In this chapter it is present HeadREST, a specification language for REST APIs. The language was already design, implemented [21], and publish [74]. Several bug were found in the language validator, some of them related with the core typing rules, so a new language validation algorithm is presented here, culminating in a new version of HeadREST language.

3.1 Basic Ideias

The main goal of the HeadREST specification language is to support the description of the operations in a REST API. It is based on two key ideas:

- Types to express properties of states and of data exchanged in interactions;
- 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.

Pre and post-conditions relations are expressed using Hoare Triples, a mathematical logical notation proposed by Tony Hoare [35]. Adapting this notation to our propose results in assertions composed by four parts:

$$\{e_1\}\ m\ u\ \{e_2\} \tag{3.1}$$

1. e_1, a boolean expression that establishes the pre-condition;
2. m, the HTTP method of the corresponding REST request (currently, the supported methods are `get`, `post`, `put` and `delete`);
3. u, an URI template that, according to the pre-condition, expands to a valid API endpoint;
4. e_2, a boolean expression that establishes the post-condition.

The typing system and its rules are based on the Dminor language [7]. This first-order functional language has two type primitives that accounts for its expressiveness:

- Refinement types, `x:T where e`, consisting of values `x` of type `T` that satisfy property `e`;
- A predicate, `e in T`, which returns **`true`** or **`false`** depending on whether the value of expression `e` is or is not of type `T`.

Refinement types were introduced by Freeman and Pfenning with the application to the language ML [27]. Hayashi describes a refinement type has a subset of an ordinary type, but not necessarily a subtype of this [34]. Figure 3.1 shows possible refinements of the integer type and their relation, using the set interpretation.

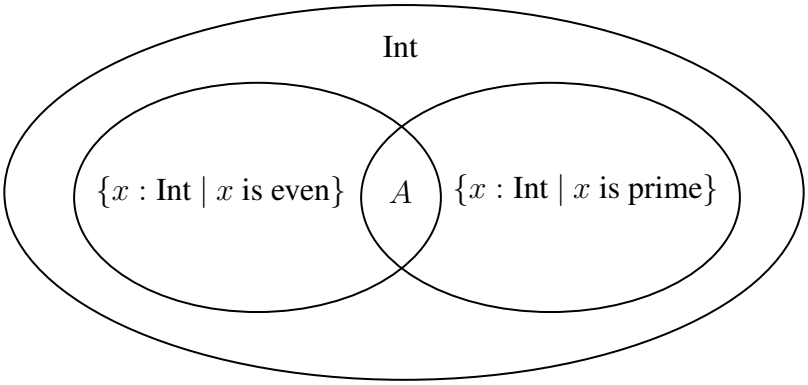

Figure 3.1: Refinement types as sets: an example with an integer type

Refining a type allow us to reduce the possible values of a type. In most imperative languages, method declarations are accompanied by descriptions, which sometimes include restrictions. Figure 3.2 introduces an example in the Java programming language, where is necessary to restrict the possible values of the arguments.

```
/**
 * Finds the minimal distance from a node to all other nodes
 * using Dijkstra algorithm
 * @param graph A graph represented as an adjacency list
 * @param i The source node
 * @requires 0 <= i < graph.length
 * @return An array with the minimal distance from i to all other nodes
 */
public static int[] dijkstra(Map<Integer, Integer>[] graph, int i)
```

Figure 3.2: Example of a method javadoc with *requires* clause

The arguments restrictions are addressed to humans only, since the standard Java compiler ignores them. Using the refinement types it is possible to restrict the type of the

parameters, `(i: int where 0 <= i && i < graph.length)`, providing more information for the compiler to detect type errors. Similar problems happen in basic expressions, for example when accessing an array object that must lie within array bounds or when performing an integer division, where the divisor must be non-zero.

In REST applications it is also common to restrict the types of the request data as well as the response data as a function of the request. Below is the same example, but as an HeadREST specification for an hypothetical Dijkstra REST API endpoint:

```
1  specification GraphLand
2
3  // a matrix
4  type Graph = (g: Integer[][] where
5      forall i: Integer . 0 <= i && i < length(g) ==>
6          length(g[i]) == length(g))
7
8  {
9      request in {body: Graph} &&
10     request.template.node in
11         (i: Integer where 0 <= i && i < length(request.body))
12 }
13     get '/dijkstra/{node}'
14 {
15     response.code == 200 &&
16     response in {body: (a: Integer[] where
17         length(a) == length(request.body) &&
18         a[request.template.node] == 0)}
19 }
```

First, we declare the representation of the graph as a type, in this case a simple matrix. Next, we show the success case, where the input corresponds exactly what is expected in the pre-condition: a body with a graph in the specified format, and the initial `node` parameter as an integer within the graph bounds. If these conditions are met, then the post-condition must apply: the response code must be 200 (OK) and the response body must contain an array with the length of the input graph, where the distance to the source node should by 0. These last conditions show the dependency of the response format with respect to the request. Other assertions can be written for the cases where the node index is out of bounds or the body is not in the right format. In such cases the response body and code should certainly have different types.

This example demonstrates that the dependency on types can be naturally described using refinement types, which demonstrates the power, expressiveness, and usefulness of including them in the HeadREST specification language.

Although this is a valid example of a REST endpoint, the endpoint described does not exactly match the spirit of the REST architecture. The URL is not a resource, so the operation does not manipulate a resource. In this case the operation is **`get`**, so it is

expected to return a resource representation in the body, but instead it returns the result of a static operation that does not depend or change the resource set.

3.2 A Running Example

For a more realistic example, we consider again the Dummy API [19]. Below, we meet a partial specification of one of its endpoints.

```
1  specification DummyAPI
2
3  resource Employee
4
5  type EmployeeRequest = {name: String, salary: String, age: String}
6  type EmployeeRepresentation = EmployeeRequest & {id: String}
7
8  // Creation of an employee, the successful case
9  {
10     request in {body: EmployeeRequest} &&
11     (forall e: Employee .
12         (forall eR: EmployeeRepresentation .
13             eR repof e => eR.name != request.body.name
14         )
15     )
16 }
17     post '/create'
18 {
19     response.code == 200 &&
20     response in {body: EmployeeRepresentation} &&
21     response.body.name == request.body.name &&
22     response.body.salary == request.body.salary &&
23     response.body.age == request.body.age &&
24     (exists e: Employee . response.body repof e)
25 }
```

It is possible (and sometimes desirable) to separate the specification of each endpoint in two: description of the data exchanged and description of the state, in terms of its resources. In this example, for the creation of a new employee the request data must have a body with a `EmployeeRequest` (line 10), that is by type definition, an object with fields `name`, `salary`, and `age`. If the pre-condition is satisfied, then the specification expects a response with the code 200 and with a body of the type `EmployeeRepresentation`, equal to `EmployeeRequest` but additionally with the field `id`, where all fields except this last one are equal to the ones send in request body (lines 19-23).

For describing the resource set it is declared the types of resources that exist in the system. The example declares the resource `Employee` in line 3, and also the type

`EmployeeRepresentation`, the only representation of this resource. The pre-condition includes a restriction on the state of the resource set: the name of the employee to be created must not be equal to any name of any representation of a Employee (lines 11-15). The post-condition also has guarantees that the body of the response is a representation of a resource (line 24).

This example shows how to issue a valid request to this endpoint. In general, each endpoint may have multiple successful cases, depending of the input sent in the request or the state of the resource set, and each can be described using additional Hoare triples for same endpoint URL and method. It is also possible and important to describe the unsuccessful cases of a call. Below is specified one of the unsuccessful cases of the employee creation endpoint, where the request includes an employee name equal to an existing name.

```
25 type ErrorMessage = {error: {text: String}}
26
27 // Creation of an employee, unsuccessful case: name already exists
28 {
29     request in {body: EmployeeRequest} &&
30     (exists e: Employee .
31         (exists eR: EmployeeRepresentation .
32             eR repof e && eR.name == request.body.name
33         )
34     )
35 }
36     post '/create'
37 {
38     response.code == 200 &&
39     response in {body: ErrorMessage}
40 }
```

The pre-conditions of triples over a same pair HTTP method - URI template may intersect each other, resulting in calls where possible multiple post-condition apply. In this example, and according to the DummyAPI [19], if the the request body is well formed the status code in the response is always 200, regardless of whether the employee was created or not. Note that in the post-condition of both triples the response code is ensure to be 200, but in this last, as the resource was not created, by the REST architecture style it should be a 4XX error, which report a client error. The triple described below asserts that, if the body of the call is as expected, then the response code should be 200. In fact, these two triples could omit the specification of the response code in the post-condition if we add the following triple.

```
40 // Creation of an employee, general response code
41 {
42     request in {body: EmployeeRequest}
43 }
```

```
44      post '/create'
45  {
46      response.code == 200
47  }
```

3.3 Syntax

The syntax and the validation system of HeadREST are heavily influenced by Dminor language [7]. Adaptations and additions to Dminor types, expressions and their respective validation rules were made to address the specific needs of REST.

3.3.1 Core Syntax

The HeadREST core syntax is presented in figure 3.3.

Expression	e	$::=$	$x \mid c \mid \oplus(e_1, \ldots, e_n) \mid e_1 \,?\, e_2 : e_3 \mid e \,\mathsf{in}\, T$
			$\mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l \mid [e_1, \ldots, e_n] \mid e_1[e_2]$
			$\mid \mathsf{forall}\ x : T \,.\, e \mid \mathsf{exists}\ x : T \,.\, e$
Scalar constant	c	$::=$	$n \mid s \mid \mathsf{true} \mid \mathsf{false} \mid u \mid r \mid \mathsf{null}$
Type	T	$::=$	$\mathsf{Any} \mid G \mid \{\} \mid \{l\!:T\} \mid T[] \mid (x\!:T\ \mathsf{where}\ e) \mid \alpha$
Basic type	G	$::=$	$\mathsf{Integer} \mid \mathsf{String} \mid \mathsf{Boolean} \mid \mathsf{Regexp} \mid \mathsf{URITemplate}$
Verb	m	$::=$	$\mathsf{get} \mid \mathsf{put} \mid \mathsf{post} \mid \mathsf{delete}$
Specification	S	$::=$	$\epsilon \mid \mathsf{var}\, x\!:T\ S \mid \mathsf{resource}\, \alpha\ S \mid \{e_1\}\, m\, u\, \{e_2\}\ S$

Figure 3.3: HeadREST syntax

Dminor has three scalar types **`Integer`**, **`String`** and **`Boolean`**. HeadREST added two new scalar types: **`URITemplate`**, necessary to represent a service endpoint, or a group of resources URIs, and **`Regexp`**, representing regular expressions, important to add extra expressiveness to string operations. Arrays (`T[]`) substitute Dminor bags, so that an order between elements might be established and enabling indexed access, aspects very relevant in the data exchanged in REST applications. The empty object type (`{}`) was added in order to enable empty objects and to provide a super type for all objects. The resource type (α) is particularly useful, since REST uses resources to represent data. **`Any`** is the top type.

To Dminor integer, string, and boolean scalar constants the **`null`** value was added. The **`null`** is a value of **`Any`** type and it is not a value of the object types. To inhabit the **`Regexp`** and **`URITemplate`** types two scalar constants were added: regular expressions and URI templates values. The regular expression of HeadREST form a subset of those in

JavasScript (defined in ECMAScript [69]). The syntax of URI Templates is conform to RCF-6570 [31], and is defined in figure 3.4.

URI Template	u	$::=$	$`t`$
Term	t	$::=$	$\epsilon \mid l\,t \mid \{v,\ldots,v\}\,t \mid \{?v,\ldots,v\}\,t$
Literal	l	$::=$	`([^"'%<>\^{|}] | %[0-9A-F]{2})+`
Variable	v	$::=$	`([0-9A-Z_a-z]|%[0-9A-F]{2})([.0-9A-Z_a-z]|%[0-9A-F]{2})*`

Figure 3.4: The syntax of URI templates

Expressions are those of Dminor, except the array literal and array access that replace the expressions for bags. Two further expressions were added: the **forall** and **exists** quantifiers. They are mandatory when expressing constraints related to resources and their representations. They also help in the declarative description of arrays and are also useful for the expressiveness of refinement of types. For example, the type that represents all prime numbers can only be represented using quantifiers:

```
1 type Prime = (x: Integer where x > 1 &&
2         forall i: Integer. 1 < i && i < x ==> x % i != 0)
```

HeadREST also inherits the primitive operators of Dminor, and adds a few more. Figure 3.5 introduces all HeadREST primitive operators and their type signatures.

$<=>$: Boolean, Boolean $\to$ Boolean	$=>$: Boolean, Boolean $\to$ Boolean
$\mid$: Boolean, Boolean $\to$ Boolean	$\&$: Boolean, Boolean $\to$ Boolean
$==$: Any, Any $\to$ Boolean	$!=$: Any, Any $\to$ Boolean
$<$: Integer, Integer $\to$ Boolean	$<=$: Integer, Integer $\to$ Boolean
$>$: Integer, Integer $\to$ Boolean	$>=$: Integer, Integer $\to$ Boolean
repof: Any, α $\to$ Boolean	uriof: String, α $\to$ Boolean
$+$: Integer, Integer $\to$ Integer	$-$: Integer, Integer $\to$ Integer
$++$: String, String $\to$ String	$*$: Integer, Integer $\to$ Integer
$\%$: Integer, Integer $\to$ Integer	$/$: Integer, Integer $\to$ Integer
!: Boolean $\to$ Boolean	$-$: Integer $\to$ Integer
length: Any [] $\to$ Integer	size: String $\to$ Integer
expand: URITemplate, {} $\to$ String	matches: Regexp, String $\to$ Boolean

Figure 3.5: Operators signatures: $\oplus$: $T_1,\ldots,T_n \to T$

Functions `length` and `size` determine, respectively, the length of an array and the size of a string, two internal and immutable features of each literal; `matches` checks whether

a string matches a regular expression, and `expand` performs the URI template expand operation as defined in RFC-6570 [31]. Two infix operators were added to manipulate resources: **`repof`** and **`uriof`**. The former checks whether a given value is a representation of some resource. The latter checks whether a given string is an URI that identifies a specific resource. As demonstrated in the examples, these two operators are important to describe the state of the resource set, namely to state pre-conditions involving the resource set, or to describe changes of the resource set after the call.

A HeadREST specification consist in three type of declarations: variable declarations (**`var`** `x: T`), resource declarations (**`resource`** `a`), and assertions (Hoare Triples) that describes the behavior of an endpoint. Currently HeadREST supports the four principal HTTP methods: **`get`**, **`post`**, **`put`** and **`delete`**.

3.3.2 Derived Syntax

Although HeadREST has a small core of types, these types together with HeadREST expressions, are quite expressive, due to the interplay between refinement types and the *it* predicate. Many other useful types are derivable, several of them that are primitive in other programming languages. Figure 3.6 shows some of them, where $\mathrm{fv}(\cdot)$ represents the free variables of an expression or type.

$$
\begin{array}{rcll}
[e\colon T] & \triangleq & (x\colon T \text{ where } x == e) & x \notin \mathrm{fv}(e) \\
[e] & \triangleq & [e\colon \mathsf{Any}] & \\
T \mid U & \triangleq & (x\colon \mathsf{Any} \text{ where } (x \text{ in } T \mid x \text{ in } U)) & x \notin \mathrm{fv}(T, U) \\
T \,\&\, U & \triangleq & (x\colon \mathsf{Any} \text{ where } (x \text{ in } T \,\&\, x \text{ in } U)) & x \notin \mathrm{fv}(T, U) \\
!T & \triangleq & (x\colon \mathsf{Any} \text{ where } !(x \text{ in } T)) & x \notin \mathrm{fv}(T) \\
\|e\| & \triangleq & (x\colon \mathsf{Any} \text{ where } e) & x \notin \mathrm{fv}(e) \\
\text{if } e \text{ then } T \text{ else } U & \triangleq & (T \text{ where } e) \mid (U \text{ where } !e) & \\
\{?l : T\} & \triangleq & (x\colon \{\} \text{ where } x \text{ in } \{l\colon \mathsf{Any}\} \Rightarrow x \text{ in } \{l\colon T\}) & x \notin \mathrm{fv}(T) \\
\{\star l_1\colon T_1, \ldots, \star l_n\colon T_n\} & \triangleq & \{\star l_1\colon T_1\} \& \ldots \& \{\star l_n\colon T_n\} \quad \text{where } \star \text{ is } ? \text{ or } \epsilon & n \geq 2 \\
\mathsf{Natural} & \triangleq & (x\colon \mathsf{Integer} \text{ where } x \geq 0) & \\
\mathsf{Empty} & \triangleq & (x\colon \mathsf{Any} \text{ where } \mathsf{false}) &
\end{array}
$$

Figure 3.6: Type abbreviations

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. For example, `{l:` **`Integer`**`, m:` **`Boolean`**`}` abbrevi-

ates the type `(x: Any where (x in {l: Integer} & x in {m: Boolean})`, which only uses core types. Another important derived type is the optional field type, `{?l: T}`. According to the definition in RCF, URI templates can have optional variables, as seen in the last term of URI Template syntax (figure 3.4). So the type of these variables can only be represented with this derived type, which asserts that if an object has a field `l` then its type is `T`.

The derived singleton type, `[e]`, is particular to refinement types. In HeadREST each type can be seen as a set of values. A singleton type allows to restrict the type of an expression to a singleton set. Normally in a standard programming language, an expressions such as `7 * (2 + 4)` has simply the type `Integer`. In HeadREST, using the singleton type, the same expression has the type `[7 * (2 + 4)]`, which is equivalent to `(x: Any where x == 42)`.

The derived expressions of HeadREST are presented in figure 3.7. The function `isdefined` checks whether an object has a given field. Other three are boolean shortcut operators.

$$
\begin{aligned}
\mathsf{isdefined}(e.l_1.l_2 \ldots l_n) &\triangleq e \text{ in } \{l_1 \colon \{l_2 \colon \{\ldots \{l_n \colon \mathsf{Any}\} \ldots\}\}\} \\
e \;\&\&\; f &\triangleq e \,?\, f : \mathsf{false} \\
e \;||\; f &\triangleq e \,?\, \mathsf{true} : f \\
e \implies f &\triangleq e \,?\, f : \mathsf{true}
\end{aligned}
$$

Figure 3.7: Derived expressions

In what concerns declarations, HeadREST supports type aliases: type $X = T$. Occurrences of the type identifier X are directly replaced by T in the rest of the specification, without the creation of a new type. This and other declarations can appear in any order, but each only affects the declarations that apper after, so the order is relevant.

3.4 Validation

3.4.1 Algorithmic Type Checking

Typing checking is the more difficult and complex part of validating an HeadREST specification. Figure 3.8 introduces additional syntax necessary for the validation algorithm: a variable context Γ that maps variables in scope to their declared types, and a resource context Δ, a list of resources.

Type validation is made via a bidirectional system of rules, composed by two main relations: one that synthesizes the type of each expression and one that checks whether an expression is of a given type. Additionally, there are rules to check the good formation

Variable context	$\Gamma \ ::= \ \epsilon \mid \Gamma, x\colon T$
Resource context	$\Delta \ ::= \ \epsilon \mid \Delta, \alpha$

Figure 3.8: Validation contexts

of the contexts, types, and specifications, as well as a rule for the subtyping relation. Figure 3.9 summarizes the judgments of the algorithmic type system.

$\Delta \vdash \Gamma$	in Δ, context Γ is well formed	Figure 3.10
$\Delta; \Gamma \vdash T$	in $\Delta; \Gamma$, type T is well formed	Figure 3.11
$\Delta; \Gamma \vdash e \to T$	in $\Delta; \Gamma$, expression e synthesizes type T	Figure 3.12
$\Delta; \Gamma \vdash e \leftarrow T$	in $\Delta; \Gamma$, expression e checks against type T	Figure 3.13
$\Delta; \Gamma \vdash S$	in $\Delta; \Gamma$, specification S is well formed	Figure 3.14
$\vdash u \twoheadrightarrow T$	URI template u synthesizes type T	Figure 3.16
$\Delta; \Gamma \vdash T <: U$	in $\Delta; \Gamma$, type T is a subtype of type U	Figure 3.17

Figure 3.9: Judgments of the algorithmic type system

The first relation, presented in figure 3.10, verifies whether a variable context is well formed given the resource context. Note that the variable context can reference resources, namely in the types of the variables, but not the contrary.

$$\frac{}{\Delta \vdash \epsilon} \qquad \frac{\Delta; \Gamma \vdash T \quad x \notin \Gamma}{\Delta \vdash \Gamma, x\colon T}$$

Figure 3.10: Algorithmic context formation: $\Delta \vdash \Gamma$

The validation checks whether types of each variable are well formed. To check whether a type is well formed, it is used the second relation, whose rules are presented in figure 3.11.

$$\frac{}{\Delta; \Gamma \vdash \mathsf{Any}} \qquad \frac{}{\Delta; \Gamma \vdash G} \qquad \frac{}{\Delta; \Gamma \vdash \{\}} \qquad \frac{\alpha \in \Delta}{\Delta; \Gamma \vdash \alpha}$$

$$\frac{\Delta; \Gamma \vdash T}{\Delta; \Gamma \vdash T\,[]} \qquad \frac{\Delta; \Gamma \vdash T}{\Delta; \Gamma \vdash \{l\colon T\}} \qquad \frac{\Delta; \Gamma \vdash T \quad \Delta; \Gamma, x\colon T \vdash e \leftarrow \mathsf{Boolean}}{\Delta; \Gamma \vdash (x\colon T \ \mathsf{where}\ e)}$$

Figure 3.11: Algorithmic type formation: $\Delta; \Gamma \vdash T$

The only type that actually needs a type validation is the refinement type, where the expression must be a boolean condition given the variable context loaded with the bind of the refinement. The type presented in the bind must be well formed in order to be added to the variable context. The other rules check whether the types within the main type (if any) are well formed, except the resource type rule that checks whether the resource is included in the resource context.

Type synthesis of an expression is defined by the rules in figure 3.12. In general, the rules use the *check against* rule to check whether the expression arguments have the expected type. The rules synthesize the singleton type of each expression (or the most precise type). This type is accompanied with a less precise type for better error reporting. For example, in the Synth Const rule, and for the `Integer` expression `1`, it is not synthesize the type `(x: Any where x == 1)`, but the equivalent type `(x: Integer where x == 1)`. It is introduced a single rule for all scalar constants, with function $\mathsf{typeof}(c)$ yielding the primitive type of the scalar constant c.

The ternary operator has the particularity of passing the information of the condition to each of its branches. In other words, in the evaluation of the true branch the fact that the condition e_1 is true is added to the context, and in the evaluation of the false branch the fact that the condition e_1 is false is added to the context. This is done in the rule Synth Cond by adding a dummy variable to the context, whose type is $\|e_1\|$ or $\|!e_1\|$. Recall from figure 3.6 that $\|e\|$ stands for the type $(x\colon \mathsf{Any\ where}\ e)$. For example, if `obj` is a variable with the type `{}` then the expression `obj in {l: boolean} ? obj.l : false` is valid, since the formation of `obj.l` is guaranteed by the condition that `obj` has the field `l`. An expression with the same meaning can be written as the conjunction short-cut operator: `obj in {l: boolean} && obj.l`.

The object and array access rules require type extraction. For the object field access, first the type of the object is synthesized, then this type is normalized to a Disjunctive Normal Form (DNF) and from the DNF type the type of the field is extracted. The DNF type syntax, D, the normalization rules, $\mathrm{norm}(T)$, and the extraction rules, $D.l \rightsquigarrow U$ and $D.\,\mathrm{items} \rightsquigarrow U$, are from Dminor and are presented in Appendix A. Adaptations where needed in the extraction rules to support arrays instead of Dminor bags.

These rules have a problem: they are partial, so they do not work on all cases. The $\mathrm{norm}_r(T)$ function of the type normalization, that is responsible for normalize refinement types, only extract types from some expressions, missing the ternary operator, for example. To work around this situation, it is possible to, in the case an unsuccessful extraction, make an additional check if the object expression has that access: $\Delta; \Gamma \vdash e \leftarrow \{l\colon \mathsf{Any}\}$. If the check is positive, then it is synthesized a type with only the singleton of the expression, which is a type equivalent to the one synthesize if the extraction were successful.

Array access rules checks whether the index has an in-bounds integer type ensuring that the access is valid.

$$\frac{(x\colon T) \in \Gamma}{\Delta;\Gamma \vdash x \to [x\colon T]} \qquad \frac{}{\Delta;\Gamma \vdash c \to [c\colon \mathsf{typeof}(c)]} \quad \text{(Synth Var, Synth Const)}$$

$$\frac{\oplus : T_1, \ldots, T_n \to T \quad \Delta;\Gamma \vdash e_i \leftarrow T_i \quad \forall i \in 1..n}{\Delta;\Gamma \vdash \oplus(e_1, \ldots, e_n) \to [\oplus(e_1, \ldots, e_n)\colon T]} \quad \text{(Synth Operator)}$$

$$\frac{\Delta;\Gamma \vdash e_1 \leftarrow \mathsf{Boolean} \quad \Delta;\Gamma, \_\colon \|e_1\| \vdash e_2 \to T_2 \quad \Delta;\Gamma, \_\colon \|!e_1\| \vdash e_3 \to T_3}{\Delta;\Gamma \vdash (e_1 \mathbin{?} e_2 : e_3) \to (\mathsf{if}\ e_1\ \mathsf{then}\ T_2\ \mathsf{else}\ T_3)} \quad \text{(Synth Cond)}$$

$$\frac{\Delta;\Gamma \vdash T \quad \Delta;\Gamma \vdash e \leftarrow \mathsf{Any}}{\Delta;\Gamma \vdash e\ \mathsf{in}\ T \to [e\ \mathsf{in}\ T\colon \mathsf{Boolean}]} \quad \text{(Synth Test)}$$

$$\frac{\Delta;\Gamma \vdash e_i \to T_i \quad \forall i \in 1..n}{\Delta;\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} \to \{l_1\colon T_1, \ldots, l_n\colon T_n\}} \quad \text{(Synth Obj)}$$

$$\frac{\Delta;\Gamma \vdash e \to T \quad \mathrm{norm}(T) = D \quad D.l \rightsquigarrow U}{\Delta;\Gamma \vdash e.l \to [e.l\colon U]} \quad \text{(Synth Dot)}$$

$$\frac{\Delta;\Gamma \vdash e_i \to T_i \quad T = T_1 \mid \ldots \mid T_n\ (n > 0, \mathsf{Any}\ \mathsf{when}\ n = 0) \quad \forall i \in 1..n}{\Delta;\Gamma \vdash [e_1, \ldots, e_n] \to [[e_1, \ldots, e_n] : T\,[]]} \quad \text{(Synth Array)}$$

$$\frac{\Delta;\Gamma \vdash e_1 \to T \quad \mathrm{norm}(T) = D \quad D.\mathrm{items} \rightsquigarrow U \quad \Delta;\Gamma \vdash e_2 \leftarrow (i\colon \mathsf{Natural}\ \mathsf{where}\ i < \mathsf{length}(e_1))}{\Delta;\Gamma \vdash e_1[e_2] \to [e_1[e_2]\colon U]} \quad \text{(Synth Array Elem)}$$

$$\frac{\Delta;\Gamma \vdash T \quad \Delta;\Gamma, x\colon T \vdash e \leftarrow \mathsf{Boolean}}{\Delta;\Gamma \vdash \mathsf{forall/exists}\ x : T\,.\,e \to [(\mathsf{forall/exists}\ x : T\,.\,e)\colon \mathsf{Boolean}]} \quad \text{(Synth Quantifier)}$$

Figure 3.12: Algorithmic type synthesis: $\Delta;\Gamma \vdash e \to T$

The check against system are presented is figure 3.13. The principal rule is the Check Swap. It can be applied to any type: the type of the expression is synthesized and then it is compared against the expected type using subtyping. The algorithm works only with this single rule, the other three were added because they provide in practice a better precision in the type-checking, according to Bierman et al [7], and yield better error messages. They also help in not reach the synthesis rules of object and array access that have a partial extraction.

This bidirectional system of synthesis and check against is responsible for type-checking expressions, including those inside refinement types. In HeadREST it is used for checking the pre and post-condition of each assertion. Variable and resource declarations have the responsibility of setting the variable and resource context. Figure 3.14 presents the rules for checking the good formation of a HeadREST specification.

$$\frac{\Delta;\Gamma \vdash e_1 \leftarrow \mathsf{Boolean} \quad \Delta;\Gamma, \_\colon \|e_1\| \vdash e_2 \leftarrow T \quad \Delta;\Gamma, \_\colon \|!e_1\| \vdash e_3 \leftarrow T}{\Delta;\Gamma \vdash (e_1\ ?\ e_2 : e_3) \leftarrow T} \text{(Check Cond)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_2 \leftarrow (x\colon \mathsf{Natural\ where}\ x < \mathsf{length}(e_1)) \\ \Delta;\Gamma \vdash e_1 \leftarrow (a\colon \mathsf{Any}\,[]\ \mathsf{where}\ a[e_2]\ \mathsf{in}\ T)\end{array}}{\Delta;\Gamma \vdash e_1[e_2] \leftarrow T} \text{(Check Array Elem)}$$

$$\frac{\Delta;\Gamma \vdash e \leftarrow \{l\colon T\}}{\Delta;\Gamma \vdash e.l \leftarrow T} \qquad \frac{\Delta;\Gamma \vdash e \rightarrow T \quad \Delta;\Gamma \vdash T <: U}{\Delta;\Gamma \vdash e \leftarrow U} \text{(Check Dot, Check Swap)}$$

Figure 3.13: Algorithmic type checking: $\Delta;\Gamma \vdash e \leftarrow T$

$$\frac{}{\Delta;\Gamma \vdash \epsilon} \qquad \frac{\Delta;\Gamma \vdash T \quad \Delta;\Gamma, x\colon T \vdash S}{\Delta;\Gamma \vdash \mathsf{var}\, x\colon T; S} \qquad \frac{\Delta, \alpha;\Gamma \vdash S}{\Delta;\Gamma \vdash \mathsf{resource}\, \alpha; S}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash S \quad \vdash u \twoheadrightarrow T \quad \Delta;\Gamma, \mathsf{request}\colon \mathsf{Request} \,\&\, T, \mathsf{root}\colon \mathsf{String} \vdash e_1 \leftarrow \mathsf{Boolean} \\ \Delta;\Gamma, \mathsf{request}\colon \mathsf{Request} \,\&\, T, \mathsf{response}\colon \mathsf{Response}, \mathsf{root}\colon \mathsf{String}, \_ : \|e_1\| \vdash e_2 \leftarrow \mathsf{Boolean}\end{array}}{\Delta;\Gamma \vdash \{e_1\}\, m\, u\, \{e_2\}; S}$$

Figure 3.14: Algorithmic specification formation: $\Delta;\Gamma \vdash S$

For the evaluation of the pre and post-conditions three variables are added: request and response variables that correspond to the call and respective response, and root which is the absolute URL of the service. The request and response types are described in figure 3.15, and are according to the HTTP format presented in section 2.1.

$$\begin{aligned}\mathsf{Request} &\triangleq \{\mathsf{location}\colon \mathsf{String}, ?\mathsf{template}\colon \{\}, \mathsf{header}\colon \{\}, ?\mathsf{body}\colon \mathsf{Any}\} \\ \mathsf{Response} &\triangleq \{\mathsf{code}\colon \mathsf{Integer}, \mathsf{header}\colon \{\}, ?\mathsf{body}\colon \mathsf{Any}\}\end{aligned}$$

Figure 3.15: Request and response types

To the `Request` type we add the type extracted from the URI template of the assertion. The URI template contains a set of variables (possibly empty) that must be substituted by their values to execute the call to a valid endpoint URL. The values of these variables can be accessed via the field template of the request variable. Figure 3.16 introduces the rules for type extraction from an URI template. Note that no type is extracted from the optional variables, since they are not guaranteed to be present.

For the evaluation of the post-condition we consider that the pre-condition holds, so a dummy variable is added for that purpose. This simplifies the post-condition, since

$$\frac{\vdash t \twoheadrightarrow T}{\vdash \text{`}t\text{`} \twoheadrightarrow T} \qquad \frac{}{\vdash \epsilon \twoheadrightarrow \{\}} \qquad \frac{\vdash t \twoheadrightarrow T}{\vdash l\ t \twoheadrightarrow T} \qquad \frac{\vdash t \twoheadrightarrow T}{\vdash \{?v_1, \ldots, v_n\}\ t \twoheadrightarrow T}$$

$$\frac{\vdash t \twoheadrightarrow T}{\vdash \{v_1, \ldots, v_n\}\ t \twoheadrightarrow T\ \&\ \{\textit{template}\colon \{v_1\colon \mathsf{Any}, \ldots, v_n\colon \mathsf{Any}\}\}}$$

Figure 3.16: URI template type extraction: $\vdash u \twoheadrightarrow T$

it does not need to restrict again the type of the request variable. However, resource operations (**repof** and **uriof**) may change after the call and therefore a expression with these operators may have two different values in pre and post-conditions of the same triple. So, when the pre-condition is placed in the content of the post, both operations are encapsulated with an internal operator pre (an uninterpreted operation), permitting distinguish values from before and after the call is done.

Finally, for this system to work it is necessary a subtyping algorithm; its rules are presented in figure 3.17. There are two types of rules: a semantic rule, the last in the figure, and the syntactic rules, all the others.

$$\frac{}{\Delta; \Gamma \vdash T <: \mathsf{Any}} \qquad \frac{}{\Delta; \Gamma \vdash G <: G} \qquad \text{(Subt Any, Subt Scalar)}$$

$$\frac{}{\Delta; \Gamma \vdash \{\} <: \{\}} \qquad \frac{\Delta; \Gamma \vdash T <: U}{\Delta; \Gamma \vdash \{l\colon T\} <: \{l\colon U\}} \qquad \text{(Subt Empty Obj, Subt Obj)}$$

$$\frac{}{\Delta; \Gamma \vdash \{l\colon T\} <: \{\}} \qquad \frac{\Delta; \Gamma \vdash T <: U}{\Delta; \Gamma \vdash T\,[] <: U\,[]} \qquad \text{(Subt Nonempty Obj, Subt Array)}$$

$$\frac{\Delta; \Gamma \vdash T <: U}{\Delta; \Gamma \vdash (x\colon T\ \mathsf{where}\ e) <: U} \qquad \frac{}{\Delta; \Gamma \vdash \alpha <: \alpha} \qquad \text{(Subt Refine, Subt Resource)}$$

$$\frac{x \notin \mathsf{dom}(\Gamma) \quad \vDash (\mathbf{F}'[\![\Gamma]\!] \wedge \mathbf{F}'[\![T]\!](x)) \Rightarrow \mathbf{F}'[\![U]\!](x)}{\Delta; \Gamma \vdash T <: U} \qquad \text{(Subt Semantic)}$$

Figure 3.17: Algorithmic subtyping: $\Delta; \Gamma \vdash T <: U$

The presence of refinement types and the *in* predicate does not allow a syntactic approach to subtyping. So a semantic rule is included, where the subtyping relation, together with variable and resource context, is transformed into a first order logic formula to be checked by an SMT solver. Details of this transformation are presented in next subsection.

The semantic rule handle all subtyping goals, and by itself is sufficient for the algorithm to work. However, we added a set of syntactic rules. These are applied whenever possible before the semantic rule, trying to solve the subtyping rule without using the SMT solver, which is a time costly operation.

For each set of rules, it is considered in what context the rules are called, to remove redundant checks in the precondition of some rules. This happens particularly with the context formation rules. Figure 3.18 presents a top formation rule for specification that checks the formation of the contexts before checking the specification. This allows to remove all context validations from the algorithm rules, since all variables and resources added to the context are guaranteed valid by other checks, as happens for example in the rule Synth Quantifier (figure 3.12) or the refinement type formation (figure 3.11).

$$\frac{\Delta \vdash \Gamma \quad \Delta; \Gamma \vdash S}{\Delta; \Gamma \vdash_t S}$$

Figure 3.18: Algorithmic specification formation (top level): $\Delta; \Gamma \vdash_t S$

3.4.2 Semantic Subtyping

In HeadREST type system, a type T is subtype of U if and only if all values that belong to T also belong to U. The subtyping rule uses function $\mathbf{F}'[\![T]\!](e)$, which transforms the expression `e in T` into a first order logic formula. So, if for all x, $\mathbf{F}'[\![T]\!](x) \Rightarrow \mathbf{F}'[\![U]\!](x)$, then T is a subtype of U.

FOL formulas are evaluated using an SMT solver. Our implementation uses Z3, an efficient SMT solver developed by Microsoft [13]. The procedure of the translation is based on Bierman et al. work [7]. Figure 3.19 shows the translation for the *in type* relation, where t is a FOL term.

$$\begin{aligned}
\mathbf{F}'[\![\mathsf{Any}]\!](t) &= \mathsf{true} \\
\mathbf{F}'[\![\mathsf{Integer}]\!](t) &= \mathsf{In_Integer}(t) \\
\mathbf{F}'[\![\mathsf{Boolean}]\!](t) &= \mathsf{In_Boolean}(t) \\
\mathbf{F}'[\![\mathsf{String}]\!](t) &= \mathsf{In_String}(t) \\
\mathbf{F}'[\![\mathsf{URI}]\!](t) &= \mathsf{In_URITemplate}(t) \\
\mathbf{F}'[\![\mathsf{Regexp}]\!](t) &= \mathsf{In_Regexp}(t) \\
\mathbf{F}'[\![\alpha]\!](t) &= \mathsf{Good_R}(t) \wedge \mathsf{is_resource_of}(t, \alpha) \\
\mathbf{F}'[\![T\,[]]\!](t) &= \mathsf{Good_A}(t) \wedge \\
&\qquad (\forall i : \mathsf{int}.\, \mathsf{array_has_value}(t, i) \Rightarrow \mathbf{F}'[\![T]\!](\mathsf{v_nth}(t, i)) \\
\mathbf{F}'[\![\{\}]\!](t) &= \mathsf{Good_O}(t) \\
\mathbf{F}'[\![\{l : T\}]\!](t) &= \mathsf{Good_O}(t) \wedge \mathsf{v_has_field}(t, l) \wedge \mathbf{F}'[\![T]\!](\mathsf{v_dot}(t, l)) \\
\mathbf{F}'[\![(x\colon T \text{ where } e)]\!](t) &= \mathbf{F}'[\![T]\!](t) \wedge \mathbf{V}[\![[t/x]e]\!] = \mathbf{V}[\![\mathsf{true}]\!]
\end{aligned}$$

Figure 3.19: Type to FOL: $\mathbf{F}'[\![T]\!](t)$

The translation uses auxiliary predicates, uninterpreted functions and axioms that can be found in appendix B. They are described using the standard SMT-LIB 2 syntax [4]. The axiomatization is provided to the SMT when a subtyping goal is evaluated. The variable context is also added to the evaluation of the relation, as showed in the rule Subt Semantic (figure 3.17). Figure 3.20 presents the transformation of variable contexts into FOL formulas.

$$\begin{aligned}\mathbf{F}'[\![\epsilon]\!] &= \mathsf{true}\\ \mathbf{F}'[\![\Gamma, x : T]\!] &= \mathbf{F}'[\![T]\!](x) \wedge \mathbf{F}'[\![\Gamma]\!]\end{aligned}$$

Figure 3.20: Variable context to FOL: $\mathbf{F}'[\![\Gamma]\!]$

The refinement type contains an arbitrary expression, so all expressions must be transformed to formulas. Figure 3.21 presents $\mathbf{V}[\![e]\!]$, the transformation of HeadREST expressions into Value, a datatype common to all values: scalar constants, arrays, objects, regular expressions, URI template ans resources. This datatype is defined in the axiomatization (appendix B).

$$\begin{aligned}\mathbf{V}[\![\mathsf{forall}\ x : T\,.\,e]\!] &= \mathsf{G\_Boolean}(\forall x.(\mathbf{F}'[\![T]\!](x) \Rightarrow \mathbf{V}[\![e]\!] = \mathbf{V}[\![\mathsf{true}]\!]))\\ \mathbf{V}[\![\mathsf{exists}\ x : T\,.\,e]\!] &= \mathsf{G\_Boolean}(\exists x.(\mathbf{F}'[\![T]\!](x) \wedge \mathbf{V}[\![e]\!] = \mathbf{V}[\![\mathsf{true}]\!]))\\ \mathbf{V}[\![e_1\ ?\ e_2 : e_3]\!] &= \textbf{if}\ \mathbf{V}[\![e_1]\!] = \mathbf{V}[\![\mathsf{true}]\!]\ \textbf{then}\ \mathbf{V}[\![e_2]\!]\ \textbf{else}\ \mathbf{V}[\![e_3]\!]\\ \mathbf{V}[\![\oplus(e_1, \ldots, e_n)]\!] &= \mathbf{V}[\![\oplus]\!](\mathbf{V}[\![e_1]\!], \ldots, \mathbf{V}[\![e_n]\!])\\ \mathbf{V}[\![e\ \mathsf{in}\ T]\!] &= \mathsf{G\_Boolean}(\mathbf{F}'[\![T]\!](\mathbf{V}[\![e]\!]))\\ \mathbf{V}[\![e_1[e_2]]\!] &= \mathsf{v\_nth}(\mathbf{V}[\![e_1]\!], \mathbf{V}[\![e_2]\!])\\ \mathbf{V}[\![e.l]\!] &= \mathsf{v\_dot}(\mathbf{V}[\![e]\!], l)\\ \mathbf{V}[\![\{l_1 : e_1, \ldots, l_n : e_n\}]\!] &= \mathsf{O}(\{l_1 \mapsto \mathbf{V}[\![e_1]\!], \ldots, l_n \mapsto \mathbf{V}[\![e_n]\!]\})\\ \mathbf{V}[\![[e_1, \ldots, e_n]]\!] &= \mathsf{A}(\{0 \mapsto \mathbf{V}[\![e_1]\!], \ldots, n-1 \mapsto \mathbf{V}[\![e_n]\!]\}, n)\\ \mathbf{V}[\![\mathsf{true}]\!] &= \mathsf{v\_tt}\\ \mathbf{V}[\![\mathsf{false}]\!] &= \mathsf{v\_ff}\\ \mathbf{V}[\![n]\!] &= \mathsf{G\_Integer}(n)\\ \mathbf{V}[\![s]\!] &= \mathsf{G\_String}(s)\\ \mathbf{V}[\![r]\!] &= \mathsf{G\_Regexp}(r)\\ \mathbf{V}[\![u]\!] &= \mathsf{G\_UriTemplate}(u)\\ \mathbf{V}[\![\mathsf{null}]\!] &= \mathsf{v\_null}\\ \mathbf{V}[\![x]\!] &= x\end{aligned}$$

Figure 3.21: Expressions to FOL: $\mathbf{V}[\![e]\!]$

The boolean and integer scalar constants are translated to the respective SMT primitive value. String and regular expressions values use operations in the Z3str3 solver [5]. Arrays and objects are translated to maps, the former from integers to values, and the latter from string to values. URI template values are decomposed in their atomic variables and literals, so that the expand operation may be performed primitively. The operations are translated using functions from the axiomatization (figure 3.22). Exceptions are the operators that involve resources, **repof** and **uriof**, and the internal pre operator, which are uninterpreted functions.

$$
\begin{array}{rrr}
\mathbf{V}[\![<=>]\!] = \mathsf{O\_Equiv} & \mathbf{V}[\![=>]\!] = \mathsf{O\_Implies} & \mathbf{V}[\![|]\!] = \mathsf{O\_Or} \\
\mathbf{V}[\![\&]\!] = \mathsf{O\_And} & \mathbf{V}[\![==]\!] = \mathsf{O\_EQ} & \mathbf{V}[\![!=]\!] = \mathsf{O\_NE} \\
\mathbf{V}[\![<]\!] = \mathsf{O\_NT} & \mathbf{V}[\![<=]\!] = \mathsf{O\_LE} & \mathbf{V}[\![>]\!] = \mathsf{O\_GT} \\
\mathbf{V}[\![>=]\!] = \mathsf{O\_GE} & \mathbf{V}[\![\mathsf{repof}]\!] = \mathsf{r\_repof} & \mathbf{V}[\![\mathsf{uriof}]\!] = \mathsf{r\_uriof} \\
\mathbf{V}[\![+]\!] = \mathsf{O\_Sum} & \mathbf{V}[\![-]\!] = \mathsf{O\_Sub} & \mathbf{V}[\![++]\!] = \mathsf{O\_++} \\
\mathbf{V}[\![*]\!] = \mathsf{O\_Mult} & \mathbf{V}[\![\%]\!] = \mathsf{O\_Rem} & \mathbf{V}[\![/]\!] = \mathsf{O\_IntDiv} \\
\mathbf{V}[\![!]\!] = \mathsf{O\_Not} & \mathbf{V}[\![-]\!] = \mathsf{O\_Minus} & \mathbf{V}[\![\mathsf{length}]\!] = \mathsf{v\_length} \\
\mathbf{V}[\![\mathsf{size}]\!] = \mathsf{v\_size} & \mathbf{V}[\![\mathsf{expand}]\!] = \mathsf{v\_expand} & \mathbf{V}[\![\mathsf{matches}]\!] = \mathsf{v\_matches}
\end{array}
$$

Figure 3.22: Conversion of operators: $\mathbf{V}[\![\oplus]\!]$

SMT solvers determine whether an given formula is satisfied, i.e, if there is some assignment of values to constants, variables, and uninterpreted function where the formula logically evaluates to true. If yes the SMT reports *SAT* and the conditions make the formula true, else it reports *UNSAT*. The subtyping relation asserts that $\mathbf{F}'[\![T]\!](x) \Rightarrow \mathbf{F}'[\![U]\!](x)$, for all variables x. If we negate the formula, the SMT reports *SAT* when there is a value for x such that the subtyping relation does not hold. If it reports *UNSAT* then the subtyping relations holds.

The major limitation of Z3 solver regarding HeadREST semantic is the quantifiers. Z3 has by default a model-based quantifier instantiation (MBQI) for solving quantifier formulas [29]. Although being very powerful, MBQI is also slow, which in our case is exacerbated by an axiomatization with many quantifiers. Hence, we have MBQI deactivated in order to speed up formula evaluation, with the down side of increasing the number quantified formulas that can not be evaluated.

Chapter 4

SafeScript Language

To accomplish our goal of statically validating REST calls from within a programming language, we divide our work into two subgoals: first create a language similar to JavaScript with strong type validation; then extend this language with primitive REST call operators with static validation.

This chapter presents the first language, which was named SafeScript. Although the principal design decisions were made having the REST extension in mind, the language by itself has a set of relevant features, namely when seen from the perspective of static analysis and formal program validation. In this way, SafeScript can be extended for supporting other different forms of validation, thanks to its expressive type system.

4.1 Main Ideas

SafeScript is intend to be a typed safe JavaScript. JavaScript features type coercion: any value can be converted to a value of each primitive type. The coercion is necessary since JavaScript operators are total in general. For example, multiplying a string by an array (`"hello" * [12, 34]`) or accessing the 42th position of a boolean value (`true[42]`) are valid JavaScript expressions.

Universal implicit type coercion can be regarded as a plus because situations such as null dereference, division by zero, or accessing an array outside bounds do not raise runtime exceptions, greatly decreasing the possibility of runtime errors, and, consequently, of program crashes. On the other hand, semantically dubious operations can negatively affect the intended meaning of a program. The behavior of JavaScript expressions are considered surprising and unintuitive by programmers [46]. Hence, operators do not crash the program but their results can cause the program to behave wrongly, which can be even worst. The fact that JavaScript variables are not declared with a type, and therefore may contain any type of value at any time, exacerbates this problem.

SafeScript was designed to be, at the syntactic level, as close as possible to JavaScript, so that it can be used easily by JavaScript programmers. It also compiles to JavaScript, so

JavaScript programs can easily interact with SafeScript.

SafeScript adopts the HeadREST type system, not only for its envisaged REST extension, but also to provide a very precise static type checking. Each variable is declared with a type, so that the values that can be assigned to a variable are restricted to the type the variable was declared with (as happens in strongly typed languages). Operators define the type of their arguments. Each variable has also an effective type that correspond to set of values the variable may have at a given point in a program. SafeScript features flow typing, since effective types change with the program flow. The effective type of a variable is necessarily a subtype of its declared type.

Compared with other typed extensions of JavaScript, such TypeScript, the principal novelty in SafeScript is the incorporation of refinement types in the type system. On the one hand, they support the definition of many useful types. For example, the type of the second argument of the division operator can become `(x: int where x != 0)`, so that a possible division by zero can be statically detected. On the other hand, as discussed in the previous chapter, refinement types demand a semantic subtyping analysis. This is even more challenging to accomplish in an imperative language.

4.2 Syntax

The syntax of SafeScript is presented in figure 4.1. In fact, this is a simplified version of the language that has been implemented. For simplicity, the grammar of statements is ambiguous.

Expression	e	::=	$x \mid c \mid \oplus(e_1, \ldots, e_n) \mid e_1 \,?\, e_2 : e_3 \mid e \,\mathsf{in}\, T$
			$\mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l \mid [e_1, \ldots, e_n] \mid e_1[e_2]$
			$\mid f(e_1, \ldots, e_n) \mid \mathsf{forall}\ x : T\,.\,e \mid \mathsf{exists}\ x : T\,.\,e$
Scalar constant	c	::=	$n \mid s \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{null} \mid \mathsf{undefined}$
Type	T	::=	$\mathsf{any} \mid G \mid \{\} \mid \{l\colon T\} \mid T[] \mid (x\colon T \,\mathsf{where}\, e) \mid \mathsf{void}$
Basic Type	G	::=	$\mathsf{int} \mid \mathsf{boolean} \mid \mathsf{string}$
Statement	S	::=	$u = e \mid \mathsf{if}\ (\,e\,)\ S_1\ \mathsf{else}\ S_2 \mid \mathsf{while}\ (\,e_1\,)\ \mathsf{inv}\ e_2\ S$
			$\mid \mathsf{return}\ e \mid S_1; S_2 \mid \epsilon$
LHS	u	::=	$x \mid u\,[e] \mid u.l$
Function	F	::=	$T\ f\ (\,T_1\,x_1, \ldots, T_n\,x_n\,)\ \{\,U_1\ y_1 = e_1\,;\ \ldots\ ;\ U_m\ y_m = e_m\,;\ S\,\}$
			$\mid \mathsf{function}\ f\ (\,T_1\,x_1, \ldots, T_n\,x_n\,)\ \{\,e\,\}$
Declaration	D	::=	$F \mid \mathsf{var}\ T\,x \mid \mathsf{type}\ x = T$
Program	P	::=	$\epsilon \mid D\ P$

Figure 4.1: SafeScript syntax

Expressions, scalar constants, and types are those of HeadREST with three exceptions: resource types are not available because they are not necessary in SafeScript, a new type `void` and a new constant `undefined` are added, the latter denotes `void`'s unique value. In JavaScript, `undefined` represents the value of undeclared variables and unexisting object field, among others cases. In SafeScript, `undefined` represents the value returned by a function with return type `void`.

A SafeScript program is composed of functions, global variables and type abbreviations. A function definition includes the return type, its name, the list of parameters with their respective types, and the function body. In order to simplify variable scope validation, the body starts with the declaration and initialization of all local variables. The initialization is mandatory since there are no default values for all types, in particular because of refinement types. The function's body consists of a sequence of statements, that define the control flow and the return value. Statements include variable assignments, conditional statements, loops, and return statements. The left hand side of an assignment is a variable, a object field or an index access of an array. This allows to update a specific element of an array or object. 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.

SafeScript includes all operators available in HeadREST, except for the ones related with resources, URI templates and regular expressions. The only new operator is mkarray, which creates an array of a given size, filled with the value of a given expression. The sets of derived types and expressions are equal to HeadREST. From SafeScript core statements, additional derived statements were defined. Figure 4.2 presents some of them. Syntactic sugar was introduced for bindings, as shown in figure 4.3. For example, if a function needs to receive a non zero integer `x`, instead of the tedious `(i: int where i != 0) x` argument declaration, it can be simply declare as `int x where x != 0`.

$$
\begin{aligned}
u \oplus= e &\triangleq u = \oplus(u,\, e) \\
\mathsf{if}\ (\, e \,)\ S &\triangleq \mathsf{if}\ (\, e \,)\ S\ \mathsf{else}\ \epsilon \\
\mathsf{while}\ (\, e \,)\ S &\triangleq \mathsf{while}\ (\, e \,)\ \mathsf{inv}\ \mathsf{true}\ S \\
\mathsf{for}\ (\, T\ x = e_1 \,;\ e_2 \,;\ u = e_3 \,)\ S &\triangleq T\ x = e_1 \,;\ \mathsf{while}\ (\, e_2 \,)\ \{\, S \,;\ u := e_3 \,\} \\
\mathsf{for}\ (x\ \mathsf{of}\ e)\ S &\triangleq \mathsf{for}\,(\mathsf{int}\ y = 0 \,;\ y < \mathsf{length}(e) \,;\ y \mathrel{+}= 1) \\
&\qquad [e\,[y]\,/x]S \\
\mathsf{return} &\triangleq \mathsf{return}\ \mathsf{undefined}
\end{aligned}
$$

where y a fresh variable

Figure 4.2: Derived statements

$$T\ x \text{ where } e \triangleq (y : T \text{ where } [y/x]e)\ x \qquad\qquad y \notin \mathrm{fv}(T, e)$$

Figure 4.3: Derived bindings

4.3 Examples

To better understand SafeScript and its type system, we present in this section a set of examples.

Assignments Consider first a function that calculates the sucessor of an integer:

```
1 int increment(int n) {
2     n = n + 1;
3     return n;
4 }
```

Checking the validity of this program requires to perform three typing validations: both arguments of the addition operation have type `int`, after the assignment, the variable `n` has a value of type `int` (the declared type), and the expression used in the return statement matches the function return type. Intuitively, if these validations succeed, the program is valid. However, if we add to the program another function that calls `increment`, it might be necessary to guarantee that the result is greater than the value provided as input. For that, we need a more precise return type: `(i: int where i > n)`. Typing validation still holds after this change. Although the variable `x` was declared as `int`, which is not a subtype of this return type, its effective type is changed after the assignment. We could be even more precise and change the return type to `(i: int where i == n + 1)`. The resulting program would still valid.

Conditional statements Function `increment` illustrates flow typing caused by a simple assignment. Other statements, namely conditional and loops are also responsible for changing the variables effective type. Consider the following example with a conditional statement:

```
1 boolean toBoolean(int|boolean bit) {
2     if (bit in int) {
3         return bit > 0;
4     }
5     else {
6         return bit;
7     }
8 }
```

The function extracts the boolean value of a bit, which can be an integer or a boolean as declared in the function signature. The conditional statement checks whether `bit` is an integer, using the *in type* operator. In the first branch,the effective type of `bit` is **`int`**, a subtype of **`int|boolean`**, therefore it can be used as the argument of the comparison operator. In the second branch, `bit` is ensured to not be an integer, so its effective type is **`boolean`** and, hence, is a valid return value for the function.

Refinement types help to statically detect common runtime errors. An usual example is the division by zero:

```
1 int f(nat x) {
2     return 42 / x; // error!
3 }
```

The divisor argument of a division operation must be non-zero. Hence, an error is reported at line 2, since the effective type of `x` in that point (in this case is equal to the declared one) includes the value `0`. Consider now the function below, which uses the `increment` function:

```
1 int f(nat x) {
2     return 42 / increment(x); // error?
3 }
```

This is a concrete example of the usefulness of restricting the return type of the `increment` function. If the return type is simply **`int`**, then there is still an error. If the return type is instead `(i: int where i > n)`, then the type of expression `increment(x)` is `(i: int where i > 0)`, which is a subtype of what is expected in the division, and the program is valid.

Arrays In JavaScript, there is no distinction between arrays and objects, because arrays are interpreted as objects that map integers to values. In SafeScript, arrays and objects are distinct, each having its own type. An array access is only valid if it is possible to prove that the value of the index expression is within the array bounds. Note that these are fixed since the length is a primitive and immutable characteristic of an array.

```
1 void g(int[] a) {
2     a[0] = 1; // error!
3 }
```

In the example above, there is an error in line 2 since array `a` may be of length 0. This can be fixed by restricting the type of the argument to be not empty, using a refinement type:

```
1 void g((x: int[] where length(x) > 0) a) {
2     a[0] = 1;
3 }
```

These two examples also illustrate the use of type **void**. In SafeScript, all functions must return a value, that must be included in the set of values defined by the return type. As mentioned in the syntax section, **void** represents a singleton type inhabited by **undefined** value. In this way, each function defined in a SafeScript program has an implicitly **return** statement at the end of its body. As presented in figure 4.2, **return** abbreviates **return undefined** and, hence, matches the return type **void**.

Loops Iterating over the elements of an array is a common operation. In this case, it must be guarantee that in every step of the loop iteration, the array is accessed inside its bounds. The example below illustrates a function that calculates the sum of all elements of an integer array:

```
1 int sum(int[] a) {
2     int i = 0;
3     int sum = 0;
4     while (i < length(a)) {
5         sum = sum + a[i]; // error!
6         i = i + 1;
7     }
8     return sum;
9 }
```

Although the function effectively sums the elements of any integer array, the validator reports an error in the array access at line 5. The loop guard ensures that `i` does not exceed the array size but the validator is not able to prove that in every loop iteration `i` is non-negative. To overcome this issue, it is necessary to provide some additional information to the validator. There are two alternatives: add a loop invariant stating that `i` is non negative or declare `i` with a type that prevents it from being negative. The latter strategy is more adequate, since it restricts the variable in all its usages, working as a global invariant for the variable in the function. Both alternatives are presented below.

```
1  int sum(int[] a) {
2      int i = 0;
3      int sum = 0;
4      while (i < length(a))
5      inv i >= 0 {
6          sum = sum + a[i];
7          i = i + 1;
8      }
9      return sum;
10 }
```

```
11 int sum(int[] a) {
12     nat i = 0;
13     int sum = 0;
14     while (i < length(a)) {
15         sum = sum + a[i];
16         i = i + 1;
17     }
18     return sum;
19 }
```

Objects If an object is declared to have a collection of fields, then the object must have at least those fields, but may have others. Field access is valid only if it is possible to

prove that the object has that field.

```
1 {a: int} bToOne({a: int} x) {
2     x.b = 1; // error!
3     return x;
4 }
```

In the example above, parameter `x` is declared as an object with an integer field `a` and, hence, the update of field `b` at line 2 is not valid. The function could be changed so that the update of `b` is only performed if it indeed exists and is of type integer, as presented below:

```
1 {a: int} bToOne({a: int} x) {
2     if (x in {b: int}) {
3         x.b = 1;
4     }
5     return x;
6 }
```

The update is valid because of flow typing: at line 3, `x` has the effective type `{a: int, b: int}`. Note that it is not possible to declare that the return type of the function is `{a: int,b: int}`, because the field `b` may not exist or if exists, it may not be an integer. A valid and more precise (and also more complex) return type would be `(o: {a: int} where o in {b: int} ==> o.b == 1)`.

The examples presented so far illustrate the main features of SafeScript. In section 6, more complex programs are presented.

4.4 Validation

As it is the case in HeadREST, typing validation is the major challenge for the SafeScript validator. SafeScript is an imperative language with flow typing and, hence, typing validation relies on the ability to synthesize the effective type of a general expression in a given a context. If the type cannot be synthesized, then the subtyping rule cannot be applied and the SMT cannot be used directly.

To address this problem, validation is realized by a translation to Boogie [3]. Boogie is an intermediate verification language, designed to be a platform on which to build program verifiers for other languages. Boogie is also the name of the tool that checks whether programs are correct, generating verification conditions written as first order logic formulas and using the Z3 SMT solver to semantically evaluate formulae satisfiability.

4.4.1 Declarative Type System

The declarative type system for SafeScript is composed of the set of declarative judgments presented in Figure 4.5. These include judgments of the form $\Gamma \vdash T$, where Γ is an

environment and T is a type. The syntax of environments is introduced in Figure 4.4.

Variable Context	$\Gamma \ ::= \ \epsilon \mid \Gamma, x\colon T$
Assertion	$\varphi \ ::= \ e$

Figure 4.4: The environment syntax

The variable context Γ is a map that binds the declared variables, including global ones, with their respective declared type. The effective type of each variable is determined from the assertion φ, a boolean expression that characterises the possible values of the variables in a given point in the program flow. Such an expression may be added to Γ as a entry with a dummy variable: $\Gamma, \_\colon \|\varphi\|$, abbreviated as Γ, φ.

$\vdash \Gamma$	$\equiv$	context Γ is well-formed	Figure 4.6
$\Gamma \vdash T$	$\equiv$	in Γ, type T is well-formed	Figure 4.7
$\Gamma \vdash e : T$	$\equiv$	in Γ, expression e has type T	Figure 4.8
$\Gamma \vdash T_1 <: T_2$	$\equiv$	in Γ, type T_1 is subtype of T_2	Figure 4.9
$\Gamma; \varphi_1 \vdash S : (T; \varphi_2)$	$\equiv$	in Γ, statement S if executed when φ_1 holds, returns type T, and ensures that φ_2 holds in the next execution point	Figure 4.10
$\varphi \vdash \Gamma$	$\equiv$	in φ, environment Γ is consistent	Figure 4.11
$\Gamma \vdash F$	$\equiv$	in Γ, function F is well-formed	Figure 4.12

Figure 4.5: Judgments of the declarative type system

The first judgment checks whether the variable context is well formed. Figure 4.6 presents the two judgment rules. A variable context is well formed if and only if its variables have a well formed type. The rules for this judgment are presented in figure 4.7.

$$\frac{}{\vdash \epsilon}\ \text{(W-EmpEnv)} \qquad \frac{\Gamma \vdash T \quad x \notin \Gamma}{\vdash \Gamma, x\colon T}\ \text{(W-VarEnv)}$$

Figure 4.6: Context formation: $\vdash \Gamma$

The rules for type assignment to expressions (figure 4.8) are similar to the type checking rules for expressions in HeadREST. The rule for function calls is similar to the rule for operators. As this type system is not algorithmic, each rule assigns a general type. The rule T-Single restricts the type of an expression to a singleton type. Conversely, the

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{any}} \text{ (W-AnyType)} \quad \frac{\vdash \Gamma}{\Gamma \vdash G} \text{ (W-ScalarType)}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \{\}} \text{ (W-EmpObjType)} \quad \frac{\Gamma \vdash T}{\Gamma \vdash \{l\colon T\}} \text{ (W-ObjType)} \quad \frac{\Gamma \vdash T}{\Gamma \vdash T\,[]} \text{ (W-ArrType)}$$

$$\frac{\Gamma, x\colon T \vdash e : \mathsf{boolean}}{\Gamma \vdash (x\colon T \text{ where } e)} \text{ (W-RefType)} \quad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{void}} \text{ (W-VoidType)}$$

Figure 4.7: Well-formed types: $\Gamma \vdash T$

rule T-SubSump widens the type of an expression, using the subtyping relation. This is equal to HeadREST, as shown in figure 4.9.

$$\frac{\vdash \Gamma \quad (x\colon T) \in \Gamma}{\Gamma \vdash x : T} \text{ (T-ExpVar)} \quad \frac{\vdash \Gamma}{\Gamma \vdash c : \mathsf{typeof}(c)} \text{ (T-ScalarExp)}$$

$$\frac{\oplus : T_1, \ldots, T_n \to T \quad \Gamma \vdash e_i : T_i \quad \forall i \in 1..n}{\Gamma \vdash \oplus(e_1, \ldots, e_n) : T} \text{ (T-Op)}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{boolean} \quad \Gamma, \_\colon \|e_1\| \vdash e_2 : T \quad \Gamma, \_\colon \|!e_1\| \vdash e_3 : T}{\Gamma \vdash e_1 \,?\, e_2 : e_3 : T} \text{ (T-Ternary)}$$

$$\frac{\Gamma \vdash e : \mathsf{any} \quad \Gamma \vdash T}{\Gamma \vdash e \,\mathsf{in}\, T : \mathsf{boolean}} \text{ (T-InType)}$$

$$\frac{\Gamma \vdash e_i : T_i \quad \forall i \in 1..n}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1\colon T_1, \ldots, l_n\colon T_n\}} \text{ (T-ObjExp)}$$

$$\frac{\Gamma \vdash e : \{l\colon T\}}{\Gamma \vdash e.l : T} \text{ (T-ObjAcc)} \quad \frac{\Gamma \vdash e_i : T \quad \forall i \in 1..n}{\Gamma \vdash [e_1, \ldots, e_n] : T\,[]} \text{ (T-ArrExp)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_2 : (i\colon \mathsf{int} \text{ where } 0 \le i < \mathsf{length}(e_1)) \\ \Gamma \vdash e_1 : (a\colon \mathsf{any}\,[] \text{ where } a[e_2] \,\mathsf{in}\, T)\end{array}}{\Gamma \vdash e_1[e_2] : T} \text{ (T-ArrAcc)}$$

$$\frac{\begin{array}{c}f\colon (x_1\colon T_1, \ldots, x_n\colon T_n) \to T \text{ is defined} \\ \Gamma, x_1\colon T_1, \ldots, x_{i-1}\colon T_{i-1} \vdash e_i : T_i \quad \forall i \in 1..n\end{array}}{\Gamma \vdash f(e_1, \ldots, e_n) : [e_1/x_1] \ldots [e_n/x_n]T} \text{ (T-FunCall)}$$

$$\frac{\Gamma, x\colon T \vdash e : \mathsf{boolean}}{\Gamma \vdash \mathsf{forall}\ x\ :\ T\,.\,e : \mathsf{boolean}} \text{ (T-Forall)} \quad \frac{\Gamma, x\colon T \vdash e : \mathsf{boolean}}{\Gamma \vdash \mathsf{exists}\ x\ :\ T\,.\,e : \mathsf{boolean}} \text{ (T-Exists)}$$

$$\frac{\Gamma \vdash e : T_1 \quad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash e : T_2} \text{ (T-SubSump)} \quad \frac{\Gamma \vdash e : T}{\Gamma \vdash e : \langle e\colon T \rangle} \text{ (T-Single)}$$

Figure 4.8: Type assignment to expressions: $\Gamma \vdash e : T$

$$\frac{\Gamma \vdash T_1 \quad \Gamma \vdash T_2 \quad x \notin dom(\Gamma) \quad \vDash (\mathbf{F}'[\![\Gamma]\!] \wedge \mathbf{F}'[\![T_1]\!](x)) \Rightarrow \mathbf{F}'[\![T_2]\!](x)}{\Gamma \vdash T_1 <: T_2}$$

Figure 4.9: Semantic subtyping: $\Gamma \vdash T_1 <: T_2$

Rules for statements are presented in figure 4.10. Judgment $\Gamma; \varphi_1 \vdash S : (T; \varphi_2)$ means that given a context Γ and an assertion φ_1 that holds when statement S is executed, assigns to S (i) a returning type T, representing the type of the value returned by the function (if any), and (ii) a new assertion φ_2, that represents the condition that holds in the next execution point. The condition φ_1 is usually called the pre-condition. The condition φ_2 is similar to a post-condition, in the sense that it captures the changes in the effective types of the variables in the context that results form the execution of S. However, since SafeScript includes a return statement, it also captures whether the next execution point exists.

$$\frac{\Gamma, [e/u]\varphi \vdash u : T_1 \quad \Gamma, [e/u]\varphi \vdash e : T_2 \quad \varphi \vdash \Gamma}{\Gamma; [e/u]\varphi \vdash u = e : (\mathsf{empty}; \varphi)} \text{ (T-Assign)}$$

$$\frac{\Gamma, \varphi_1 \vdash e : \mathsf{boolean} \quad \Gamma; \varphi_1 \,\&\, e \vdash S_1 : (T; \varphi_2) \quad \Gamma; \varphi_1 \,\&\, !e \vdash S_2 : (T; \varphi_2)}{\Gamma; \varphi_1 \vdash \mathsf{if}\ (\ e\ )\ S_1\ \mathsf{else}\ S_2 : (T; \varphi_2)} \text{ (T-If)}$$

$$\frac{\begin{array}{c}\Gamma, \varphi \vdash e_1 : \mathsf{boolean} \quad \Gamma, \varphi \vdash e_2 : \mathsf{boolean} \\ \Gamma, e_1 \,\&\, e_2 \vdash S : (T; (e_1\ \mathsf{in}\ \mathsf{boolean}) \,\&\, (e_2\ \mathsf{in}\ \mathsf{boolean}) \,\&\&\, e_2)\end{array}}{\Gamma; \varphi \vdash \mathsf{while}\ (\ e_1\ )\ \mathsf{inv}\ e_2\ S : (T; !e_1 \,\&\, e_2)} \text{ (T-While)}$$

$$\frac{\Gamma, \varphi \vdash e : T}{\Gamma; \varphi \vdash \mathsf{return}\ e : (T; \mathsf{false})} \text{ (T-Return)}$$

$$\frac{\Gamma; \varphi_1 \vdash S_1 : (T; \varphi_2) \quad \Gamma; \varphi_2 \vdash S_2 : (T; \varphi_3)}{\Gamma; \varphi_1 \vdash S_1; S_2 : (T; \varphi_3)} \text{ (T-Seq)} \qquad \frac{\vdash \Gamma, \varphi}{\Gamma; \varphi \vdash \epsilon : (\mathsf{empty}; \varphi)} \text{ (T-EmpStat)}$$

$$\frac{\Gamma; \varphi_1 \vdash S : (T_1; \varphi_2) \quad T_1 <: T_2}{\Gamma; \varphi_1 \vdash S : (T_2; \varphi_2)} \text{ (T-SubtStat)}$$

$$\frac{\Gamma \vDash \varphi_1 \rightarrow \varphi_2 \quad \Gamma; \varphi_2 \vdash S : (T; \varphi_3)}{\Gamma; \varphi_1 \vdash S : (T; \varphi_3)} \text{ (T-LStr)}$$

$$\frac{\Gamma; \varphi_1 \vdash S : (T; \varphi_2) \quad \Gamma \vDash \varphi_2 \rightarrow \varphi_3}{\Gamma; \varphi_1 \vdash S : (T; \varphi_3)} \text{ (T-RWeak)}$$

Figure 4.10: Type checking statements: $\Gamma; \varphi_1 \vdash S : (T; \varphi_2)$

Flow typing is inspired in the Floyd-Hoare logic [25, 35]. In the T-If rule, the branch condition is added to the pre-condition φ_1 for the evaluation of the statements of both

branches, which should produce a common type T and a common assertion φ_2 as post-condition.

The post-condition of the T-While rule depends on the invariant condition specified in the statement. This invariant must be true immediately before and after the evaluation of the guard. If the invariant and the guard hold, the loop body must preserve the invariant. Additionally, if the invariant and the guard hold, the loop body must preserve the boolean type of the guard and the of invariant. This is achieved done by strengthening the post-condition of the loop body with two *in type* conditions. The post-condition of the loop statement consists of the invariant and the negation of the guard, discarding completely the pre-condition φ.

The T-Return rule states that the type of a return statement is the type of its expression and that false holds at the next execution point, meaning that there is no next execution point. This rule works together with the sequential composition rule, which expects the same type from both statements. If the first statement is a return then false becomes a pre-condition of the second statement and the same retuning type can be concluded. The T-EmpStat rule assigns to an empty statement the empty return type. These rules imply that typing errors in unreachable code are just ignored.

The T-SubtStat rule shrinks the returning type assigned to a statement. The two rules, T-LStr and T-RWeak, are for, respectively, strengthening the pre-condition and weakening the post-condition of a statement.

The post-condition of the rule T-Assign is defined as in the assignment axiom of Hoare logic. In addition, the precondition must entail that the left hand sides are well formed and that the environment is consistent with the post-condition. This means that the effective type of all variables after the execution of the assignment is still a subtype of the type in the context. Figure 4.11 introduces the rules for checking the consistent of the environment.

$$\frac{}{\varphi \vdash \epsilon}\ \text{(W-EmpEnv)} \qquad \frac{\varphi \vdash \Gamma \quad x \notin \Gamma \quad \Gamma, x\colon \mathsf{any}, \varphi \vdash [x] <\colon T}{\varphi \vdash \Gamma, x\colon T}\ \text{(W-VarEnv)}$$

Figure 4.11: Consistent environment: $\varphi \vdash \Gamma$

An environment is consistent if all of the variables in the variable context have an effective type, represented by the singleton type of the variable, subtype of the variable declared type. The effective type is determined only by the assertion, and not by the declared type, since it is not considered in the subtyping evaluation. Note that all variables must be verified after each update, and not only the particular variable invoked in the update, because types can be refined to depend on other variables. For example, suppose we have `x: int` and `y: (i: int where i > x)`, i.e, `y` is greater than `x`. If `x` is updated to a value greater than `y`, then `y` value no longer belongs to its type, although its value did not change in the update.

Finally, the top judgment of the declarative type system checks the formation of function declarations. Its single rule is presented in figure 4.12. The rule states that the types of the parameters and local variables must be well formed. Since types of the parameters can refer the previous parameters in the list, it is possible to define restrictions such as that the second parameter is an integer greater than the first. The rule also states that, given an environment that reflects the function parameters and types, the initialization of the local variables, followed by the body of the function and a return statement, must match the function type. Recall that the final return statement, (of type **`void`**), ensures that a function always has a return type. The variable context passed to the function rule should contain all the global variables of the program, including the ones declared after the function.

$$\frac{\begin{array}{c}\Gamma, x_1 : T_1, \ldots, x_{i-1} : T_{i-1} \vdash T_i \quad (\forall i \in 1..m) \\ \Gamma, x_1 : T_1, \ldots, x_k : T_k, \mathsf{true} \vdash x_1 = e_{k+1} \; ; \; \ldots \; ; \; x_m = e_m \; ; \; S \; ; \; \mathsf{return} : (T; \varphi)\end{array}}{\Gamma \vdash T \; f \; (T_1 \, x_1, \ldots, T_k \, x_k) \; \{ T_{k+1} \; x_{k+1} = e_{k+1} \; ; \; \ldots \; ; \; T_m \; x_m = e_m \; ; \; S \}}$$

Figure 4.12: Well-formed functions: $\Gamma \vdash F$

4.4.2 Translation to Boogie

To validate a SafeScript program we translate it to Boogie. We add a set of type-related assertions, and use Boogie tool to verify whether all assertions hold. As in the validation of HeadREST using directly aN SMT, the types are not directly translated to Boogie, but the relation *in type*. The axiomatization of the typing relation is inspired in Whiley [56].

Whiley is a multi-paradigm programming language with extended static checking performed by an internal Automated Theorem Proving (ATP). Utting et al. proposed and tested the replacement of Whiley's internal ATP to Boogie for the verification phase of the compiler [73]. Since Whiley has a type system similar to that of SafeScript (except for refinement types), the authors faced similar to ours problems, and found cleaver solutions to encode the Whiley type system into Boogie. However, the Whiley typing validation is done prior to the usage of either the internal ATP or the Boogie tool. These are only used to prove verification conditions. In our case, all typing validation is done via Boogie.

The SafeScript Boogie axiomatization is presented in the appendix C. Values and types are modeled using sets, as for refinement types. All SafeScript values, independently of their type, belong to a single Boogie type: `Value`. For each type are introduced functions and axioms that define the subset of values of the type. All types have a common form. For example, given a type `X` and its Boogie internal representation `Y`, the base functions and axioms are the following.

```
1 function isX(Value) returns (bool);
2 function toX(Value) returns (Y);
```

```
3  function fromX(Y) returns (Value);
4
5  axiom (forall y: Y :: isX(fromX(y)));
6  axiom (forall y: Y :: toX(fromX(y)) == y);
7  axiom (forall v: Value :: isX(v) ==> fromX(toX(v)) == v);
```

Function `isX`, checks whether a value belongs to type `X`, and returns a Boogie primitive boolean. Function `toX` converts Boogie type to its internal representation `Y`, and `fromX` performs the inverse operation. The axioms introduce 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.

For example, for the scalar type **`int`**, the following functions and axioms are introduced.

```
1  function isInt(Value) returns (bool);
2  function toInt(Value) returns (int);
3  function fromInt(int) returns (Value);
4
5  axiom (forall i: int :: isInt(fromInt(i)));
6  axiom (forall i: int :: toInt(fromInt(i)) == i);
7  axiom (forall v: Value :: isInt(v) ==> fromInt(toInt(v)) == v);
```

In Dminor, Bierman et al. translate the opertator (`e in T`) into a first order logic formula for each type `T`. SafeScript uses a similar process, relying on the functions defined in the axiomatization. Figure 4.13 presents the transformation of the *in* predicate into Boogie boolean expressions.

$$
\begin{aligned}
\mathbf{F}'[\![\mathsf{any}]\!](e) &= \mathsf{true} \\
\mathbf{F}'[\![\mathsf{int}]\!](e) &= \mathsf{isInt}(e) \\
\mathbf{F}'[\![\mathsf{boolean}]\!](e) &= \mathsf{isBool}(e) \\
\mathbf{F}'[\![\mathsf{string}]\!](e) &= \mathsf{isString}(e) \\
\mathbf{F}'[\![T\,[]]\!](e) &= \mathsf{isArray}(e) \wedge (\textbf{forall}\ y\colon \mathsf{int} :: \mathsf{isValidIndex}(e, y) \\
&\qquad \Rightarrow \mathbf{F}'[\![T]\!](\mathsf{getIndexValue}(e, y))) \\
\mathbf{F}'[\![\{\}]\!](e) &= \mathsf{isObject}(e) \\
\mathbf{F}'[\![\{l\colon T\}]\!](e) &= \mathsf{isObject}(e) \wedge \mathsf{hasField}(e, l) \wedge \mathbf{F}'[\![T]\!](\mathsf{getFieldValue}(e, l)) \\
\mathbf{F}'[\![(x\colon T \text{ where } e_1)]\!](e) &= \mathbf{F}'[\![T]\!](e) \wedge \mathbf{V}[\![[e/x]e_1]\!] == \mathbf{V}[\![\mathsf{true}]\!]
\end{aligned}
$$

where y is a fresh variable

Figure 4.13: Translation of *in type* predicates: $\mathbf{F}'[\![T]\!](e)$

The translation of expressions is presented in figure 4.14. The translation of operators is presented in figure 4.15. To each operator corresponds an interpreted function defined in the axiomatization (appendix C).

$$
\begin{aligned}
\mathbf{V}[\![\mathsf{forall}\ x : T\,.\,e]\!] &= \mathsf{fromBool}((\textbf{forall}\ x\colon \mathsf{Value} :: \mathbf{F}'[\![T]\!](x) \Rightarrow \mathbf{V}[\![e]\!] == \mathbf{V}[\![\mathsf{true}]\!])) \\
\mathbf{V}[\![\mathsf{exists}\ x : T\,.\,e]\!] &= \mathsf{fromBool}((\textbf{exists}\ x\colon \mathsf{Value} :: \mathbf{F}'[\![T]\!](x) \wedge \mathbf{V}[\![e]\!] == \mathbf{V}[\![\mathsf{true}]\!])) \\
\mathbf{V}[\![e_1\ ?\ e_2 : e_3]\!] &= \textbf{if}\ \mathbf{V}[\![e_1]\!] == \mathbf{V}[\![\mathsf{true}]\!]\ \textbf{then}\ \mathbf{V}[\![e_2]\!]\ \textbf{else}\ \mathbf{V}[\![e_3]\!] \\
\mathbf{V}[\![\oplus(e_1, \ldots, e_n)]\!] &= \mathbf{V}[\![\oplus]\!](\mathbf{V}[\![e_1]\!], \ldots, \mathbf{V}[\![e_n]\!]) \\
\mathbf{V}[\![e\ \mathsf{in}\ T]\!] &= \mathsf{fromBool}(\mathbf{F}'[\![T]\!](\mathbf{V}[\![e]\!])) \\
\mathbf{V}[\![e_1\,[e_2]]\!] &= \mathsf{getIndexValue}(\mathbf{V}[\![e_1]\!], \mathsf{toInt}(\mathbf{V}[\![e_2]\!])) \\
\mathbf{V}[\![e.l]\!] &= \mathsf{getFieldValue}(\mathbf{V}[\![e]\!], l) \\
\mathbf{V}[\![f(e_1, \ldots, e_n)]\!] &= f(\mathbf{V}[\![e_1]\!], \ldots, \mathbf{V}[\![e_n]\!]) \\
\mathbf{V}[\![\{l_1\colon e_1, \ldots, l_n\colon e_n\}]\!] &= \mathsf{fromObject}(\mathsf{objectConst}()[l_1 := \mathsf{maybeOf}(\mathbf{V}[\![e_1]\!])] \\
&\qquad \ldots [l_n := \mathsf{maybeOf}(\mathbf{V}[\![e_n]\!])]) \\
\mathbf{V}[\![[e_1, \ldots, e_n]]\!] &= \mathsf{fromArray}(\mathsf{arrayConst}()[0 := \mathsf{maybeOf}(\mathbf{V}[\![e_1]\!])] \\
&\qquad \ldots [n-1 := \mathsf{maybeOf}(\mathbf{V}[\![e_n]\!])], n) \\
\mathbf{V}[\![\mathsf{true}]\!] &= \mathsf{True} \\
\mathbf{V}[\![\mathsf{false}]\!] &= \mathsf{False} \\
\mathbf{V}[\![n]\!] &= \mathsf{fromInt}(n) \\
\mathbf{V}[\![\text{"}c_1 \ldots\ c_n\text{"}]\!] &= \mathsf{fromString}(\mathsf{emptyString}()[0 := c_1] \ldots [n-1 := c_n]\,, n) \\
\mathbf{V}[\![\mathsf{null}]\!] &= \mathsf{Null} \\
\mathbf{V}[\![\mathsf{undefined}]\!] &= \mathsf{Undefined} \\
\mathbf{V}[\![x]\!] &= x
\end{aligned}
$$

Figure 4.14: Translation of expressions: $\mathbf{V}[\![e]\!]$

$$
\begin{array}{llll}
\mathbf{V}[\![<=>]\!] = \mathsf{equi} & \mathbf{V}[\![=>]\!] = \mathsf{imp} & \mathbf{V}[\![|]\!] = \mathsf{or} & \mathbf{V}[\![\&]\!] = \mathsf{and} \\
\mathbf{V}[\![==]\!] = \mathsf{eq} & \mathbf{V}[\![!=]\!] = \mathsf{ne} & \mathbf{V}[\![<]\!] = \mathsf{lt} & \mathbf{V}[\![<=]\!] = \mathsf{le} \\
\mathbf{V}[\![>]\!] = \mathsf{gt} & \mathbf{V}[\![>=]\!] = \mathsf{ge} & \mathbf{V}[\![+]\!] = \mathsf{sum} & \mathbf{V}[\![-]\!] = \mathsf{sub} \\
\mathbf{V}[\![++]\!] = \mathsf{concat} & \mathbf{V}[\![*]\!] = \mathsf{mult} & \mathbf{V}[\![\%]\!] = \mathsf{rem} & \mathbf{V}[\![/]\!] = \mathsf{div} \\
\mathbf{V}[\![!]\!] = \mathsf{neg} & \mathbf{V}[\![-]\!] = \mathsf{min} & \mathbf{V}[\![\mathsf{mkarray}]\!] = \mathsf{mkArray} & \mathbf{V}[\![\mathsf{length}]\!] = \mathsf{length} \\
\mathbf{V}[\![\mathsf{size}]\!] = \mathsf{size} & & &
\end{array}
$$

Figure 4.15: Translation of operator names: $\mathbf{V}[\![\oplus]\!]$

Integer and boolean values are translated to the corresponding boogie primitive type. Boogie does not support strings, so we represent them using a map of integers to integers, the first representing the index, and the second the unicode of each character. An additional type is necessary for objects and arrays: `MaybeValue`, that represents a concrete value or its absence. Objects and arrays are represented using maps from, fields and integers, respectively, to a `MaybeValue`. Since Boogie maps are total, i.e., are defined on all

their domains, `MaybeValue` helps in checking whether an object has a field and relating the array length against a given index. The array length, as the string size, is a constant that is given in the construction of a value.

Figure 4.16 presents the translation of SafeScript expressions with validation. The translation returns a list of Boogie statements where validations are done via assertions, and where the resulting value is placed in the given variable. The assertions are according in the declarative type assignment rules for expressions (figure 4.8)

$$
\begin{aligned}
\mathbf{V}^*[\![\mathsf{forall}\ x_1 : T \,.\, e]\!](x) &= \mathbf{W}[\![T]\!]\ \textbf{assume}\ \mathbf{F}'[\![T]\!](y_1);\ \mathbf{V}^*[\![[y_1/x_1]e]\!](y_2)\\
&\quad\ \textbf{assert}\ \mathbf{F}'[\![\mathsf{boolean}]\!](y_2);\\
&\quad\ x := \mathbf{V}[\![\mathsf{forall}\ x_1 : T \,.\, x_1 == y_1 \Rightarrow y_2]\!];\\
\mathbf{V}^*[\![\mathsf{exists}\ x_1 : T \,.\, e]\!](x) &= \mathbf{W}[\![T]\!]\ \textbf{assume}\ \mathbf{F}'[\![T]\!](y_1);\ \mathbf{V}^*[\![[y_1/x_1]e]\!](y_2)\\
&\quad\ \textbf{assert}\ \mathbf{F}'[\![\mathsf{boolean}]\!](y_2);\\
&\quad\ x := \mathbf{V}[\![\mathsf{exists}\ x_1 : T \,.\, x_1 == y_1 \wedge y_2]\!];\\
\mathbf{V}^*[\![e_1\ ?\ e_2 : e_3]\!](x) &= \mathbf{V}^*[\![e_1]\!](y_1)\ \textbf{assert}\ \mathbf{F}'[\![\mathsf{boolean}]\!](y_1);\\
&\quad\ \textbf{if}\,(y_1 == \mathbf{V}[\![\mathsf{true}]\!])\ \{\ \mathbf{V}^*[\![e_2]\!](x)\ \}\ \textbf{else}\ \{\ \mathbf{V}^*[\![e_3]\!](x)\ \}\\
\mathbf{V}^*[\![\oplus(e_1,\ldots,e_n)]\!](x) &= \mathbf{V}^*[\![e_1]\!](y_1)\ldots\mathbf{V}^*[\![e_n]\!](y_n)\\
&\quad\ \textbf{assert}\ \mathbf{F}'[\![T_1]\!](y_1)\wedge\cdots\wedge\mathbf{F}'[\![T_n]\!](y_n);\\
&\quad\ x := \mathbf{V}[\![\oplus(y_1,\ldots,y_n)]\!];\ \text{where}\ \oplus\colon T_1,\ldots,T_n\to T\\
\mathbf{V}^*[\![e\ \mathsf{in}\ T]\!](x) &= \mathbf{V}^*[\![e]\!](y)\ \mathbf{W}[\![T]\!]\ x := \mathbf{V}[\![y\ \mathsf{in}\ T]\!];\\
\mathbf{V}^*[\![e_1[e_2]]\!](x) &= \mathbf{V}^*[\![e_1]\!](y_1)\ \mathbf{V}^*[\![e_2]\!](y_2)\ \textbf{assert}\ \mathbf{F}'[\![\mathsf{any}\ []]\!](y_1)\wedge\\
&\quad\ \mathbf{F}'[\![(i : \mathsf{nat}\ \mathsf{where}\ i < \mathsf{length}(y_1)]\!](y_2);\ x := \mathbf{V}[\![y_1[y_2]]\!];\\
\mathbf{V}^*[\![e.l]\!](x) &= \mathbf{V}^*[\![e]\!](y)\ \textbf{assert}\ \mathbf{F}'[\![\{l\colon \mathsf{any}\}]\!](y);\ x := \mathbf{V}[\![y.l]\!];\\
\mathbf{V}^*[\![f(e_1,\ldots,e_n)]\!](x) &= \mathbf{V}^*[\![e_1]\!](y_1)\ldots\mathbf{V}^*[\![e_n]\!](y_n)\\
\text{if } f \text{ pure function}\quad &\quad\ \textbf{assert}\ \mathbf{F}'[\![T_1]\!](y_1)\wedge\cdots\wedge\mathbf{F}'[\![T_n]\!](y_n);\\
&\quad\ x := \mathbf{V}[\![f(y_1,\ldots,y_n)]\!];\ \text{where}\ f\colon T_1,\ldots,T_n\to T\\
\mathbf{V}^*[\![f(e_1,\ldots,e_n)]\!](x) &= \mathbf{V}^*[\![e_1]\!](y_1)\ldots\mathbf{V}^*[\![e_n]\!](y_n)\\
\text{if } f \text{ not pure function}\quad &\quad\ \textbf{call}\ x := \mathbf{V}[\![f(y_1,\ldots,y_n)]\!];\ \text{where}\ f\colon T_1,\ldots,T_n\to T\\
\mathbf{V}^*[\![\{l_1\colon e_1,\ldots,l_n\colon e_n\}]\!](x) &= \mathbf{V}^*[\![e_1]\!](y_1)\ldots\mathbf{V}^*[\![e_n]\!](y_n)\ x := \mathbf{V}[\![\{l_1\colon y_1,\ldots,l_n\colon y_n\}]\!];\\
\mathbf{V}^*[\![[e_1,\ldots,e_n]]\!](x) &= \mathbf{V}^*[\![e_1]\!](y_1)\ldots\mathbf{V}^*[\![e_n]\!](y_n)\ x := \mathbf{V}[\![[y_1,\ldots,y_n]]\!];\\
\mathbf{V}^*[\![c]\!](x) &= x := \mathbf{V}[\![c]\!];\\
\mathbf{V}^*[\![x]\!](z) &= z := \mathbf{V}[\![x]\!];
\end{aligned}
$$

where $y, y_1, \ldots, y_n$ are fresh variables

Figure 4.16: Translation of expressions with type validation: $\mathbf{V}^*[\![e]\!](x)$

These translation rules follow a general pattern: the inner sub-expressions are recursively translated (with validation), then it is checked whether the variables that contain the sub-expressions have the expected type, and finally, value translation is used to assign the

expression to the given variable. The assert commands are aligned with the expected types presented in declarative system, namely in type assignment to expressions (figure 4.8).

Types must also be validated since refinement types contain expressions. Figure 4.17 presents the translation of the well formation of SafeScript into Boogie statements. The validation follows the rules in figure 4.7.

$$
\begin{aligned}
\mathbf{W}[\![T\,[]]\!] &= \mathbf{W}[\![T]\!] \\
\mathbf{W}[\![\{l\colon T\}]\!] &= \mathbf{W}[\![T]\!] \\
\mathbf{W}[\![(x\colon T \text{ where } e)]\!] &= \mathbf{W}[\![T]\!] \ \textbf{assume}\ \mathbf{F}'[\![T]\!](y_1);\ \mathbf{V}^*[\![[y_1/x]e]\!](y_2) \\
&\qquad\qquad \textbf{assert}\ \mathbf{F}'[\![\mathsf{boolean}]\!](y_2); \\
\text{otherwise}\quad \mathbf{W}[\![T]\!] &= \epsilon
\end{aligned}
$$

where y_1 and y_2 are fresh variables

Figure 4.17: Translation of type formation: $\mathbf{W}[\![T]\!]$

SafeScript statements are translated directly to equivalent Boogie statements, along with the necessary validations. Initial versions of Boogie only had goto and return statements for controlling the flow of execution [3]. Boogie version 2 includes control flow structures as found in high-level languages, including conditionals and while loops, which are subsequently transform into goto statements [45]. Figure 4.18 presents the translation of SafeScript statements to Boogie 2.

$$
\begin{aligned}
\mathbf{B}[\![u = e]\!] &= \mathbf{V}^*[\![e]\!](y)\ \mathbf{U}[\![u]\!](y) \\
\mathbf{B}[\![\mathsf{if}\ (\ e\ )\ S_1\ \mathsf{else}\ S_2]\!] &= \mathbf{V}^*[\![e]\!](y)\ \textbf{assert}\ \mathbf{F}'[\![\mathsf{boolean}]\!](y); \\
&\qquad\qquad \textbf{if}\ (y == \mathbf{V}[\![\mathsf{true}]\!])\ \{\ \mathbf{B}[\![S_1]\!]\ \}\ \textbf{else}\ \{\ \mathbf{B}[\![S_2]\!]\ \} \\
\mathbf{B}[\![\mathsf{while}\ (\ e_1\ )\ \mathsf{inv}\ e_2\ S]\!] &= \mathbf{V}^*[\![e_1]\!](y_1)\ \mathbf{V}^*[\![e_2]\!](y_2) \\
&\qquad\qquad \textbf{assert}\ \mathbf{F}'[\![\mathsf{boolean}]\!](y_1) \wedge \mathbf{F}'[\![\mathsf{boolean}]\!](y_2); \\
&\qquad\qquad \textbf{while}\ (\mathbf{V}[\![y_1]\!] == \mathbf{V}[\![\mathsf{true}]\!]) \\
&\qquad\qquad \textbf{invariant}\ \mathbf{V}[\![e_2]\!] == \mathbf{V}[\![\mathsf{true}]\!]; \\
&\qquad\qquad \textbf{free invariant}\ \mathbf{F}'[\![T_1]\!](z_1) \wedge \cdots \wedge \mathbf{F}'[\![T_n]\!](z_n); \\
&\qquad\qquad \{\ \mathbf{B}[\![S]\!]\ \mathbf{V}^*[\![e_1]\!](y_1)\ \textbf{assert}\ \mathbf{F}'[\![\mathsf{boolean}]\!](y_1);\ \} \\
\mathbf{B}[\![\mathsf{return}\ e]\!] &= \mathbf{V}^*[\![e]\!](\mathsf{result})\ \textbf{return}; \\
\mathbf{B}[\![S_1; S_2]\!] &= \mathbf{B}[\![S_1]\!]\ \mathbf{B}[\![S_2]\!] \\
\mathbf{B}[\![\epsilon]\!] &= \epsilon
\end{aligned}
$$

where y, y_1, y_2 and y_3 are fresh variables, $z_1, \ldots, z_n$ the variables in scope, and $T_1, \ldots, T_n$ declared types of those variables

Figure 4.18: Translation of statements: $\mathbf{B}[\![S]\!]$

If and while statements require a validation of the condition and the invariant, as in the declarative type system (figure 4.10). In the case of while, the guard must be verified before every execution, so is added at the end of the loop body an additional type validation. The invariant is a Boogie primitive. The Boogie code checks whether the given invariant is a valid loop invariant, i.e., if it holds after the loop condition evaluation in each loop step. The declared type of each variable is also considered to be an invariant at every loop step. These invariants are prefixed with the keyword **free** which indicates to Boogie that there is no need to verify whether they hold, since such invariants are validated after every update.

One important detail about the invariant validation is the exact moment where it must be true. In Floyd-Hoare logic, expressions do not have side effects, so it is not relevant if the invariant must be considered before or after the evaluation of the loop condition. Since SafeScript expressions may have a side effect in global variables (by a function call), we make sure that the invariant should be true exactly after the loop condition evaluation.

The translation rule for return places the return expression in a special variable result, and ends the method execution. In Boogie, the value of a procedure must be placed in a variable declared in the method header, result in our case.

The variable update complexity demands a particular set of translation rules, which are presented in figure 4.19. They make the necessary validation of the left hand side, update the target variable, and check whether the declared type is respected. The functions arrayUpdate and objectUpdate are defined in the Boogie axiomatization.

$$
\begin{aligned}
\mathbf{U}[\![x]\!](e) &= x := e;\ \textbf{assert}\ \mathbf{F}'[\![T_1]\!](z_1) \wedge \cdots \wedge \mathbf{F}'[\![T_n]\!](z_n);\\
\mathbf{U}[\![u\,[e_1]]\!](e) &= \mathbf{V}^*[\![u]\!](y_1)\ \mathbf{V}^*[\![e_1]\!](y_2)\ \mathbf{V}^*[\![y_1[y_2]]\!](y_3)\ \mathbf{U}[\![u]\!](\mathsf{arrayUpdate}(y_1, y_2, e))\\
\mathbf{U}[\![u.l]\!](e) &= \mathbf{V}^*[\![u]\!](y_1)\ \mathbf{V}^*[\![y_1.l]\!](y_2)\ \mathbf{U}[\![u]\!](\mathsf{objectUpdate}(y_1, l, e))
\end{aligned}
$$

where y_1, y_2 and y_3 are fresh variables, $z_1, \ldots, z_n$ the variables in scope, and $T_1, \ldots, T_n$ types of those variables

Figure 4.19: Translation of variable update: $\mathbf{U}[\![u]\!](e)$

As discussed in the declarative type system of section 4.4.1, after an assignment it must be verified that *all* variables in scope have a effective type subtype of the declared type. The translation rule for variable update reflects this idea. However, many of these checks are redundant because there may be no connection between the updated variable and the declared type of the variable to be checked. To reduce the assertions that the Boogie must prove, and therefore the time of typing validation, it is possible to create a graph with type dependencies of the variables, and then check only the necessary variables according to the dependency graph.

The translation rules proposed in figure 4.19 have redundant verification of the left hand side. For example, the translation of the expression `x.a.b` validates two times the

access `x.a`. An equivalent translation, less intuitive, but optimized in the validation, is presented in figure 4.20. Here, $\bar{S}$ and $\bar{e}$ represent Boogie statements and expressions, respectively.

$$\begin{aligned}
\mathbf{U}'[\![x_0]\!](x, \bar{S}, \bar{e}) &= x := x_0;\ \bar{S}\ x_0 := \bar{e};\ \textbf{assert}\ \mathbf{F}'[\![T_1]\!](z_1) \wedge \cdots \wedge \mathbf{F}'[\![T_n]\!](z_n);\\
\mathbf{U}'[\![u\,[e]]\!](x, \bar{S}, \bar{e}) &= \mathbf{U}'[\![u]\!](y_1, \mathbf{V}^*[\![e]\!](y_2)\ \mathbf{V}^*[\![y_1[y_2]]\!](x)\ \bar{S}, \mathsf{arrayUpdate}(y_1, y_2, \bar{e}))\\
\mathbf{U}'[\![u.l]\!](x, \bar{S}, \bar{e}) &= \mathbf{U}'[\![u]\!](y, \mathbf{V}^*[\![y.l]\!](x)\ \bar{S}, \mathsf{objectUpdate}(y, l, \bar{e}))
\end{aligned}$$

where y, y_1 and y_2 are fresh variables, $z_1, \ldots, z_n$ the variables in scope, and $T_1, \ldots, T_n$ types of those variables

Figure 4.20: Translation of variable update optimized: $\mathbf{U}'[\![u]\!](x, \bar{S}, \bar{e})$

Figure 4.21 presents the transformation of SafeScript functions into Boogie procedures, making the necessary validations according with the declarative system (figure 4.12).

$$\begin{aligned}
&\mathbf{B}[\![T\ f\ (\,T_1\,x_1, \ldots, T_n\,x_n\,)\ \{\,U_1\ y_1 = e_1\,;\ \ldots\ ;\ U_m\ y_m = e_m\,;\ S\,\}]\!] =\\
&\textbf{procedure}\ f(x_1 : \mathsf{Value}, \ldots, x_n : \mathsf{Value})\ \textbf{returns}\ (\mathsf{result} : \mathsf{Value})\\
&\textbf{requires}\ \mathbf{F}'[\![T_1]\!](x_1) \wedge \cdots \wedge \mathbf{F}'[\![T_n]\!](x_n);\\
&\textbf{modifies}\ g_1, \ldots, g_p;\\
&\textbf{ensures}\ \mathbf{F}'[\![T]\!](\mathsf{result});\\
&\{\ \textbf{var}\ z_1, \ldots, z_k,\ y_1, \ldots, y_m,\ w_1, \ldots, w_n, : \mathsf{Value};\\
&w_1 := x_1; \ldots\ w_n := x_n;\\
&\mathbf{W}[\![T_1]\!] \ldots \mathbf{W}[\![T_n]\!]\ \mathbf{W}[\![U_1]\!] \ldots \mathbf{W}[\![U_m]\!]\ \mathbf{W}[\![T]\!]\\
&\mathbf{B}[\![y_1 = e_1\,;\ \ldots\ ;\ y_m = e_m\,; [w_1/x_1] \ldots [w_n/x_n]S\,;\ \mathsf{return}]\!]\ \}
\end{aligned}$$

where $y_1, \ldots, y_n$ are fresh variables representing each parameter, $z_1, \ldots, z_k$ are the respective fresh local variables of the procedure, and $g_1, \ldots, g_p$ are the global variables

Figure 4.21: Translation of function definitions: $\mathbf{B}[\![F]\!]$

Boogie procedures parameters are immutable. Since in SafeScript parameters are mutable, new variables are necessary to replace the parameters in the function body. The requires clause validates the calls made to this procedure from this or other methods, i.e., it checks whether the call arguments belong to the parameters declared types. The modifies header asserts that all global variables can be modified by the procedure. The ensures condition checks whether in all returning points of the procedure the returned result matches the expected function return type, and asserts the return conditions for the procedure caller. As in SafeScript, Boogie local variables must be declared at the beginning of a procedure, so all fresh variables require by the translation must also be declared before the Boogie statements.

The declaration of global variables is presented in figure 4.22. The initialization expression is discarded since Boogie does not support global variable initialization, and when a procedure is called the only fact that the procedure knows about the value of each global variable is that belongs to its type. The global variables declaration types and initial values are validated via an additional procedure with a fresh name f.

$$\mathbf{B}[\![T\ x = e]\!] = \textbf{var}\ x : \mathsf{Value}\ \textbf{where}\ \mathbf{F}'[\![T]\!](x);\ \mathbf{V}[\![T\ f()\ \{\ \mathsf{return}\ e\ \}]\!]$$

where f is a fresh procedure name

Figure 4.22: Translation of global variable declarations: $\mathbf{B}[\![T\ x = e]\!]$

To complete the section we present an example of the translation of a simple SafeScript program to Boogie: the example of the section 4.3, where an invalid array access is attempted.

```
1  void g(int[] a) {
2      a[0] = 1; // error!
3  }
```

Below is the Boogie code generated from the previous SafeScript function, according to the translation presented in this section. This is an excerpt from the code sent to the Boogie verifier; the axiomatization (appendix C) was removed. The variables with the prefix `var#` are fresh variables.

```
272 procedure g(a: Value) returns (.result: Value)
273     requires (.isArray(a) && (forall var#8: int :: .isValidIndex(a,
        var#8) ==> .isInt(.getIndexValue(a, var#8))));
274     modifies ;
275     ensures .isVoid(.result);
276 {
277     var var#a, var#0, var#1, var#2, var#3, var#4, var#6: Value;
278     var#a := a;
279     var#0 := .fromInt(1);
280     var#2 := var#a;
281     var#3 := .fromInt(0);
282     var#4 := var#2;
283     assert (.isArray(var#4) && (forall var#5: int ::
        .isValidIndex(var#4, var#5) ==> true));
284     var#6 := var#3;
285     assert .isInt(var#6);
286     assert 0 <= .toInt(var#6) && .toInt(var#6) < .arraylen(var#4);
287     var#1 := .getIndexValue(var#4, .toInt(var#6));
288     var#a := .arrayUpdate(var#2, var#3, var#0);
289     assert (.isArray(var#a) && (forall var#7: int ::
        .isValidIndex(var#a, var#7) ==> .isInt(.getIndexValue(var#a,
```

```
        var#7))));
290      assert .isVoid(.Nothing);
291      .result := .Nothing;
292  }
```

The assertion in line 286, which checks the validity of the array access index, cannot be proven by Boogie, since it cannot show that the index is within the arrays bounds. The Boogie verifier produces the following output reporting the error:

```
1  code.bpl(286,2): Error BP5001: This assertion might not hold.
2  Execution trace:
3      code.bpl(278,8): anon0
4
5  Boogie program verifier finished with 0 verified, 1 error
```

4.5 Translation to JavaScript

SafeScript compiles to JavaScript, so that JavaScript programs can be used together with SafeScript ones. The SafeScript syntax and semantics, namely its expressions and statements, was heavily based on JavaScript, so the translation is almost direct.

In JavaScript types are not declared, so SafeScript type declarations are forgotten in the translation. However, types in SafeScript can influence control flow via the *in* predicate (`x in T`), so we have to translate types to JavaScript expressions. Figure 4.23 presents the translation, where $\mathbf{Js}[\![e \text{ in } T]\!]$ is defined as $\mathbf{Js}[\![T]\!](\mathbf{Js}[\![e]\!])$.

$$
\begin{aligned}
\mathbf{Js}[\![\mathsf{any}]\!](e) &= \mathsf{true} \\
\mathbf{Js}[\![\mathsf{int}]\!](e) &= \mathsf{typeof}(e) === \text{"}\mathsf{number}\text{"} \\
\mathbf{Js}[\![\mathsf{boolean}]\!](e) &= \mathsf{typeof}(e) === \text{"}\mathsf{boolean}\text{"} \\
\mathbf{Js}[\![\mathsf{string}]\!](e) &= \mathsf{typeof}(e) === \text{"}\mathsf{string}\text{"} \\
\mathbf{Js}[\![T\,[]]\!](e) &= (a \Rightarrow \mathsf{Array.isArray}(a) \wedge a.\mathsf{every}(x \Rightarrow \mathbf{Js}[\![T]\!](x)))(e) \\
\mathbf{Js}[\![\{\}]\!](e) &= (o \Rightarrow o \text{ instanceof Object} \wedge \text{!}\mathsf{Array.isArray}(o))(e) \\
\mathbf{Js}[\![\{l\colon T\}]\!](e) &= (o \Rightarrow o \text{ instanceof Object} \wedge \text{!}\mathsf{Array.isArray}(o) \\
&\qquad \wedge o.\mathsf{hasOwnProperty}(l) \wedge \mathbf{Js}[\![T]\!](o.l))(e) \\
\mathbf{Js}[\![(x\colon T \text{ where } e_1)]\!](e) &= (x \Rightarrow \mathbf{Js}[\![T]\!](x) \wedge \mathbf{Js}[\![e_1]\!])(e)
\end{aligned}
$$

where a and o are fresh variables

Figure 4.23: Translation of SafeScript *in type* predicate to JavaScript: $\mathbf{Js}[\![T]\!](e)$

Every SafeScript scalar type has a direct correspondence with a JavaScript primitive type. JavaScript does not distinguish between integer and floating point numbers. All primitive integers are stored using IEEE 754 double precision floating point format, and

belong to a single type number. Arrays are represented by objects in JavaScript. The JavaScript primitive function Array.isArray is used to distinguished them.

There is substantial difference between objects and arrays: SafeScript does not use references. Therefore, when making an assignment of an object, a new object is created. This happens also with function arguments: they are passed as object values and not as references. To simulate this behavior in JavaScript, whenever a value is assigned to a variable or used in a function called, the value is deeply copied.

Functions and global variables of a SafeScript program are exported, so they can be accessed in other JavaScript files. For instance, if the examples of section 4.3 are contained in a file named `utils.ss`, then these functions can be accessed as follows (using node.js):

```
1 var utils = require("utils.js"); // imports functions
2
3 console.log(utils.sum([1, 2, 3])); // uses array sum function
```

4.5.1 Operational semantics

To precisely describe the semantics of SafeScript language an operational semantics was defined. The semantics essentially follows JavaScript, more precisely the ECMAScript Specification [69]. The major differences are related to the absence of object references, reflected in the assignment rules.

Figure 4.24 presents the additional syntax necessary to describe the operational semantics rules.

Value	v	$::=$	$c \mid \{l_1 = v_1, \ldots, l_n = v_n\} \mid [v_1, \ldots, v_n]$
Location	w	$::=$	$x \mid w.l \mid w[v]$
Store	ρ	$::=$	$\epsilon \mid \rho, x = v$
Store stack	π	$::=$	$\epsilon \mid \rho : \pi$
Local and global stores	μ	$::=$	$\pi; \rho$
Expression	e	$::=$	$\ldots \mid \{\, U_1\ y_1 = e_1\, ;\, \ldots\, ;\, U_m\ y_m = e_m\, ;\, S\,\}$

Figure 4.24: Additional syntax for evaluation (extends figure 4.1)

SafeScript values comprise scalar constants, and objects and arrays whose elements are themselves values. The value of each variable is placed in a store ρ, a map from variables to values. At a given time during the evaluation of a program there may exist multiple stores: the store on global variables ρ, and the various stores for local variables $\pi = \rho_1 : \ldots : \rho_n$, where ρ_1 is the current function store and $\rho_2 : \ldots : \rho_n$ are the

suspended stores of the functions on the stack. We introduce a new expression, the body of a function, representing a function call under evaluation.

The operational semantics is defined using the one step relation evaluation, as defined by B. Pierce [58]. Figure 4.25 summarizes the judgments of the evaluation system.

$e \mid \mu \longrightarrow e' \mid \mu$	$\equiv$	in store μ, expression e reduces to e' and yields μ'	Figure 4.26
v in $T \longrightarrow e$	$\equiv$	expression v in T reduces to e	Figure 4.27
$S \mid \mu \longrightarrow S' \mid \mu'$	$\equiv$	in store μ, statement S reduces to S' and yields μ'	Figure 4.28
$u \mid \mu \hookrightarrow u' \mid \mu'$	$\equiv$	in store μ, LHS u reduces, without variable reduction, to u' and yields μ'	Figure 4.29
$[w \mapsto v]\mu \Downarrow \mu'$	$\equiv$	in store μ, updating w with value v results in store μ'	Figure 4.30

Figure 4.25: Judgments of the evaluation system

All reduction relations but store update use small-step semantics. The first judgement describe how expressions reduce, and the rules are presented in figure 4.26. In each expression, the sub-expressions are evaluated from left to right, and only when all reach a value the operation is applied. In the variable expression, only the top store is accessible, from the current procedure (E-LocVar), or the store ρ with the global variables (E-GlobVar), in this order. There are no reduction rules for quantifiers expressions since they are only allowed in types not in the scope of an *in* predicate.

In the case of function calls, the call is replaced by the function body (a newly introduced expression), and a new store is added to the stack (E-FCall). The body is then reduced, first the local variable declarations (E-Decl), and later its statements (E-Stat). When reduction reaches a return statement, whose expression is a value, the statement reduces to the value, and the stack is popped (E-StatRet). Note that, if the function terminates, the evaluation of the body must reach a *return* statement since rule E-FCall adds a *return* statement to the end of the function body, (cf. the function validation rule, figure 4.12).

The reduction of the *in* predicate is done separately, since it is necessary to consider each primitive type. Figure 4.27 introduces the evaluation rules for the *in type* predicate. The *in type* expression does not reads or writes the store, so the judgement not includes it. Note that the translation to JavaScript (in figure 4.23) matches these reduction rules.

The reduction rules for expressions are used by the statements evaluation rules, which are presented in figure 4.28. The rules for statements rules are intuitive, except for the assignment. In JavaScript assignments are evaluated from left to right: first the left hand side of the assignment is reduced to a memory location, then the right hand side is reduced to a value, and finally the memory location is updated with the value. Figure 4.29 shows according to the syntax of SafeScript expressions the evaluation of left-hand side terms to

$$\frac{(x = v) \in \rho}{x \mid (\rho : \pi; \rho_g) \longrightarrow v \mid (\rho : \pi; \rho_g)} \text{ (E-LocVar)} \quad \frac{(x = \_) \notin \rho \quad (x = v) \in \rho_g}{x \mid (\rho : \pi : \rho_g) \longrightarrow v \mid (\rho : \pi : \rho_g)} \text{ (E-GlobVar)}$$

$$\frac{e_i \mid \mu \longrightarrow e_i' \mid \mu'}{\oplus(v_1, \ldots, v_{i-1}, e_i, \ldots, e_n) \mid \mu \longrightarrow \oplus(v_1, \ldots, v_{i-1}, e_i', \ldots, e_n) \mid \mu'} \text{ (E-OpArg)}$$

$$\frac{\oplus(v_1, \ldots, v_n) \mapsto e \quad \text{is defined}}{\oplus(v_1, \ldots, v_n) \mid \mu \longrightarrow e \mid \mu} \text{ (E-Op)} \quad \frac{e_1 \mid \mu \longrightarrow e_1' \mid \mu'}{e_1 \,?\, e_2 : e_3 \mid \mu \longrightarrow e_1' \,?\, e_2 : e_3 \mid \mu} \text{ (E-Ternary)}$$

$$\frac{}{\mathsf{true} \,?\, e_2 : e_3 \mid \mu \longrightarrow e_2 \mid \mu} \text{ (E-TerTrue)} \quad \frac{}{\mathsf{false} \,?\, e_2 : e_3 \mid \mu \longrightarrow e_3 \mid \mu} \text{ (E-TerFalse)}$$

$$\frac{e \mid \mu \longrightarrow e' \mid \mu'}{e \text{ in } T \mid \mu \longrightarrow e' \text{ in } T \mid \mu'} \text{ (E-InTypeArg)} \quad \frac{v \text{ in } T \longrightarrow e}{v \text{ in } T \mid \mu \longrightarrow e \mid \mu} \text{ (E-InType)}$$

$$\frac{e_i \mid \mu \longrightarrow e_i' \mid \mu'}{\{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = e_i, \ldots, l_n = e_n\} \mid \mu \longrightarrow \{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = e_i', \ldots, l_n = e_n\} \mid \mu'} \text{ (E-ObjExp)}$$

$$\frac{e \mid \mu \longrightarrow e' \mid \mu'}{e.l \mid \mu \longrightarrow e'.l \mid \mu'} \text{ (E-ObjAccArg)}$$

$$\frac{}{\{l_1 = v_1, \ldots, l_i = v_i, \ldots, l_n = v_n\}.l_i \mid \mu \longrightarrow v_i \mid \mu} \text{ (E-ObjAcc)}$$

$$\frac{e_i \mid \mu \longrightarrow e_i' \mid \mu'}{[v_1, \ldots, v_{i-1}, e_i, \ldots, e_n] \mid \mu \longrightarrow [v_1, \ldots, v_{i-1}, e_i', \ldots, e_n] \mid \mu'} \text{ (E-ArrExp)}$$

$$\frac{e_1 \mid \mu \longrightarrow e_1' \mid \mu'}{e_1[e_2] \mid \mu \longrightarrow e_1'[e_2] \mid \mu'} \text{ (E-ArrAccArg1)} \quad \frac{e_2 \mid \mu \longrightarrow e_2' \mid \mu'}{v_1[e_2] \mid \mu \longrightarrow v_1[e_2'] \mid \mu'} \text{ (E-ArrAccArg2)}$$

$$\frac{}{[v_1, \ldots, v_{n+1}, \ldots, v_m][n] \mid \mu \longrightarrow v_{n+1} \mid \mu} \text{ (E-ArrAcc)}$$

$$\frac{e_i \mid \mu \longrightarrow e_i' \mid \mu'}{f(v_1, \ldots, v_{i-1}, e_i, \ldots, e_n) \mid \mu \longrightarrow f(v_1, \ldots, v_{i-1}, e_i', \ldots, e_n) \mid \mu'} \text{ (E-FCallArg)}$$

Figure 4.26: Expression evaluation: $e \mid \mu \longrightarrow e' \mid \mu'$ *(part 1)*

location, hence the expression evaluation rules (figure 4.26) cannot be used.

After the left hand side is reduced via rule E-AssLeft to a location w, and the expression is reduced to a value via rule E-AssRight, the store is updated via rule E-Assign. The rules for store update are presented in figure 4.30, using a big step semantics. The rule E-StrObjAcc for object access of form $w.l$ first evaluates w as an expression to a value. The relation $e \mid \mu \longrightarrow^* v \mid \mu'$ denotes the reflexive and transitive closure of the $e \mid \mu \longrightarrow e' \mid \mu'$ relation.

$$\frac{T\ f\ (\ T_1\,x_1, \ldots, T_n\,x_n\ )\ \{\ U_1\ y_1 = e_1\,;\ \ldots\ U_m\ y_m = e_m\,;\ S\ \}\quad \text{is defined}}{\begin{array}{c} f(v_1, \ldots, v_n) \mid (\pi; \rho_g) \longrightarrow \{\ U_1\ y_1 = e_1\,;\ \ldots\ ;\ U_m\ y_m = e_m\,;\ S\,;\ \mathsf{return}\ \} \\ \mid ((x_1 = v_1, \ldots, x_n = v_n) : \pi; \rho_g) \end{array}}\ \text{(E-FCall)}$$

$$\frac{e_1 \mid \mu \longrightarrow e_1' \mid \mu'}{\begin{array}{c} \{\ U_1\ y_1 = e_1\,;\ \ldots\ ;\ U_m\ y_m = e_m\,;\ S\ \} \mid \mu \longrightarrow \\ \{\ U_1\ y_1 = e_1'\,;\ \ldots\ ;\ U_m\ y_m = e_m\,;\ S\ \} \mid \mu' \end{array}}\ \text{(E-DeclExp)}$$

$$\frac{}{\begin{array}{c} \{\ U_1\ y_1 = v_1\,; U_2\ y_2 = e_2\,;\ \ldots\ ;\ U_m\ y_m = e_m\,;\ S\ \} \mid \rho : \pi : \rho_g \longrightarrow \\ \{\ U_2\ y_2 = e_2\,;\ \ldots\ ;\ U_m\ y_m = e_m\,;\ S\ \} \mid \rho, y_1 = v_1 : \pi : \rho_g \end{array}}\ \text{(E-Decl)}$$

$$\frac{S \mid \mu \longrightarrow S' \mid \mu'}{\{\ S\ \} \mid \mu \longrightarrow \{\ S'\ \} \mid \mu'}\ \text{(E-Stat)} \qquad \frac{}{\{\ \mathsf{return}\ v\ \} \mid \rho : \pi : \rho_g \longrightarrow v \mid \pi : \rho_g}\ \text{(E-StatRet)}$$

Figure 4.26: Expression evaluation: $e \mid \mu \longrightarrow e' \mid \mu'$ *(part 2)*

$$\begin{aligned}
v\ \mathsf{in\ any} &\longrightarrow \mathsf{true} \\
v\ \mathsf{in}\ G &\longrightarrow \mathsf{true} \quad \text{if } v \in K(G) \\
\{l_1 = v_1, \ldots, l_n = v_n\}\ \mathsf{in}\ \{\} &\longrightarrow \mathsf{true} \\
\{l_1 = v_1, \ldots, l_i = v_i, \ldots, l_n = v_n\}\ \mathsf{in}\ \{l_i\colon T_i\} &\longrightarrow v_i\ \mathsf{in}\ T_i \\
[v_1, \ldots, v_n]\ \mathsf{in}\ T\ [] &\longrightarrow v_1\ \mathsf{in}\ T\ \&\ \ldots\ \&\ v_n\ \mathsf{in}\ T \\
v\ \mathsf{in}\ (x\colon T\ \mathsf{where}\ e) &\longrightarrow v\ \mathsf{in}\ T\ \&\ [v/x]e \\
\mathsf{undefined\ in\ void} &\longrightarrow \mathsf{true} \\
\text{otherwise} \quad v\ \mathsf{in}\ T &\longrightarrow \mathsf{false}
\end{aligned}$$

where $K(G)$ denotes the set of all values of scalar type G

Figure 4.27: *In type* predicate evaluation: $v\ \mathsf{in}\ T \longrightarrow e$

$$\frac{u \mid \mu \hookrightarrow u' \mid \mu'}{u = e \mid \mu \longrightarrow u' = e \mid \mu'} \text{ (E-AssLeft)} \quad \frac{e \mid \mu \longrightarrow e' \mid \mu'}{w = e \mid \mu \longrightarrow w = e' \mid \mu'} \text{ (E-AssRight)}$$

$$\frac{[w \mapsto v]\mu \Downarrow \mu'}{w = v \mid \mu \longrightarrow \epsilon \mid \mu'} \text{ (E-Assign)}$$

$$\frac{e \mid \mu \longrightarrow e' \mid \mu'}{\mathsf{if}\ (\,e\,)\ S_1\ \mathsf{else}\ S_2 \mid \mu \longrightarrow \mathsf{if}\ (\,e'\,)\ S_1\ \mathsf{else}\ S_2 \mid \mu} \text{ (E-IfCond)}$$

$$\frac{}{\mathsf{if}\ (\mathsf{true})\ S_1\ \mathsf{else}\ S_2 \mid \mu \longrightarrow S_1 \mid \mu} \text{ (E-IfTrue)}$$

$$\frac{}{\mathsf{if}\ (\mathsf{false})\ S_1\ \mathsf{else}\ S_2 \mid \mu \longrightarrow S_2 \mid \mu} \text{ (E-IfFalse)}$$

$$\frac{}{\mathsf{while}\ (\,e_1\,)\ \mathsf{inv}\ e_2\ S \mid \mu \longrightarrow \mathsf{if}\ (\,e_1\,)\ \{\,S\,;\ \mathsf{while}\ (\,e_1\,)\ \mathsf{inv}\ e_2\ S\,\}\ \mathsf{else}\ \epsilon \mid \mu} \text{ (E-While)}$$

$$\frac{S_1 \mid \mu \longrightarrow S_1' \mid \mu'}{S_1; S_2 \mid \mu \longrightarrow S_1'; S_2 \mid \mu'} \text{ (E-SeqFst)} \quad \frac{}{\epsilon; S_2 \mid \mu \longrightarrow S_2 \mid \mu} \text{ (E-SeqSnd)}$$

$$\frac{}{\mathsf{return}\ e; S_2 \mid \mu \longrightarrow \mathsf{return}\ e \mid \mu} \text{ (E-SeqRet)} \quad \frac{e \mid \mu \longrightarrow e' \mid \mu'}{\mathsf{return}\ e \mid \mu \longrightarrow \mathsf{return}\ e' \mid \mu'} \text{ (E-Return)}$$

Figure 4.28: Statement evaluation: $S \mid \mu \longrightarrow S' \mid \mu'$

$$\frac{u \mid \mu \hookrightarrow u' \mid \mu'}{u.l \mid \mu \hookrightarrow u'.l \mid \mu'} \text{ (E-ObjAccUpd)}$$

$$\frac{u \mid \mu \hookrightarrow u' \mid \mu'}{u[e] \mid \mu \hookrightarrow u'[e] \mid \mu'} \text{ (E-ArrAccUpd1)} \quad \frac{e \mid \mu \hookrightarrow e' \mid \mu'}{w[e] \mid \mu \hookrightarrow w[e'] \mid \mu'} \text{ (E-ArrAccUpd2)}$$

Figure 4.29: Left hand side evaluation: $u \mid \mu \hookrightarrow u' \mid \mu'$

$$\frac{}{[x_i \mapsto v]((x_1 = v_1, \ldots, x_i = v_i, \ldots, x_n = v_n) : \pi; \rho_g) \Downarrow ((x_1 = v_1, \ldots, x_i = v, \ldots, x_n = v_n) : \pi; \rho_g)} \text{(E-StrLocVar)}$$

$$\frac{(x = \_) \notin \rho}{[x_i \mapsto v](\rho : \pi; (x_1 = v_1, \ldots, x_i = v_i, \ldots, x_n = v_n)) \Downarrow (\rho : \pi; (x_1 = v_1, \ldots, x_i = v, \ldots, x_n = v_n))} \text{(E-StrGlobVar)}$$

$$\frac{w \mid \mu \longrightarrow^* \{l_1 = v_1, \ldots, l_i = v_i, \ldots, l_n = v_n\} \mid \mu \quad [w \mapsto \{l_1 = v_1, \ldots, l_i = v, \ldots, l_n = v_n\}]\mu \Downarrow \mu'}{[w.l_i \mapsto v]\mu \Downarrow \mu'} \text{(E-StrObjAcc)}$$

$$\frac{w \mid \mu \longrightarrow^* [v_1, \ldots, v_{i-1}, v_i, v_{i+1} \ldots, v_n] \mid \mu \quad [w \mapsto [v_1, \ldots, v_{i-1}, v, v_{i+1}, \ldots, v_n]]\mu \Downarrow \mu'}{[w[i] \mapsto v]\mu \Downarrow \mu'} \text{(E-StrArrAcc)}$$

Figure 4.30: Store update: $[w \mapsto v]\mu \Downarrow \mu'$

4.6 Implementation

The SafeScript compiler is implemented in Java using Xtext [6, 17], Xtext is a framework to develop programming languages and DSLs. It uses ANTLR 3 (ANother Tool for Language Recognition) [53] to create a lexer and a parser from the SafeScript grammar, and links the generated AST (Abstract Syntax Tree) to Eclipse editing support (or other editor that supports the Language Server Protocol [48]).

Figure 4.31 summarizes the compilation process. First, SafeScript programs are parsed using Xtext, thus generating an AST. Then, during the validation phase, the AST is translated to Boogie code in order to perform typing validation according with the validation rules presented in section 4.4. The Boogie verifier is used to check the Boogie code. Assertion that cannot be proved result in a typing error and the compilation ends. If all assertions are proved (i.e., the program has no typing error), then the AST is translated to JavaScript, according with the description and rules of section 4.5.

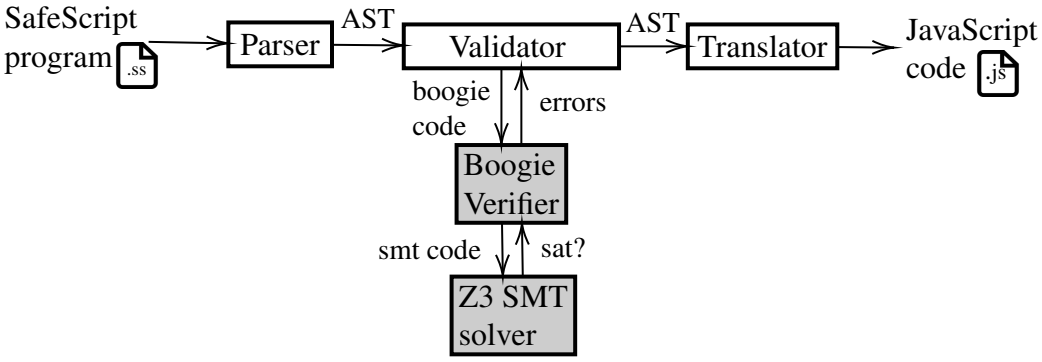

Figure 4.31: SafeScript compilation time works flow

The SafeScript editor and compiler is available as a Eclipse plugin in [75]. This allows to create SafeScript projects, which compiles SafeScript source code and, in the absence of errors, automatically generates the correspond JavaScript program. Errors are identified in the Eclipse IDE at the point of the program where they occured. For example, the invalid array access example in section 4.3 is shown in figure 4.32.

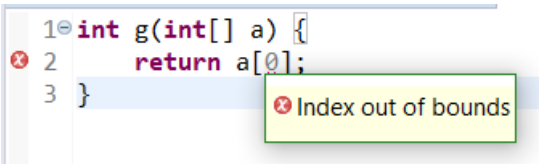

Figure 4.32: Eclipse IDE error of an invalid SafeScript program

Chapter 5

SafeRESTScript Language

This chapter presents the SafeRESTScript programming language, an extension of SafeScript with support for REST calls that are statically validated. This is the second and main goal of this work. The language validator uses HeadREST specifications, presented in chapter 3, for helping with the validation of programs with REST calls.

5.1 Main Idea

In SafeScript, the verification of a function call consists in verifying whether arguments match the parameters type. Furthermore, the function type may reference the parameters.

A REST endpoint can be seen as function that receives an HTTP request, possibly changes a global resource set state, and then returns an HTTP response. Therefore, a REST call can be seen as a call to an external function. Note that such functions should accept any argument, even those that do not conform to expectations.

A HeadREST specification of a REST API describes the relation between request and response of each API endpoint. Looking at REST endpoints as functions, each HeadREST triple specifies the relation of the input function parameters (request) against the function output (response), as well as the changes in the resource set inflicted by the execution of the function.

JavaScript is single threaded and its code should not block, since this would freeze the browser (or the tab) were the JavaScript runs. Therefore, function calls that take time to handle or execute should be executed asynchronously. Towards this, the runtime maintains a queue of asynchronous calls. REST calls fall into this category, so they should ideally be executed asynchronously. SafeRESTScript provides for asynchronous and synchronous REST calls.

5.2 Additional Syntax

SafeRESTScript extends SafeScript with new syntax to describe REST calls. The additional syntax is presented in figure 5.1.

Expression	e	::=	$\ldots$ | await $m\ s\ e$ | synch $m\ s\ e$ | await $f(e_1, \ldots, e_n)$
REST Method	m	::=	get | post | put | delete
Declaration	D	::=	$\ldots$ | specification s of s | async F

Figure 5.1: SafeRESTScript additional syntax (extends figure 4.1)

A new declaration is introduced to import HeadREST specifications. The first string literal must have the path to the HeadREST file, and the second literal must contain the absolute base path of the REST service, which is appended to the relative URL (taken from the REST calls) to form the absolute URL.

The REST call expression is composed of an HTTP method m (**`get`**, **`post`**, **`put`** or **`delete`**), a string literal with the relative URL of the target resource, and an expression that should evaluate to a request object. The URL string literal must be equal to at least one of the URI template in the HeadREST specification imported by the program. The request object expression must be of the type **`Request`**, whereas the returning value from the call is of the type **`Response`**. These two types definitions are those of HeadREST, and are defined in figure 3.15.

Recently, JavaScript introduced the operations **`async`** and **`await`** for defining asynchronous calls using promises, rather than the usual callbacks [69]. This new syntax allows creating simpler asynchronous calls with a syntax similar to that synchronous calls. The same idea was used in SafeRESTScript. Asynchronous REST calls must be preceded by the keyword **`await`**. Functions to be treated asynchronously must include **`async`** in the header, as presented in the new syntax (figure 4.1). Functions that call other asynchronous function must also be annotated with **`async`**.

5.3 A Simple Example

To illustrate the use of SafeRESTScript we program a consumer of the Dummy API introduced in section 2.1. A simplified version of the HeadREST specification of employee creation is presented below. In this version, only the success case is specified and only the request and response formats are described.

```
1 specification DummyAPI
2
3 type EmployeeRequest = {name: String, salary: String, age: String}
```

```
4  type EmployeeResponse = EmployeeRequest & {id: String}
5
6  {
7      request in {body: EmployeeRequest}
8  }
9      post '/create'
10 {
11     response.code == 200 &&
12     response in {body: EmployeeResponse}
13 }
```

Below is a simple SafeRESTScript program with a function that makes an async call to the Dummy API for adding a new employee.

```
1  specification "dummyAPI.hrest"
2          of "http://dummy.restapiexample.com/api/v1"
3
4  async string addEmployee(string empName, string empSalary,
5              string empAge) {
6      EmployeeRequest requestBody = {name = empName,
7              salary = empSalary, age = empAge};
8      Response response = await post "/create" {body = requestBody};
9      return response.body.id;
10 }
```

First the program imports the specification from the `dummyAPI.hrest` file and indicates the absolute base path for the Dummy API. The function receives the three parameters needed for employee creation: name, salary and age. Line 6 creates the request body using the function arguments. Note that the type of the variable `requestBody` is also imported from the specification. Then, in line 8, it is made an asynchronous call to the employee creation and finally, in line 9, it is returned the id that was received in the response (created by the service).

In this function there are two important validations concerning to the REST call. First we must check that the endpoint is valid, i.e., that in the API specification exists at least one triple with `"/create"` URL and the **`post`** method. Next we need to check whether the object `response.body` has field `id`. This can be concluded from the triple: since the request body belongs to type `EmployeeRequest`, then the body is in the type `EmployeeResponse`, and therefore the field `id` exists.

A final note about this example: since keyword **`await`** prefixes the REST call (making the call asynchronous), the function itself is also asynchronous and must be annotated with the **`async`** keyword in its signature.

5.4 Validation

Several additional validations are necessary to support the new syntax. The first is related with specifications imported. The given path must be a path to a HeadREST file, and the specification is validated using the HeadREST validator described in chapter 3.

The REST calls are encoded in Boogie using a simple function, restCall, as presented at the end of the axiomatization (appendix C). The function receives as parameters the REST method, a string representing the URI template relative path, and the request object, and returns the response object. The interpretation of this function is given axioms (one per triple of the specification) according to figure 5.2. The axiom relates the function return value, the REST call response defined in the post-condition, with the function request call argument that is defined in the pre-condition.

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

where u' is the string that represents u

Figure 5.2: Translation of HeadREST specification: $\mathbf{B}[\![S]\!]$

Since SafeScript does not support resources and their operator (including **uriof** and **repof** predicates) *at the line of this writing*, we assume HeadREST specifications do not include resources. It may be necessary to create a separate specification that only specifies the format of the request and response.

Figure 5.3 presents the translation of REST calls expressions with validation.

$$\begin{aligned}\mathbf{V}^*[\![\mathsf{synch}\ m\ s\ e]\!](x) = \mathbf{V}^*[\![e]\!](y)\ &\textbf{assert}\ \mathbf{F}'[\![y]\!](\mathsf{Request});\ x := \mathsf{restCall}(m, \mathbf{V}[\![s]\!], y);\\ &\textbf{assume}\ \mathbf{F}'[\![x]\!](\mathsf{Response});\\ \mathbf{V}^*[\![\mathsf{await}\ m\ s\ e]\!](x) = \mathbf{V}^*[\![\mathsf{synch}\ m\ s\ e]\!](x)\ &\textbf{havoc}\ g_1, \ldots, g_n;\end{aligned}$$

where y is a fresh variable, and $g_1, \ldots, g_n$ are the global variables.

Figure 5.3: Translation of expressions with type validation: $\mathbf{V}^*[\![e]\!](x)$ (extends figure 4.16)

The request object must be of the type **Request**, as defined in HeadREST. On the other hand, the response can be assumed to be of the type **Response**, which is also the case in HeadREST. The effective type of the REST call response is defined by the axioms of the

restCall function, and therefore by the post-condition of HeadREST triples that match the pair (HTTP method, URI template) of the REST call, for which the pre-condition with the given the request object holds.

In asynchronous REST calls the execution of the function is suspended, and when it is resumed, the global variables may have changed. The primitive Boogie statement havoc is used over all global variables to assign an arbitrary value to those values, that respect their declared type presented in there *where* clause (see global variable Boogie translation in section 4.5).

Additionally, it is also checked whether there exists at least one triple in the imported specification for the pair (REST method, URI template) of each REST call. The specification may not cover all possible scenarios, i.e., the disjunction of the pre-conditions for a given method and URI template may not be true, so that calls may not have triples that apply. In that case, the only aspect known about the response is that it belongs to the generic `Response` type. Triples in the HeadREST specification may also be inconsistent: two or more triples that have a non false pre-condition intersection may have a false post-condition intersection. In that case, the axioms generated will also be inconsistent and Boogie will validate all assertions, including false ones. Therefore, this sort of REST calls validation only makes sense with consistent specifications.

5.5 Translation to JavaScript

The translation of REST calls is achieved by calling auxiliary functions, one for synchronous and another by asynchronous calls. These functions are added to the generated JavaScript code and are detailed in appendix D.

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 [31], only for the level of URI templates supported by the HeadREST language. We add the content-type JSON 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.

Many REST APIs endpoints can only be used successfully with authentication, that is including a special token in the header authorization. The simpler and most common type of authentication is the Basic authentication scheme, defined by RFC 7617 [60]. This authentication is not secure and must be used with HTTPS. The token must be encoded in base64 [40], which is done by the native JavaScript function *btoa*. To simply this process, an additional header *basicAuthorization* is supported, that, before the call, performs the necessary encoding and adds the necessary authorization header.

The `await` and `async` keywords are directly translated to JavaScript. Functions that

are asynchronous return a promise with the returning value, and the **await** operator reads that value. An example of a JavaScript program is presented below. This program calls the asynchronous function presented in section 5.3 and programmed in SafeRESTScript, made available in JavaScript by the SafeRESTScript compiler.

```
1 var dummyClient = require("dummyClient.js"); // imports function
2
3 // prints the employees id after the response is received in a
4 // non deterministic order
5 dummyClient.addEmployee("Robin", "750", "24").then(console.log);
6 dummyClient.addEmployee("Francisco", "750", "24").then(console.log);
7
8 // this log is printed first
9 console.log(42);
```

5.6 Implementation

SafeRESTScript is implemented as an extension of SafeScript, so SafeRESTScript compiler its implemented with the same language and tools. Figure 5.4 presents a diagram that summarizes the compilation of a SafeRESTScript program.

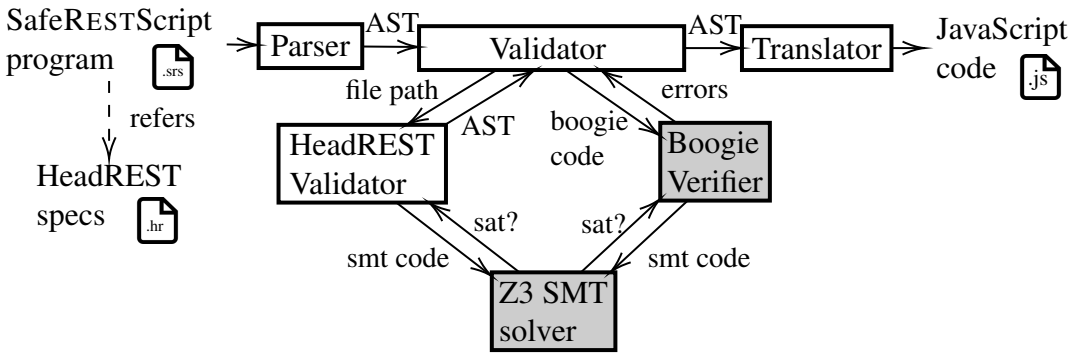

Figure 5.4: SafeRESTScript compilation time works flow

The main difference with SafeScript (cf. figure 4.31) is in the validation phase. A SafeRESTScript program can reference a HeadREST specification, which is validated using the HeadREST validator (described in chapter 3), using the same SMT solver for the semantic evaluation. If the validation fails, an error is reported, else the HeadREST compiler retrieves an AST that represents the specification. Each REST call in the AST is validated by translating each specification triple to Boogie as described in section 5.4. The JavaScript translation is similar since the SafeRESTScript specification is only used for the validation phase.

SafeRESTScript compiler and editor is available as a SafeScript extension in [75]. The plugin compiles SafeRESTScript files, and is also equipped with the HeadREST validator

and editor, so that specifications can be edited as well. Figure 5.5 presents the structure of a SafeRESTScript project, that uses the DummyAPI example. In the project root we find SafeRESTScript programs. The generated JavaScript files are generated and placed in the *src-gen* folder. In this case HeadREST specifications were placed in folder *specs*, but SafeRESTScript programs can import specifications from arbitrary locations (inside or outside the project).

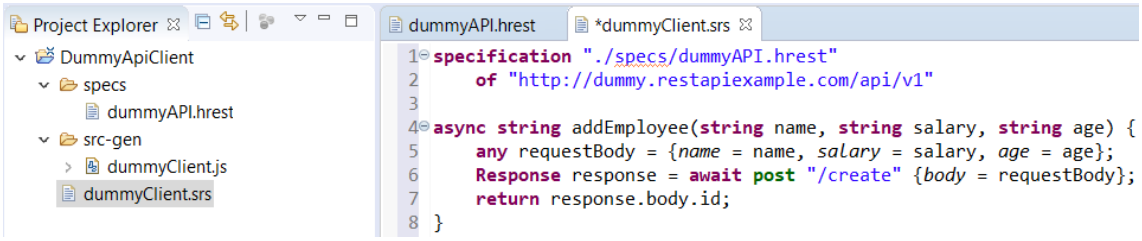

Figure 5.5: SafeRESTScript project structure

As shown in SafeScript implementation section, errors related to REST calls are also highlighted by the Eclipse IDE. Figure 5.6 shows an error detected by the SafeRESTScript validator: the post-condition of the triple for **post** does not ensure that the object body contains the address field (so it cannot be accessed in the response body).

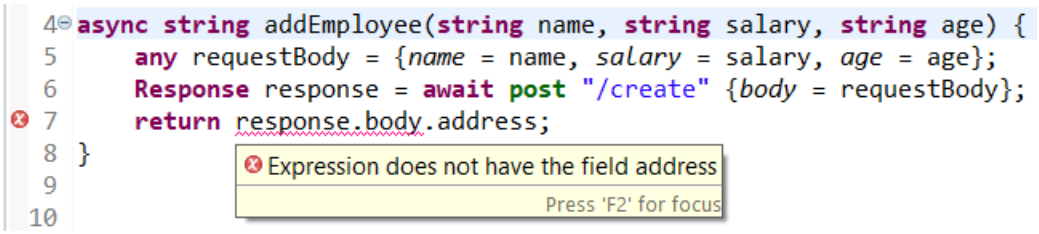

Figure 5.6: Eclipse IDE error report in an invalid SafeRESTScript program

IDEs help programmers in writing code or fixing errors, and Eclipse is no exception. A few features were added to the SafeRESTScript editor, including one that assists the programmer in writing REST calls against a specification. Figure 5.7 presents an example: when the programmer introduces the method of the REST call (**get**), the IDE shows the possible URI templates available for the method, according to the imported specification.

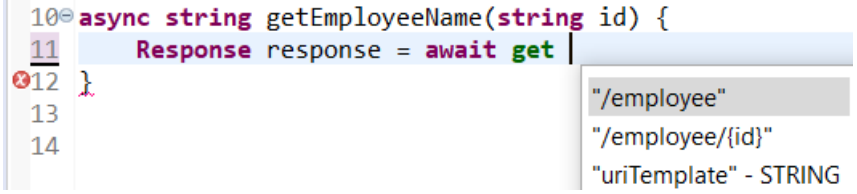

Figure 5.7: Eclipse IDE content assist in an REST call

A common mistake when writing code that consumes REST services is to forget to declare the asynchronous calls inside asynchronous functions. Figure 5.8 shows two *quick fixes* for this kind of error: add the **async** keyword the function signature, or change the REST call to synchronous.

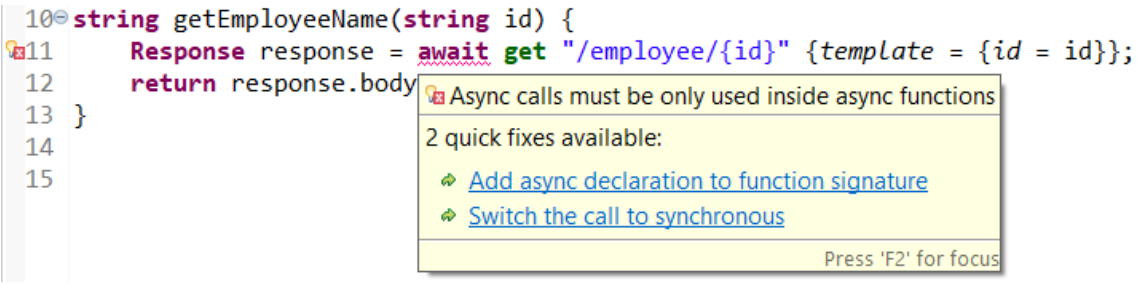

Figure 5.8: Eclipse IDE quick fix example

Chapter 6

Evaluation

This chapter presents some experiments that were carried out in order to evaluate SafeScript and its extension SafeRESTScript. These experiments have several goals: (i) show the ability of SafeScript validator and comparing the performance with other verification tool, (ii) compare with a similar propose language (TypeScript) the code that is necessary to achieve some functionally or error, and (iii) present a set of practical and sophisticated examples which include complex REST calls that could be found in real examples.

In this chapter it is also discussed the languages limitations and, from these, delineated the future work.

6.1 Verification Benchmarks Challenge

In 2008, Weide et al. published a suite with eight incremental benchmarks for verification tools and techniques that prove total correction of object-based and object-oriented software [79], to answer Hoare et al. Verified Software Initiative [36]. Some authors of verification tools (e.g. Dafny [44]) accepted the challenge and proposed solutions for some of this benchmarks. This section presents solutions for two of these benchmarks using SafeScript.

Benchmark #1: Adding and Multiplying Numbers *Verify an operation that adds two numbers by repeated incrementing. Verify an operation that multiplies two numbers by repeated addition, using the first operation to do the addition. Make one algorithm iterative, the other recursive.*

An iterative version of the addition and recursive version of the multiplication were developed. Below it is presented the addition function.

```
1 (x: int where x == a + b) sum(int a, int b) {
2     int r = a;
3     int l = b;
4     if (b > 0) {
5         while (l != 0)
```

```
6           inv l >= 0 && r == a + b - l {
7               r = r + 1;
8               l = l - 1;
9           }
10      }
11      else {
12          while (l != 0)
13          inv l <= 0 && r == a + b - l {
14              r = r - 1;
15              l = l + 1;
16          }
17      }
18      return r;
19  }
```

This solution is very similar to the one developed in Dafny. While Dafny asserts the addition correction using an ensures clause, SafeScript asserts the same using a refinement type for the function return type. Thanks also to refinement types, as show below, it is possible to program the function with a single loop.

```
1  (x: int where x == a + b) sum(int a, int b) {
2      int r = a;
3      (x: int where sameSign(b, x)) l = b;
4      int d = b > 0 ? 1 : -1;
5      while (l != 0) inv r == a + b - l {
6          r = r + d;
7          l = l - d;
8      }
9      return r;
10 }
11
12 // auxiliary specification function
13 function sameSign(int x, int y) {
14     x > 0 => y >= 0 & x < 0 => y <= 0
15 }
```

The main difference between both loops is the invariant which asserts the sign of `l`, that, during the loop execution, must have the same signal as `b` (including zero). This information can be establish in the variable declaration, refinement the integer that `l` can contain, which depends on `b`'s value. Note the use of a specification function which helps to assert this dependency. As explained in chapter 4, the variable type declaration must hold during all execution points of the function, including all loops iterations, so it works as a loop invariant as well, and therefore it helps, in this case, to prove the return type condition.

The recursive version of the multiplication function is presented next. It uses the addition function defined before.

```
(x: int where x == a * b) multiply(int a, int b) {
    if (a == 0) {
        return 0;
    }
    if (a > 0) {
        return sum(multiply(a - 1, b), b);
    }
    else {
        return -multiply(-a, b);
    }
}
```

Benchmark #2: Binary Search in an Array *Verify an operation that uses binary search to find a given entry in an array of entries that are in sorted order.*

The developed solution has a binary search over an array of integers, sorted by their natural order, and in an iterative way. Below, it is presented the algorithm.

```
type IntArrOrd = (a: int[] where forall i: int . forall j: int .
        i >= 0 && j < length(a) && i < j ==> a[i] <= a[j]);

(i: int where bs(a, k, i)) binarySearch(IntArrOrd a, int k) {
    int mid = 0;
    int low = 0;
    int up = length(a) - 1;
    while (low <= up)
        inv 0 <= low && up < length(a)
        inv (forall x: int . up < x && x < length(a) ==> a[x] > k)
        inv (forall x: int . 0 <= x && x < low ==> a[x] < k)
    {
        mid = (low + up) / 2;
        if (a[mid] == k) {
            return mid;
        }
        if (a[mid] > k) {
            up = mid - 1;
        } else {
            low = mid + 1;
        }
    }
    return -1;
}

// auxillary specification function
function bs(IntArrOrd a, int k, int i) {
    i >= -1 && i < length(a) && // index in array bounds
    (i >= 0 ?
```

```
30          a[i] == k : // target found, index positive
31          (forall x : int . 0 <= x && x < length(a)
32              ==> a[x] != k) // target not found, index -1
33      )
34  }
```

The function returns the index of the target element in that array if it is present, and -1 otherwise. The binary search requires that the array is sorted and this is specified by the type `IntArrOrd`. An integer array is sorted if and only if given any two of its valid indexes, the greater index has the greater integer, or equivalently, if given any two of its consecutive indexes, the second must have an integer greater or equal than the first. Hence, we could define `IntArrOrd` type as follows:

```
1  type IntArrOrd = (a: int[] where forall i: int .
2          i >= 0 && i < length(a) - 1 ==> a[i] <= a[i + 1]);
```

However, if this type definition replaces the other, the Boogie tool cannot verify the invariants in the loop, even though the types are equivalent. The same problem happens also in Dafny when a similar declaration is done. This shows one of the limitations of SafeScript and of other verifiers: quantifiers are not decidable.

Performance The presented benchmarks can be used to establish a execution time comparison between verifiers. Dafny solutions to those benchmarks are available, and its verifier also uses Boogie as an intermediate language and the Z3 SMT solver. Table 6.1 presents the execution time comparison between Dafny and SafeScript.

	Dafny v2.3.0		SafeScript	
	Terminal	Boogie	Terminal	Boogie
Benchmark #1: addition	1,7	0,9	6,1	0,6
Benchmark #1: multiplication	1,9	1,0	6,2	0,6
Benchmark #2: binary search	1,9	1,0	9,7	4,0

Table 6.1: Comparison between Dafny and SafeScript execution time (in seconds)

The time benchmarks presented in table were taken in a machine with an Intel Core i7-7700HQ CPU, with 2.80 GHz and 16 GB of RAM memory, under Windows 10 environment. The times are the average of three runs.

Boogie and Z3 versions were the same for both verifying tools. It was chosen to compare two different times: the execution time of the terminal version of each tool, which includes the parsing, validation and translation to another language (in Dafny, .NET), and the execution time of the Boogie tool given the generated Boogie code during the validation phase.

SafeScript terminal version has a heavy initialization, about 5,5 seconds, namely because it was designed to be used with Eclipse IDE. In the IDE, the initialization is only

done once, and the compilation time is similar to the time that Boogie tool takes to make the necessary validations.

6.2 Comparison with TypeScript

TypeScript has the motto *JavaScript that scales*. It was designed to detect statically common JavaScript errors with the introduction of type declaration. SafeScript has a similar goal, so the comparison with TypeScript is important to understand the power of SafeScript. It will be used examples adapted form the TypeScript documentation [50].

TypeScript has a type system similar to SafeScript, except for the refinement types. The object and arrays types and respective literals have some differences, namely in terms of semantics. SafeScript structural objects correspond to TypeScript interfaces, but TypeScript classes does not have an equivalence into SafeScript.

A simple function that returns the field of an object is presented below, in TypeScript and SafeScript.

```
1 // TypeScript
2 function returnLabel(labeledObj: {label: string}) : string {
3     return labeledObj.label;
4 }
```

```
1 // SafeScript
2 string returnLabel({label: string} labeledObj) {
3     return labeledObj.label;
4 }
```

In this case there is almost no difference, the object type declared in the TypeScript function argument is an interface and behaves like SafeScript objects types: the function can receive all objects that have the field `label` that is a string, and that objects can have other fields as well.

A more complex example, with optional fields in objects, is presented next.

```
1  // TypeScript
2  interface SquareConfig {color?: string; width?: number}
3
4  function createSquare(config: SquareConfig): {color: string;
5          area: number} {
6      let newSquare = {color: "white", area: 100};
7      if (config.color) {
8          newSquare.color = config.color;
9      }
10     if (config.width) {
11         newSquare.area = config.width * config.width;
12     }
13     return newSquare;
```

```
14 }
```

```
1  // SafeScript
2  type SquareConfig = {?color: string, ?width: int};
3
4  {color: string, area: int} createSquare(SquareConfig config) {
5      any newSquare = {color = "white", area = 100};
6      if (config in {color: any}) {
7          newSquare.color = config.color;
8      }
9      if (config in {width: any}) {
10         newSquare.area = config.width * config.width;
11     }
12     return newSquare;
13 }
```

In TypeScript the optional fields in interfaces, identified by the quotation mark, have the same semantics as in SafeScript: if the object has the field, then it must be of the declared type. TypeScript also has flow typing, as this example shows: the object fields can only be accessed in lines 8 and 11 because the if conditions in the previous lines are taken in account. The field checking in the conditions uses the JavaScript coercion that is supported by TypeScript. In SafeScript the verification must be done with the *in type* operator.

TypeScript interface can only defined an object type, and SafeScript type declaration can define any type abbreviation. For example, if at least one of the fields in `SquareConfig` is mandatory, then the type can be represented in SafeScript using the union and intersection types as follows:

```
1  type SquareConfig = {?color: string, ?width: int} &
2          ({color: any} | {width: any});
```

TypeScript supports hierarchy in interfaces (and also classes), so an interface can extend other. A simple example is presented below.

```
1  // TypeScript
2  interface Shape {
3      color: string;
4  }
5
6  interface PenStroke {
7      penWidth: number;
8  }
9
10 interface Square extends Shape, PenStroke {
11     sideLength: number;
12 }
13
```

```
14 function doSomething(x: Square) { /* ... */ }
15
16 doSomething({color: "green", sideLength: 4}) // error!
```

An object of type `Square` must have the three fields: `sideLength`, `penWidth` and `color`. Therefore, the call to the `doSomething` function is not valid, since the object passed as argument does not have the field `penWidth`. Equivalent object types can be declared in SafeScript thanks to intersection types:

```
1  // SafeScript
2  type Shape = {color: string};
3
4  type PenStroke = {penWidth: int};
5
6  type Square = Shape & PenStroke & {sideLength: int};
7
8  void doSomething(Square x) { /* ... */ }
9
10 void main() {
11     doSomething({color = "green", sideLength = 4}) // error!
12 }
```

6.3 SafeRESTScript: More Examples

Chapter 5 presented a simple example of a REST call supported by a HeadREST specification. In this section it is presented a set of more real and complex examples of SafeScript programs that consume the REST API of GitHub, a very popular git based version control.

The first target endpoint, `/users{?since}`, allows to get all registered users of GitHub, in the order that they signed up for GitHub. This is done by pagination: each call does not retrieve all users, but a small list of them, where the begin of the list is defined by the optional parameter `since`. If the parameter is not sent, then the default value is zero. Below it is described this endpoint in HeadREST. Note that the user representation is declared only with the fields that are used in the examples of this section.

```
1  specification GitHub
2
3  type User = {
4      login: String,
5      id: Integer,
6      site_admin: Boolean
7  }
8
9  // since parameter is present
10 {
```

```
    request in {template: {since: Integer}}
}
    get '/users{?since}'
{
    response in {body: User[]} &&
    (forall i: Integer . 0 <= i && i < length(response.body) ==>
        response.body[i].id == i + request.template.since + 1)
}

// since parameter is not present
{
    request in !{template: {since: Any}}
}
    get '/users{?since}'
{
    response in {body: User[]} &&
    (forall i: Integer . 0 <= i && i < length(response.body) ==>
        response.body[i].id == i + 1)
}
```

There is no REST endpoint that gets only the user with a given id. Below it is presented a SafeRESTScript program with a function that does exactly that. It takes advantage of the endpoint specification to prove the return type, ensuring that the obtained user has the same id as the function argument.

```
specification "GitHub.hrest" of "https://api.github.com"

type NotFoundError = (x: string where x == "not found");

async (u: User where u.id == id)|NotFoundError getUserById(int id) {
    Request request = {template = {since = id - 1}};
    Response response = await get "/users{?since}" request;
    User[] users = response.body;
    if (length(users) > 0) {
        return users[0]; // valid id
    }
    else {
        return "not found"; // invalid id
    }
}
```

In the successful case, the object body has a non-empty array, the function returns the user in the position zero. This user object is guaranteed to have an id equal to the function parameter id, thanks to the post-condition of the specification triple, namely the forall quantifier that asserts the id of each user according to the `since` variable.

Another possible operation over this endpoint is searching for an user with a certain characteristics. The SafeRESTScript function presented below searches over the GitHub

users to find an administrator, characteristic ensured by the field `site_admin` of an user representation.

```
14
15 type AdminUser = (u: User where u.site_admin);
16 type NoAdminError = (x: string where x == "There are no admins!");
17
18 async AdminUser|NoAdminError getSiteAdmin() {
19     nat i = 0; // start from the first user
20     nat j = 0;
21     User[] users = [];
22     while (true) {
23         users = (await get "/users{?since}"
24                 {template = {since = i}}).body;
25         j = 0;
26         while (j < length(users)) {
27             if (users[j].site_admin) {
28                 return users[j];
29             }
30             j = j + 1;
31         }
32         if (length(users) == 0) { // empty page
33             return "There are no admins!";
34         }
35         i = i + j - 1;
36     }
37 }
```

The search is done getting the various pages of users and stopping when one of them has an user with admin privileges, or returning an error message when all pages were checked and non admin user was found. It is guarantee that the returned user representation (if any) is an administrator thanks to the flow typing caused by the if conditional at line 27, which allows to statically validate the return type.

For the next example it is necessary to introduce two additional GitHub endpoints and resource representations. Each user has a set of repositories, that has a set of collaborators, and each repository contains a list of commits, that identify the collaborated user that made the commit (author). The endpoint `/repos/{owner}/{repo}/collaborators` retrieves the collaborators of a given repository `repo` for a given user `owner`, and the endpoint `/repos/{owner}/{repo}/collaborators` retrieves the repository list of commits. Below this endpoints are specified in HeadREST, extending the specification presented before. Only the successful cases of each endpoint are presented.

```
30 type Repository = {
31     id: Integer,
32     name: String,
33     owner: User
```

```
34 }
35
36 type Commit = {
37     commit: {
38         message: String
39     },
40     author: User
41 }
42
43 {
44     request in {template: {owner: String, repo: String}}
45 }
46     get '/repos/{owner}/{repo}/collaborators'
47 {
48     response in {body: User[]} &&
49     (exists i: Integer . 0 <= i && i < length(response.body) ==>
50         response.body[i].login == request.template.owner)
51 }
52
53 {
54     request in {template: {owner: String, repo: String}}
55 }
56     get '/repos/{owner}/{repo}/commits'
57 {
58     response in {body: Commit[]}
59 }
```

Using these endpoints it is possible to program a function that verifies the collaborators of an repository that did not contribute to the project, i.e., did not make a commit. The correspondent SafeRESTScript function is presented next.

```
34 type DidCommit = {user: string, isCommitter: boolean};
35
36 async DidCommit[] checkUncommitters(string repositoryUser,
37         string repository, string key) {
38     nat i = 0;
39     nat j = 0;
40     Request request = {template = {owner = repositoryUser,
41             repo = repository}, header = {basicAuthorization = key}};
42     User[] users = (await get "/repos/{owner}/{repo}/collaborators"
43             request).body;
44     Commit[] commits = (await get "/repos/{owner}/{repo}/commits"
45             request).body;
46     (d: DidCommit[] where length(d) == length(users)) didCommit =
47             mkarray({user = "", isCommitter = false}, length(users));
48     while (i < length(users)) {
49         didCommit[i].user = users[i].login;
```

```
50          i = i + 1;
51      }
52      i = 0;
53      while (i < length(commits)) {
54          j = 0;
55          while (j < length(didCommit)) {
56              if (didCommit[j].user == commits[i].author.login) {
57                  didCommit[j].isCommitter = true;
58              }
59              j = j + 1;
60          }
61          i = i + 1;
62      }
63      return didCommit;
64  }
```

The function crosses the information obtained in both endpoint calls to return an array with all repository collaborators and if they made a commit or not. 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 (as explained in section 5.5). In this case the HeadREST specification triples are essentially important to validate statically sequential access to the body object of both endpoints responses, like in lines 49 and 56.

6.4 Limitations

From the evaluation presented in this chapter, and also from the languages theory defined in previous chapters, it is possible to identify some limitations of SafeScript language and its extension SafeRESTScript. This section discusses them.

No References SafeScript does not have references, so when an object or array is passed as a function argument, it is not passed a reference but the object or array value. Therefore, functions do not have side effects over the values of variables provided as arguments. On one hand, this can be good, since it guarantees that when calling a function the state of the local variables cannot change, simplifying the validation. On the other hand, for having functions that change and object or array, it is necessary to have the changing object or array in the returning value, which can be complicated in some cases. For example, in a function that performs the traditional pop operation over a stack, it is necessary to return the top element and also the remaining stack. As there are no tuples and a function only returns a single value, this requires the creation of an object structure for the effect.

Since SafeScript compiles to JavaScript (that has references), it is necessary to introduce copies of objects and arrays when necessary, as explained in section 4.5, and which carries an unnecessary complexity to the generated JavaScript, as well as complicating

the transition from one language to other. In the next section it is discussed a possible way of adding references to SafeScript.

Partial correctness SafeScript only ensures a partial correction, i.e., that the result of a function belongs to the function return type. For total correctness, it would be additionally necessary to prove that each function terminates. Two aspects prevent a trivial termination proof: while loops and recursive calls (possibly indirect). Relatively to the loops, there is an additional rule in Hoare calculus that, substituting the traditional while rule, guarantees a total correctness. However, in a general case, it is not possible to apply the rule without additional information provided by the loop statement.

URL construction In SafeScript, the relative URL endpoint for making a REST call must be a string literal, that matches an URI template in the imported HeadREST specification. Therefore, it is not possible to construct the URL string with the values of the URI template variables, but these must be indirectly indicated in the object request. The problem is that it is common in REST APIs to have URLs as parte of resources representations, linking one resource to another. For example, in the GitHub API described before, the user representation contains a hyperlink to the user repositories, which can be used as an URL in another REST call. However, the syntax of SafeRESTScript does not allow this, namely because the necessary correspondence with the an URI template of the HeadREST specification, that must be made during the validation phase.

Error messages Good compilation error messages are essential in a programming language. In a case of type errors, it is common to have an error message with at least two important information: the expected type and the actual type of the expression that is causing the typing error. In SafeScript, it is only possible to inform the programmer with the expected type, because the typing algorithm does not synthesize the type of the expressions. Note that Boogie evaluates an assertion with the expression *in type* operator, and not with a subtyping relation. In contrast, HeadREST does not have this problem, since the type validation algorithm synthesize the types of the expressions, which are then shown in the error messages.

6.5 Future Work

This section presents a set of identified challenges that can improve SafeScript language and its extension SafeRESTScript.

References Introducing references in objects and arrays is not trivial and adds additional complexity to Boogie translation, which may difficult Boogie tool validation, in-

creasing the compilation time. On the other hand, it makes SafeScript even more similar to JavaScript and simplifies the operational semantics of object and arrays updates.

Dafny has a clever solution with object references translation to Boogie [43]: the object heap is a global variable that maps references to values, which can be also other references. The state of the heap is defined in a procedure requires clause, and the procedure changes are reported in the ensures clause. The objects that can change in the procedure have to declared in the function signature, so the post-condition of the heap can be formed: asserting that all existing references that does not belong to these objects maintain the same value in the heap, and others have a new undefined value. In object updates, the left hand side is reduced to a reference and this is updated in heap variable. A similar procedure can be accomplish for arrays as well.

However, this solution only works because Dafny does not have a subtyping relation between types, since there is no type hierarchy. If this solution is directly applied to SafeScript, a problem emerges, as the following example demonstrates:

```
1 void m1({x: int} a) /* modifies a */ {
2     a.x = -1; // error?
3 }
4
5 void m2() {
6     {x: nat} v = {x = 1};
7     m1(v);
8 }
```

By definition, `{x: nat}` is subtype of `{x: int}`, so the variable `v` may be an argument of a call to the function `m1`. The function `m1` change the field `x` to a negative value, that belongs to the type `int`, declared in the function signature. However, this does not belong to the `nat` type of the variable `v`, and therefore when the call to the function `m1` ends, `v` will have a value that does not belong to its declared type.

This problem is well known in some programming languages. It happens, for example, when writing arrays in Java. In SafeScript it can also happen in objects because the subtyping relation is structural for objects. A possible solution is to have an invariant validation for expressions arguments in function calls that can be modified inside the function. For example, in the case presented above, `v` could only be used as `m1` argument if, and only if, had the declaration type `{x: int}` (or other semantically equivalent).

This solution uncovers other challenge: determine the declaration type of a general expression. This is trivial if the expression is a variable, but not if it is a consecutive object or arrays access, namely when the declaration type has refinement types. As it was presented in HeadREST validation, performing a type normalization and extraction may not cover all possible cases. In short, there is still a lot of work to do for introducing references in SafeScript.

Functions as values Functions are values in JavaScript: they can be stored in variables or sent as arguments in a function call. They are particularly useful in Javascript due to its asynchronicity, since callbacks functions are necessary in several function calls. Adding functions as values will make SafeScript one step closer to JavaScript. It is also necessary to add a new type, the function type, that will increase the expressivity of the language type system. The major challenge is to represent the function in Boogie, as well as translating the *in type* relation of the function type. The subtyping relation in the function type must be contravariant in its arguments and convariant in its return type:

$$\frac{T_i <: U_i \quad \forall i \in 1..n \quad U_{n+1} <: T_{n+1}}{(U_1, \ldots, U_n) \to U_{n+1} <: (T_1, \ldots, T_n) \to T_{n+1}}$$

External functions As SafeScript compiles to JavaScript, it has available JavaScript standard libraries, and others that can be imported. To use these functions in SafeScript, taking advantage from all JavaScript existing libraries, it is necessary to annotate the functions, adding types to the parameters and to the return function value, so the semantic of these functions can be known in the validation phase. So, a new function declaration syntax should be added: a function declaration without body, that represents a primitive JavaScript function. A similar concept is done in TypeScript, with declaration files, where external JavaScript function, interfaces, or variables, can be declared, annotated with the TypeScript types, to be used in TypeScript code.

Safety prove One of the most important properties of a type system if safety (also known as soundness), which, in this case, states that a well typed expression never gets stuck during the evaluation stage. From the rules presented in chapter 4 it in principle should be possible to prove the safety of SafeScript, or at least for a subset of this. Two steps are necessary for this proof: a progress theorem, which asserts that a well-typed expression, $\Gamma \vdash e : T$, is a value or can take a step according to the evaluation rules; and a preservation theorem, which asserts that if a well-typed term takes a small step in the evaluation rules, then the resulting term is also well typed: $\Gamma \vdash e : T \ \wedge \ e \mid \mu \longrightarrow e' \mid \mu \Rightarrow \Gamma \vdash e' : T$. As expression evaluation depends on other SafeScript grammar terms, like statements, similar proofs for them must be achieved.

Inconsistent specifications HeadREST specifications may have inconsistent triples, i.e., the intersection of the post-conditions of two or more triple over the same endpoint and that the intersection of the pre-condition is not false, may be false. For example, if a, b, c are boolean expressions, then the triples $\{a\} \ldots \{c\}$ and $\{b\} \ldots \{!c\}$ are inconsistent if $a \wedge b \neq \varnothing$. This aspect does not influence the validation of a HeadREST specification, since each triple is validated independently of the others, but it can affect the validation of a SafeRESTScript program. If a specification has inconsistent triples, then the Boogie

axiomatization generated (that contains all triples) is also inconsistent, and the respective program is validated, even with errors. Therefore, adding a detection of inconsistent specification in HeadREST validation will improve the validation of SafeRESTScript programs.

Specifications with resources HeadREST resource types and the resource related operations `repof` and `uriof` cannot be translated to SafeScript expressions. The value of these expressions depend on the state of the system resources, which cannot be known in compilation time, nether in execution by the client.

A solution is to translate these operators as non interpreted functions, similar to what is done in HeadREST validation. This solution has two problems. It demands that each pair method endpoint must be described for all of its request domain. It also may assert a post-condition about the resource set, which can not be considered as a true assertion in the evaluation of a possible next REST call.

For example, for some service, and in a client-side perspective, if it is created a resource, there is the possibility that in a next call to get the resource representation the resource may already not exist (may had been deleted by another service client), so the post-condition about the resource creation must not be considered in the evaluation of sequential calls pre-conditions.

An alternative is to remove the expressions containing resources references from the triple pre and post-conditions, maintaining the expressions semantic. This solution is not trivial, namely because of the expressivity of the language semantics. So, it may be chosen a syntactic approach to remove the resource expressions.

As discussed in chapter 3, pre and post-conditions describe the format of the request and response, and the state of the system resource set. In each condition these descriptions normally do not overlap, being connected by a boolean logic operator to form the complete condition. So, a simple removal over the boolean *and* and *or* binary operator solve most cases. For more complicated and unusual cases, it may be necessary to create a separate specification that only specifies the format of the request and response.

Chapter 7

Conclusion

REST is the architecture most used for exchanging data in the web. Many web services have their data available in a REST service, so web clients can retrieve and, in some cases, change them. With the high rise of REST clients, it was expected that there were several and potentially good solutions to statically verify REST calls made by clients. However, the real state of the art does not indicate that: there are few solutions, and these tend to be very limited. This aspect is even more relevant as the principal language used for REST clients is JavaScript, which has a weak static validation.

To address the problem presented, this work introduces two new programming languages: SafeScript, a subset of JavaScript equipped with types and a strong static analyses; and SafeRESTScript, an extension of SafeScript that allows making REST calls and statically validates them against a HeadREST specification of the service.

The HeadREST specification language distinguishes from other REST specification languages by its expressivity. It uses a type system with refinement types, and also a powerfully predicate that checks whether an expression belongs to a type, both responsible for allow a rigorous description of the request and response formats of each service endpoint.

SafeScript adds a expressive type system to JavaScript syntax, including refinement types. It was presented a declarative type system and a Boogie translation that substitutes an algorithmic system. The Boogie tool validates the generated code with the help of Z3 SMT solver, ensuring a semantic evaluation of the type system. This validation system guarantees the detections of common runtime errors, like null dereference, division by zero, or access outside arrays bounds, aspects that others JavaScript based languages, as TypeScript, cannot ensure in compilation time.

Thanks to the expressive type system, SafeScript compiler is also a verifier. It was exposed some examples that demonstrate the proving capacity of the language validator. Theoretical limitations regarding the indecibility of quantifiers are currently impeding more complex and interesting proves.

The SafeRESTScript language extension introduces REST calls to SafeScript. Using a HeadREST specification of the service, the language validator checks if the REST calls

are made into a valid service endpoint, and verifies whether each call is according with the pre and post-conditions of the respective endpoint specification. To demonstrate the capacity of the language, it was exemplified a SafeRESTScript client of GitHub API.

Both languages have some limitations; future work was identified to improve them. Unlike TypeScript or Flow, SafeScript and SafeRESTScript do not have the ambition of being production languages, but rather contribute to advance the state of the art of static analyses, and demonstrate that is possible to improve the support for writing reliable code that consumes REST services. The languages compiler and editor is available as an Eclipse IDE plugin.

Finally, and in a more personal retrospective, this work was essential to really understand the world of reliable software, namely static analyses: its difficulty, its limitations, and the vast work yet to be done. I hope that this work, contextualized in a final academic learning process, has contributed, even at an infinitesimal level, for a more reliable programming.

Appendix A

HeadREST type normalization and extraction

Normal disjunction	D	$::= R_1 \vee \cdots \vee R_n$	$(n \geq 0, \mathsf{Empty}\ \text{when}\ n = 0)$
Normal refined conjunction	R	$::= x \colon C\ \mathsf{where}\ e$	
Normal conjunction	C	$::= A_1 \wedge \cdots \wedge A_n$	$(n \geq 0, \mathsf{Any}\ \text{when}\ n = 0)$
Atomic type	A	$::= G \mid \alpha \mid \{\} \mid \{l \colon D\} \mid D\,[]$	

Figure A.1: Disjunctive normal form types (DNF): D

$$\mathrm{norm}(\mathsf{Any}) \triangleq \mathsf{Any}$$
$$\mathrm{norm}(G) \triangleq y\colon G \text{ where true}$$
$$\mathrm{norm}(\alpha) \triangleq y\colon \alpha \text{ where true}$$
$$\mathrm{norm}(\{\}) \triangleq y\colon \{\} \text{ where true}$$
$$\mathrm{norm}(\{l\colon T\}) \triangleq y\colon \{l\colon T\} \text{ where true}$$
$$\mathrm{norm}(T\,[]) \triangleq y\colon T\,[] \text{ where true}$$
$$\mathrm{norm}(x\colon T \text{ where } e) \triangleq \bigvee_{i=1}^{n} \mathrm{conj}_{\mathrm{DD}}(x_i\colon C_i \text{ where } e_i, \mathrm{norm}_r(x\colon C_i \text{ where } e))$$
$$\text{if } \bigvee_{i=1}^{n}(x_i\colon C_i \text{ where } e_i) = \mathrm{norm}_r(T)$$

$$\mathrm{norm}_r(x\colon C \text{ where } x \text{ in } T) \triangleq \mathrm{norm}(C \,\&\, T) \qquad \text{if } x \notin \mathrm{fv}(T)$$
$$\mathrm{norm}_r(x\colon C \text{ where } e_1 \mid e_2) \triangleq \mathrm{norm}_r(x\colon C \text{ where } e_1) \vee \mathrm{norm}_r(x\colon C \text{ where } e_2)$$
$$\mathrm{norm}_r(x\colon C \text{ where } e_1 \,\&\, e_2) \triangleq \mathrm{conj}_{\mathrm{DD}}(\mathrm{norm}_r(x\colon C \text{ where } e_1), \mathrm{norm}_r(x\colon C \text{ where } e_2))$$
$$\mathrm{norm}_r(x\colon C \text{ where } e) \triangleq x\colon C \text{ where } e \qquad \text{otherwise}$$

$$\mathrm{conj}_{\mathrm{DD}}((R_1 \vee \cdots \vee R_n), D) \triangleq \mathrm{conj}_{\mathrm{RD}}(R_1, D) \vee \cdots \vee \mathrm{conj}_{\mathrm{RD}}(R_n, D)$$
$$\mathrm{conj}_{\mathrm{RD}}(R, (R_1 \vee \cdots \vee R_n)) \triangleq \mathrm{conj}_{\mathrm{RR}}(R, R_1) \vee \cdots \vee \mathrm{conj}_{\mathrm{RR}}(R, R_n)$$
$$\mathrm{conj}_{\mathrm{RR}}(x_1\colon C_1 \text{ where } e_1, x_2\colon C_2 \text{ where } e_2) \triangleq y\colon C_1 \wedge C_2 \text{ where } [y/x_1]e_1 \,\&\, [y/x_2]e_2$$

where y is a fresh variable in all cases

Figure A.2: Type normalisation: $\mathrm{norm}(T) = D$

$$\frac{R_i.l \rightsquigarrow U_i \quad \forall i \in 1..n}{(R_1 \mid \ldots \mid R_n).l \rightsquigarrow (U_1 \mid \ldots \mid U_n)} \qquad \text{(Field Disj)}$$

$$\frac{C.l \rightsquigarrow U}{(x\colon C \text{ where } e).l \rightsquigarrow U} \qquad \text{(Field Refine)}$$

$$\frac{(S = \{U_i \mid A_i.l \rightsquigarrow U_i\}) \neq \emptyset}{(A_1 \,\&\, \ldots \,\&\, A_n).l \rightsquigarrow (\&\, S)} \qquad \frac{}{\{l : T\}.l \rightsquigarrow T} \qquad \text{(Field Conj, Field Atom)}$$

Figure A.3: Extraction of field type: $D.l \rightsquigarrow U$

$$\frac{R_i.\mathsf{Items} \rightsquigarrow U_i \quad \forall i \in 1..n}{(R_1 \mid \ldots \mid R_n).\mathsf{Items} \rightsquigarrow (U_1 \mid \ldots \mid U_n)} \qquad \text{(Items Disj)}$$

$$\frac{C.\mathsf{Items} \rightsquigarrow U}{(x\colon C \ \mathsf{where}\ e).\mathsf{Items} \rightsquigarrow U} \qquad \text{(Items Refine)}$$

$$\frac{(S = \{U_i \mid A_i.\mathsf{Items} \rightsquigarrow U_i\}) \neq \emptyset}{(A_1 \,\&\, \ldots \,\&\, A_n).\mathsf{Items} \rightsquigarrow (\&\, S)} \qquad \frac{}{T\,[].\mathsf{Items} \rightsquigarrow T} \qquad \text{(Items Conj, Items Atom)}$$

Figure A.4: Extraction of item type: $D.\mathsf{Items} \rightsquigarrow U$

Appendix B

SMT-LIB Axiomatization in HeadREST

```
(set-info :smt-lib-version 2.0)

(set-option :auto_config false)
(set-option :smt.mbqi false)

(set-option :smt.string_solver z3str3)

(set-option :model_evaluator.completion false)
(set-option :model.v1 true)
(set-option :smt.phase_selection 0)
(set-option :smt.restart_strategy 0)
(set-option :smt.restart_factor 1.5)
(set-option :nnf.sk_hack true)
(set-option :smt.qi.eager_threshold 100.0)
(set-option :smt.arith.random_initial_value true)
(set-option :smt.case_split 3)
(set-option :smt.delay_units true)
(set-option :smt.delay_units_threshold 16)
(set-option :type_check true)
(set-option :smt.bv.reflect true)
(set-option :smt.timeout 2000)

; --------------------------------
; Values
; --------------------------------

(declare-datatypes () ((U_VarList
    EmptyList
    (U_Vars (headVar String) (tailVars U_VarList))
)))

(declare-datatypes () ((U_Fragment
    (U_Literal (of_U_Literal String))
```

```
34      (U_Expression (of_U_Expression U_VarList) (optional Bool))
35  )))
36
37  (declare-datatypes () ((UriTemplate
38      EmptyUriTemplate
39      (U_Fragments (headFragment U_Fragment) (tailFragments
        UriTemplate))
40  )))
41
42  (declare-datatypes () ((General
43      (G_Boolean (of_G_Boolean Bool))
44      (G_Integer (of_G_Integer Int))
45      (G_String (of_G_String String))
46      (G_Regexp (of_G_Regexp (RegEx String)))
47      (G_UriTemplate (of_G_UriTemplate UriTemplate))
48      G_Null
49  )))
50
51  (declare-sort SVMap)
52  (declare-sort IVMap)
53
54  (declare-datatypes () ((Value
55      (G (out_G General))
56      (O (out_O SVMap))
57      (A (out_A IVMap) (length Int))
58      (R (id Int) (type String))
59  )))
60
61  (declare-datatypes () ((ValueOption
62      NoValue
63      (SomeValue (of_SomeValue Value))
64  )))
65
66  (declare-fun Good_A (Value) Bool)
67  (assert (forall ((v Value))
68      (! (iff
69          (Good_A v)
70          (is-A v)
71      ) :pattern(Good_A v))
72  ))
73
74  (declare-fun Good_O (Value) Bool)
75  (assert (forall ((v Value))
76      (! (iff
77          (Good_O v)
78          (is-O v)
```

```
79      ) :pattern(Good_O v))
80  ))
81
82  (declare-fun Good_R (Value) Bool)
83  (assert (forall ((v Value))
84      (! (iff
85          (Good_R v)
86          (is-R v)
87      ) :pattern(Good_R v))
88  ))
89
90  ; ----------------------------------
91  ; Operations
92  ; ----------------------------------
93
94  (declare-const v_tt Value)
95  (declare-const v_ff Value)
96  (declare-const v_null Value)
97
98  (assert (= v_tt (G (G_Boolean true))))
99  (assert (= v_ff (G (G_Boolean false))))
100 (assert (= v_null (G G_Null)))
101
102 (declare-fun In_Boolean (Value) Bool)
103 (assert (forall ((v Value))
104     (! (=
105         (In_Boolean v)
106         (and (is-G v) (is-G_Boolean (out_G v)))
107     ) :pattern(In_Boolean v))
108 ))
109
110 (declare-fun In_Integer (Value) Bool)
111 (assert (forall ((v Value))
112     (! (=
113         (In_Integer v)
114         (and (is-G v) (is-G_Integer (out_G v)))
115     ) :pattern(In_Integer v))
116 ))
117
118 (declare-fun In_String (Value) Bool)
119 (assert (forall ((v Value))
120     (! (=
121         (In_String v)
122         (and (is-G v) (is-G_String (out_G v)))
123     ) :pattern(In_String v))
124 ))
```

```
125
126 (declare-fun In_Regexp (Value) Bool)
127 (assert (forall ((v Value))
128     (! (=
129         (In_Regexp v)
130         (and (is-G v) (is-G_Regexp (out_G v)))
131     ) :pattern(In_Regexp v))
132 ))
133
134 (declare-fun In_UriTemplate (Value) Bool)
135 (assert (forall ((v Value))
136     (! (=
137         (In_UriTemplate v)
138         (and (is-G v) (is-G_UriTemplate (out_G v)))
139     ) :pattern(In_UriTemplate v))
140 ))
141
142 (declare-fun O_Equiv (Value Value) Value)
143 (declare-fun O_Implies (Value Value) Value)
144 (declare-fun O_Sum (Value Value) Value)
145 (declare-fun O_Sub (Value Value) Value)
146 (declare-fun O_Mult (Value Value) Value)
147 (declare-fun O_IntDiv (Value Value) Value)
148 (declare-fun O_Rem (Value Value) Value)
149 (declare-fun O_EQ (Value Value) Value)
150 (declare-fun O_NE (Value Value) Value)
151 (declare-fun O_Not (Value) Value)
152 (declare-fun O_Minus (Value) Value)
153 (declare-fun O_And (Value Value) Value)
154 (declare-fun O_Or (Value Value) Value)
155 (declare-fun O_GE (Value Value) Value)
156 (declare-fun O_GT (Value Value) Value)
157 (declare-fun O_LT (Value Value) Value)
158 (declare-fun O_LE (Value Value) Value)
159 (declare-fun O_++ (Value Value) Value)
160
161 (assert (forall ((v1 Value) (v2 Value))
162     (! (=
163         (O_Equiv v1 v2)
164         (ite (= v1 v2) v_tt v_ff)
165     ) :pattern(O_Equiv v1 v2))
166 ))
167
168 (assert (forall ((v1 Value) (v2 Value))
169     (! (=
170         (O_Implies v1 v2)
```

```
171         (O_Or (O_Not v1) v2)
172     ) :pattern(O_Implies v1 v2))
173 ))
174
175 (assert (forall ((v1 Value) (v2 Value))
176     (! (=
177         (O_Sum v1 v2)
178         (G (G_Integer (+ (of_G_Integer (out_G v1)) (of_G_Integer
    (out_G v2)))))
179     ) :pattern(O_Sum v1 v2))
180 ))
181
182 (assert (forall ((v1 Value) (v2 Value))
183     (! (=
184         (O_Sub v1 v2)
185         (G (G_Integer (- (of_G_Integer (out_G v1)) (of_G_Integer
    (out_G v2)))))
186     ) :pattern(O_Sub v1 v2))
187 ))
188
189 (assert (forall ((v1 Value) (v2 Value))
190     (! (=
191         (O_Mult v1 v2)
192         (G (G_Integer (* (of_G_Integer (out_G v1)) (of_G_Integer
    (out_G v2)))))
193     ) :pattern(O_Mult v1 v2))
194 ))
195
196 (assert (forall ((v1 Value) (v2 Value))
197     (! (=
198         (O_IntDiv v1 v2)
199         (G (G_Integer (div (of_G_Integer (out_G v1)) (of_G_Integer
    (out_G v2)))))
200     ) :pattern(O_IntDiv v1 v2))
201 ))
202
203 (assert (forall ((v1 Value) (v2 Value))
204     (! (=
205         (O_Rem v1 v2)
206         (G (G_Integer (rem (of_G_Integer (out_G v1)) (of_G_Integer
    (out_G v2)))))
207     ) :pattern(O_Rem v1 v2))
208 ))
209
210 (assert (forall ((v1 Value) (v2 Value))
211     (! (=
```

```
212         (O_EQ v1 v2)
213         (ite (= v1 v2) v_tt v_ff)
214     ) :pattern(O_EQ v1 v2))
215 ))
216
217 (assert (forall ((v1 Value) (v2 Value))
218     (! (=
219         (O_NE v1 v2)
220         (ite (= v1 v2) v_ff v_tt)
221     ) :pattern(O_NE v1 v2))
222 ))
223
224 (assert (forall ((v Value))
225     (! (=
226         (O_Not v)
227         (ite (not (= v v_tt)) v_tt v_ff)
228     ) :pattern(O_Not v))
229 ))
230
231 (assert (forall ((v Value))
232     (! (=
233         (O_Minus v)
234         (G (G_Integer (- (of_G_Integer (out_G v)))))
235     ) :pattern(O_Minus v))
236 ))
237
238 (assert (forall ((v1 Value) (v2 Value))
239     (! (=
240         (O_And v1 v2)
241         (ite (and (= v1 v_tt) (= v2 v_tt)) v_tt v_ff)
242     ) :pattern(O_And v1 v2))
243 ))
244
245 (assert (forall ((v1 Value) (v2 Value))
246     (! (=
247         (O_Or v1 v2)
248         (ite (or (= v1 v_tt) (= v2 v_tt)) v_tt v_ff)
249     ) :pattern(O_Or v1 v2))
250 ))
251
252 (assert (forall ((v1 Value) (v2 Value))
253     (! (=
254         (O_GE v1 v2)
255         (ite (>= (of_G_Integer (out_G v1)) (of_G_Integer (out_G
        v2))) v_tt v_ff)
256     ) :pattern(O_GE v1 v2))
```

```
257 ))
258
259 (assert (forall ((v1 Value) (v2 Value))
260     (! (=
261         (O_GT v1 v2)
262         (ite (> (of_G_Integer (out_G v1)) (of_G_Integer (out_G v2)))
        v_tt v_ff)
263     ) :pattern(O_GT v1 v2))
264 ))
265
266 (assert (forall ((v1 Value) (v2 Value))
267     (! (=
268         (O_LT v1 v2)
269         (ite (< (of_G_Integer (out_G v1)) (of_G_Integer (out_G v2)))
        v_tt v_ff)
270     ) :pattern(O_LT v1 v2))
271 ))
272
273 (assert (forall ((v1 Value) (v2 Value))
274     (! (=
275         (O_LE v1 v2)
276         (ite (<= (of_G_Integer (out_G v1)) (of_G_Integer (out_G
        v2))) v_tt v_ff)
277     ) :pattern(O_LE v1 v2))
278 ))
279
280 (assert (forall ((v1 Value) (v2 Value))
281     (! (=
282         (O_++ v1 v2)
283         (G (G_String (str.++ (of_G_String (out_G v1)) (of_G_String
        (out_G v2)))))
284     ) :pattern(O_++ v1 v2))
285 ))
286
287 ; ----------------------------------
288 ; Primitive operators
289 ; ----------------------------------
290
291 (declare-fun v_size (Value) Value)
292 (declare-fun v_matches (Value Value) Value)
293 (declare-fun v_pre (Value) Value)
294
295 ;; Link v_size to str.len
296 (assert (forall ((v Value))
297     (! (=
298         (v_size v)
```

```
299         (G (G_Integer (str.len (of_G_String (out_G v)))))
300     ) :pattern((v_size v)))
301 ))
302
303 (assert (forall ((v1 Value) (v2 Value))
304     (! (=
305         (v_matches v1 v2)
306         (G (G_Boolean (str.in.re (of_G_String (out_G v2))
        (of_G_Regexp (out_G v1)))))
307     ) :pattern((v_matches v1 v2)))
308 ))
309
310 ;; pre internal function is only used for repof and uriof operations,
311 ;; so it is only necessary to define for the boolean case
312 (assert (forall ((v Value))
313     (! (=>
314         (In_Boolean v)
315         (In_Boolean (v_pre v))
316     ) :pattern((v_pre v)))
317 ))
318
319 ; ---------------------------------
320 ; Objects
321 ; ---------------------------------
322
323 ;; Entity related sorts/functions
324 (define-sort SVMapArray () (Array String ValueOption))
325 (declare-fun alphas (SVMap) SVMapArray)
326 (declare-fun betas (SVMapArray) SVMap)
327
328 (declare-fun v_dot (Value String) Value)
329 (declare-fun v_has_field (Value String) Bool)
330
331 ;; SVMap and the arrays in SVMapArray are isomorphic
332 (assert (forall ((am SVMapArray))
333     (! (= (alphas (betas am)) am)
334     :pattern(alphas (betas am)))
335 ))
336 (assert (forall ((svm SVMap))
337     (! (= (betas (alphas svm)) svm)
338     :pattern(betas (alphas svm)))
339 ))
340
341 (assert (forall ((v Value) (l String))
342     (! (iff
343         (v_has_field v l)
```

```
344         (not (= (select (alphas (out_O v)) l) NoValue))
345     ) :pattern(v_has_field v l))
346 ))
347
348 (assert (forall ((v Value) (l String))
349     (! (=
350         (v_dot v l)
351         (of_SomeValue (select (alphas (out_O v)) l))
352     ) :pattern(v_dot v l))
353 ))
354
355 ; --------------------------------
356 ; Arrays
357 ; --------------------------------
358
359 ;; Array related sorts/functions
360 (define-sort IVMapArray () (Array Int ValueOption))
361 (declare-fun alphai (IVMap) IVMapArray)
362 (declare-fun betai (IVMapArray) IVMap)
363
364 (declare-fun v_nth (Value Value) Value)
365 (declare-fun v_array_has_value (Value Int) Bool)
366 (declare-fun v_length (Value) Value)
367
368 ;; IVMap and the arrays in IVMapArray are isomorphic
369 (assert (forall ((am IVMapArray))
370     (! (= (alphai (betai am)) am)
371     :pattern(alphai (betai am)))
372 ))
373
374 (assert (forall ((ivm IVMap))
375     (! (= (betai (alphai ivm)) ivm)
376     :pattern(betai (alphai ivm)))
377 ))
378
379 (assert (forall ((v Value) (i Int))
380     (! (iff
381         (v_array_has_value v i)
382         (not (= (select (alphai (out_A v)) i) NoValue))
383     ) :pattern(v_array_has_value v i))
384 ))
385
386 (assert (forall ((v Value) (i Int))
387     (! (iff
388         (v_array_has_value v i)
389         (and (Good_A v) (>= i 0) (< i (length v)))
```

```
390     ) :pattern(v_array_has_value v i))
391 ))
392
393 (assert (forall ((v Value) (i Value))
394     (! (=
395         (v_nth v i)
396         (of_SomeValue (select (alphai (out_A v)) (of_G_Integer
        (out_G i))))
397     ) :pattern(v_nth v i))
398 ))
399
400 (assert (forall ((v Value))
401     (! (=>
402         (Good_A v)
403         (=
404             (v_length v)
405             (G (G_Integer (length v)))
406         )
407     ) :pattern(v_length v))
408 ))
409
410 ; ---------------------------------
411 ; Resources
412 ; ---------------------------------
413
414 (declare-fun r_repof (Value Value) Value)
415 (declare-fun r_uriof (Value Value) Value)
416
417 (declare-fun is_resource_of (Value String) Bool)
418 (assert (forall ((v Value) (s String))
419     (! (=
420         (is_resource_of v s)
421         (= (type v) s)
422     ) :pattern(is_resource_of v s))
423 ))
424
425 ; ---------------------------------
426 ; Expand of UriTemplate
427 ; ---------------------------------
428
429 (declare-fun v_expand (Value Value) Value)
430 (declare-fun expand (UriTemplate Value) String)
431 (declare-fun expandFragment (U_Fragment Value) String)
432 (declare-fun expandVars (U_VarList Value) String)
433 (declare-fun expandOptionalVars (U_VarList Value Bool) String)
434
```

```
435 (declare-fun toString (Value) String)
436 (declare-fun intToString (Int) String)
437 (declare-fun intToStringAux (Int) String)
438 (declare-fun arrayToString (Value Int) String)
439
440 (assert (forall ((v1 Value) (v2 Value))
441     (! (=
442         (v_expand v1 v2)
443         (G (G_String (expand (of_G_UriTemplate (out_G v1)) v2)))
444     ) :pattern(v_expand v1 v2))
445 ))
446
447 (assert (forall ((ut UriTemplate) (v Value))
448     (! (=
449         (expand ut v)
450         (ite (is-EmptyUriTemplate ut)
451             ""
452             (str.++ (expandFragment (headFragment ut) v) (expand
    (tailFragments ut) v))
453         )
454     ) :pattern(expand ut v))
455 ))
456
457 (assert (forall ((uf U_Fragment) (v Value))
458     (! (=
459         (expandFragment uf v)
460         (ite (is-U_Literal uf)
461             (of_U_Literal uf)
462             (ite (optional uf)
463                 (str.++ "?" (expandOptionalVars (of_U_Expression uf)
    v false))
464                 (expandVars (of_U_Expression uf) v)
465             )
466         )
467     ) :pattern(expandFragment uf v))
468 ))
469
470 (assert (forall ((uvl U_VarList) (v Value))
471     (! (=
472         (expandVars uvl v)
473         (ite (is-EmptyList uvl)
474             ""
475             (str.++
476                 (ite (v_has_field v (headVar uvl))
477                     (toString (v_dot v (headVar uvl)))
478                     ""
```

```
479                 )
480                 (expandVars (tailVars uvl) v)
481             )
482         )
483     ) :pattern(expandVars uvl v))
484 ))
485
486 (assert (forall ((uvl U_VarList) (v Value) (b Bool))
487     (! (=
488         (expandOptionalVars uvl v b)
489         (ite (is-EmptyList uvl)
490             ""
491             (ite (v_has_field v (headVar uvl))
492                 (str.++ (ite b "&" "") (headVar uvl) "=" (toString
    (v_dot v (headVar uvl))) (expandOptionalVars (tailVars uvl) v
    true))
493                 (expandOptionalVars (tailVars uvl) v b)
494             )
495         )
496     ) :pattern(expandOptionalVars uvl v b))
497 ))
498
499 (assert (forall ((v Value))
500     (! (=>
501         (In_Boolean v)
502         (=
503             (toString v)
504             (ite (of_G_Boolean (out_G v)) "true" "false")
505         )
506     ) :pattern(toString v))
507 ))
508
509 (assert (forall ((v Value))
510     (! (=>
511         (In_Integer v)
512         (=
513             (toString v)
514             (intToString (of_G_Integer (out_G v)))
515         )
516     ) :pattern(toString v))
517 ))
518
519 (assert (forall ((v Value))
520     (! (=>
521         (In_String v)
522         (=
```

```
523             (toString v)
524             (of_G_String (out_G v))
525         )
526     ) :pattern(toString v))
527 ))
528 
529 (assert (forall ((v Value))
530     (! (=>
531         (and (is-G v) (is-G_Null (out_G v)))
532         (=
533             (toString v)
534             ""
535         )
536     ) :pattern(toString v))
537 ))
538 
539 (assert (forall ((v Value))
540     (! (=>
541         (Good_A v)
542         (=
543             (toString v)
544             (arrayToString v 0)
545         )
546     ) :pattern(toString v))
547 ))
548 
549 (define-const _base String "0123456789")
550 
551 (assert (forall ((n Int))
552     (! (=
553         (intToString n)
554         (ite (= n 0)
555             "0"
556             (ite (> n 0)
557                 (intToStringAux n)
558                 (str.++ "-" (intToStringAux n))
559             )
560         )
561     ) :pattern(intToString n))
562 ))
563 
564 (assert (forall ((n Int))
565     (! (=
566         (intToStringAux n)
567         (ite (= n 0)
568             ""
```

```
569             (str.++ (intToStringAux (div n 10)) (str.at _base (rem n
        10)))
570         )
571     ) :pattern(intToStringAux n))
572 ))
573
574 (assert (forall ((array Value) (i Int))
575     (! (=
576         (arrayToString array i)
577         (ite (= i (length array))
578             ""
579             (ite (= i 0)
580                 (str.++ (toString (v_nth array (G (G_Integer i))))
        (arrayToString array (+ i 1)))
581                 (str.++ "," (toString (v_nth array (G (G_Integer
        i)))) (arrayToString array (+ i 1)))
582             )
583         )
584     ) :pattern(arrayToString array i))
585 ))
```

Appendix C

Boogie Axiomatization of SafeScript and SafeRESTScript

Functions and constants begin with a dot so they do not match possible generated procedures in the translation. Therefore all functions and constants reference in Boogie translation corresponde to ones here prefixed with a dot.

```
type Value;
type MaybeValue;
type Field;

// MaybeValue definitions

function .isNothing(MaybeValue) returns (bool);
function .isPresent(MaybeValue) returns (bool);
function .getValue(MaybeValue) returns (Value);
function .maybeOf(Value) returns (MaybeValue);

axiom (forall mv: MaybeValue :: .isNothing(mv) <==> !.isPresent(mv));
axiom (forall v: Value :: .isPresent(.maybeOf(v)));
axiom (forall v: Value :: .getValue(.maybeOf(v)) == v);
axiom (forall mv: MaybeValue :: .isPresent(mv) ==>
    .maybeOf(.getValue(mv)) == mv);

// Integer values

function .isInt(Value) returns (bool);
function .toInt(Value) returns (int);
function .fromInt(int) returns (Value);

axiom (forall i: int :: .isInt(.fromInt(i)));
axiom (forall i: int :: .toInt(.fromInt(i)) == i);
axiom (forall v: Value :: .isInt(v) ==> .fromInt(.toInt(v)) == v);
```

```
26
27 // Boolean values
28
29 function .isBool(Value) returns (bool);
30 function .toBool(Value) returns (bool);
31 function .fromBool(bool) returns (Value);
32
33 axiom (forall b: bool :: .isBool(.fromBool(b)));
34 axiom (forall b: bool :: .toBool(.fromBool(b)) == b);
35 axiom (forall v: Value :: .isBool(v) ==> .fromBool(.toBool(v)) == v);
36
37 const .True: Value;
38 const .False: Value;
39
40 axiom .True == .fromBool(true);
41 axiom .False == .fromBool(false);
42
43 // Arrays values
44
45 function .isArray(Value) returns (bool);
46 function .toArray(Value) returns ([int]MaybeValue);
47 function .arraylen(Value) returns (int);
48 function .fromArray([int]MaybeValue,int) returns (Value);
49
50 axiom (forall s: [int]MaybeValue, len: int :: 0 <= len ==>
       .isArray(.fromArray(s, len)));
51 axiom (forall s: [int]MaybeValue, len: int :: 0 <= len ==>
       .toArray(.fromArray(s, len)) == s);
52 axiom (forall s: [int]MaybeValue, len: int :: 0 <= len ==>
       .arraylen(.fromArray(s, len)) == len);
53 axiom (forall v: Value :: .isArray(v) ==> .fromArray(.toArray(v),
       .arraylen(v)) == v);
54 axiom (forall v: Value :: .isArray(v) ==> 0 <= .arraylen(v));
55 axiom (forall v: Value, i: int :: .isArray(v) ==> (0 <= i && i <
       .arraylen(v) <==> .isPresent(.toArray(v)[i])));
56
57 function .isValidIndex(a: Value, i: int) returns (bool) {
58     .isPresent(.toArray(a)[i])
59 }
60
61 function .getIndexValue(a: Value, i: int) returns (Value) {
62     .getValue(.toArray(a)[i])
63 }
64
65 function .emptyArray() returns ([int]MaybeValue);
66 axiom (forall i: int :: .isNothing(.emptyArray()[i]));
```

```
67
68  function .constArray(v: Value) returns ([int]MaybeValue);
69  axiom (forall v: Value, i: int :: .getValue(.constArray(v)[i]) == v);
70
71  function .arrayUpdate(a: Value, i: Value, v: Value) returns (Value) {
72      .fromArray(.toArray(a)[.toInt(i) := .maybeOf(v)], .arraylen(a))
73  }
74
75  axiom (forall a: Value, i: Value, v: Value :: {.arrayUpdate(a, i, v)}
76          .isArray(.arrayUpdate(a, i, v)) &&
77          (forall j: int :: .toInt(i) != j && .isValidIndex(a, j) ==>
        .getIndexValue(.arrayUpdate(a, i, v), j) == .getIndexValue(a, j))
78          && .getIndexValue(.arrayUpdate(a, i, v), .toInt(i)) == v
79  );
80
81  axiom (forall a1: Value, a2: Value :: {.isArray(a1), .isArray(a2)}
82      .isArray(a1) && .isArray(a2)
83      ==>
84      ((forall i: int :: .toArray(a1)[i] == .toArray(a2)[i]) <==> a1
        == a2)
85  );
86
87  // Objects values
88
89  function .isObject(Value) returns (bool);
90  function .toObject(Value) returns ([Field]MaybeValue);
91  function .fromObject([Field]MaybeValue) returns (Value);
92
93  axiom (forall s: [Field]MaybeValue :: .isObject(.fromObject(s)));
94  axiom (forall s: [Field]MaybeValue :: .toObject(.fromObject(s)) ==
        s);
95  axiom (forall v: Value :: .isObject(v) ==> .fromObject(.toObject(v))
        == v);
96
97  function .hasField(o: Value, f: Field) returns (bool) {
98      .isPresent(.toObject(o)[f])
99  }
100
101 function .getFieldValue(o: Value, f: Field) returns (Value) {
102     .getValue(.toObject(o)[f])
103 }
104
105 function .emptyObject() returns ([Field]MaybeValue);
106 axiom (forall f: Field :: .isNothing(.emptyObject()[f]));
107
```

```
108 function .objectUpdate(o: Value, f: Field, v: Value) returns (Value)
        {
109     .fromObject(.toObject(o)[f := .maybeOf(v)])
110 }
111
112 axiom (forall o1: Value, o2: Value :: {.isObject(o1), .isObject(o2)}
113     .isObject(o1) && .isObject(o2)
114     ==>
115     ((forall f: Field :: .toObject(o1)[f] == .toObject(o2)[f]) <==>
        o1 == o2)
116 );
117
118 // String values
119
120 function .isString(Value) returns (bool);
121 function .toString(Value) returns ([int]int);
122 function .stringSize(Value) returns (int);
123 function .fromString([int]int,int) returns (Value);
124
125 axiom (forall s: [int]int, len: int :: 0 <= len ==>
        .isString(.fromString(s, len)));
126 axiom (forall s: [int]int, len: int :: 0 <= len ==>
        .toString(.fromString(s, len)) == s);
127 axiom (forall s: [int]int, len: int :: 0 <= len ==>
        .stringSize(.fromString(s, len)) == len);
128 axiom (forall v: Value :: .isString(v) ==> .fromString(.toString(v),
        .stringSize(v)) == v);
129 axiom (forall v: Value :: .isString(v) ==> 0 <= .stringSize(v));
130 axiom (forall v: Value, i: int :: .isString(v) ==> (0 <= i && i <
        .stringSize(v) <==> .toString(v)[i] >= 0));
131
132 function .emptyString() returns ([int]int);
133 axiom (forall i: int :: .emptyString()[i] == -1);
134
135 axiom (forall s1: Value, s2: Value :: {.isString(s1), .isString(s2)}
136     .isString(s1) && .isString(s2)
137     ==>
138     ((forall i: int :: .toString(s1)[i] == .toString(s2)[i]) <==> s1
        == s2)
139 );
140
141 // Function values
142
143 function .isFunction(Value) returns (bool);
144 function .toFunction(Value) returns ([Value]Value);
145 function .argValid(Value) returns ([Value]bool);
```

```
146 function .fromFunction([Value]Value, [Value]bool) returns (Value);
147
148 axiom (forall f: [Value]Value, v: [Value]bool ::
        .isFunction(.fromFunction(f, v)));
149 axiom (forall f: [Value]Value, v: [Value]bool ::
        .toFunction(.fromFunction(f, v)) == f);
150 axiom (forall f: [Value]Value, v: [Value]bool ::
        .argValid(.fromFunction(f, v)) == v);
151 axiom (forall v: Value :: .isFunction(v) ==>
        .fromFunction(.toFunction(v), .argValid(v)) == v);
152
153 function .apply(f: Value, v: Value) returns (Value) {
154     .toFunction(f)[v]
155 }
156
157 // Nothing value and void type
158
159 const .Undefined: Value;
160
161 function .isVoid(v: Value) returns (bool) {
162     v == .Undefined
163 }
164
165 // Primitive functions
166
167 function .equi(x: Value, y: Value) returns (Value) {
168     .fromBool(.toBool(x) <==> .toBool(y))
169 }
170
171 function .imp(x: Value, y: Value) returns (Value) {
172     .fromBool(.toBool(x) ==> .toBool(y))
173 }
174
175 function .or(x: Value, y: Value) returns (Value) {
176     .fromBool(.toBool(x) || .toBool(y))
177 }
178
179 function .and(x: Value, y: Value) returns (Value) {
180     .fromBool(.toBool(x) && .toBool(y))
181 }
182
183 function .eq(x: Value, y: Value) returns (Value) {
184     .fromBool(x == y)
185 }
186
187 function .ne(x: Value, y: Value) returns (Value) {
```

```
188     .fromBool(x != y)
189 }
190
191 function .lt(x: Value, y: Value) returns (Value) {
192     .fromBool(.toInt(x) < .toInt(y))
193 }
194
195 function .le(x: Value, y: Value) returns (Value) {
196     .fromBool(.toInt(x) <= .toInt(y))
197 }
198
199 function .gt(x: Value, y: Value) returns (Value) {
200     .fromBool(.toInt(x) > .toInt(y))
201 }
202
203 function .ge(x: Value, y: Value) returns (Value) {
204     .fromBool(.toInt(x) >= .toInt(y))
205 }
206
207 function .concat(Value, Value) returns (Value);
208 axiom (forall s1: Value, s2: Value :: .isString(.concat(s1, s2)));
209 axiom (forall s1: Value, s2: Value, i: int ::
210     0 <= i && i < .stringSize(s1) ==> .toString(.concat(s1, s2))[i]
        == .toString(s1)[i]);
211 axiom (forall s1: Value, s2: Value, i: int ::
212     .stringSize(s1) <= i && i < .stringSize(s1) + .stringSize(s2)
        ==> .toString(.concat(s1, s2))[i] == .toString(s2)[i -
        .stringSize(s1)]);
213 axiom (forall s1: Value, s2: Value, i: int ::
214     i < 0 || i >= .stringSize(s1) + .stringSize(s2) ==>
        .toString(.concat(s1, s2))[i] == -1);
215 axiom (forall s1: Value, s2: Value :: .stringSize(.concat(s1, s2))
        == .stringSize(s1) + .stringSize(s2));
216
217 function .sum(x: Value, y: Value) returns (Value) {
218     .fromInt(.toInt(x) + .toInt(y))
219 }
220
221 function .sub(x: Value, y: Value) returns (Value) {
222     .fromInt(.toInt(x) - .toInt(y))
223 }
224
225 function .mult(x: Value, y: Value) returns (Value) {
226     .fromInt(.toInt(x) * .toInt(y))
227 }
228
```

```
229 function .div(x: Value, y: Value) returns (Value) {
230     .fromInt(.toInt(x) div .toInt(y))
231 }
232
233 function .rem(x: Value, y: Value) returns (Value) {
234     .fromInt(.toInt(x) mod .toInt(y))
235 }
236
237 function .min(x: Value) returns (Value) {
238     .fromInt(-.toInt(x))
239 }
240
241 function .neg(x: Value) returns (Value) {
242     .fromBool(!.toBool(x))
243 }
244
245 function .mkarray(v: Value, l: Value) returns (Value) {
246     .fromArray(.constArray(v), .toInt(l))
247 }
248
249 function .length(a: Value) returns (Value) {
250     .fromInt(.arraylen(a))
251 }
252
253 function .size(s: Value) returns (Value) {
254     .fromInt(.stringSize(s))
255 }
256
257 //REST calls
258
259 type RestMethod;
260
261 const unique .GET: RestMethod;
262 const unique .POST: RestMethod;
263 const unique .PUT: RestMethod;
264 const unique .DELETE: RestMethod;
265
266 function .restCall(RestMethod, Value, Value) returns (Value);
267
268 // Code to verify is added below
```

Appendix D

REST calls JavaScript Auxiliary Functions

```
1  /**
2   * Makes a synchronous rest call
3   * @param method A string with the http method
4   * @param uriTemplate A string with the complete uriTemplate of the
       call
5   * @param request An object with the fields: header, an object with
       the request
6   * headers; template, an object with the substitution of the
       variables in the
7   * uriTemplate; body (optional), the body of the request (any type)
8   * @returns An object representing the response of the rest call
       with fields:
9   * code, the integer status code of the response; header, an object
       with the
10  * response headers; body, the response body (any type)
11  */
12 function _synchRestCall(method, uriTemplate, request) {
13     var httpRequest = new XMLHttpRequest();
14     httpRequest.open(method, _expand(uriTemplate, request.template),
       false); // false -> synchronous call
15     httpRequest.setRequestHeader("Content-Type", "application/json");
16     request.header = _convertHeader(request.header);
17     for (var property in request.header)
18         if (request.header.hasOwnProperty(property))
19             httpRequest.setRequestHeader(property,
       request.header[property]);
20     httpRequest.send(JSON.stringify(request.body));
21     var response = {code: httpRequest.status, header: {}};
22
       httpRequest.getAllResponseHeaders().trim().split(/[\r\n]+/).forEach(line
       => {
```

```
23          var parts = line.split(': ');
24          var headerField = parts.shift();
25          var value = parts.join(': ');
26          response.header[headerField] = value;
27      });
28      response.body = JSON.parse(httpRequest.responseText);
29      return response;
30  }
31
32  /**
33   * Makes an asynchronous rest call
34   * @param method A string with the http method
35   * @param uriTemplate A string with the complete uriTemplate of the
       call
36   * @param request An object with the fields: header, an object with
       the request
37   * headers; template, an object with the substitution of the
       variables in the
38   * uriTemplate; body (optional), the body of the request (any type)
39   * @returns a Promise with the response
40   */
41  async function _asyncRestCall(method, uri, request) {
42      return new Promise(function (resolve) {
43          var httpRequest = new XMLHttpRequest();
44          httpRequest.open(method, _expand(uri, request.template),
      true);
45          httpRequest.setRequestHeader("Content-Type",
      "application/json");
46          request.header = _convertHeader(request.header);
47          for (var property in request.header)
48              if (request.header.hasOwnProperty(property))
49                  httpRequest.setRequestHeader(property,
      request.header[property]);
50          httpRequest.onload = () => {
51              var response = {code: httpRequest.status, header: {}};
52
      httpRequest.getAllResponseHeaders().trim().split(/[\r\n]+/).forEach(line
      => {
53                  var parts = line.split(': ');
54                  var headerField = parts.shift();
55                  var value = parts.join(': ');
56                  response.header[headerField] = value;
57              });
58              response.body = JSON.parse(httpRequest.responseText);
59              resolve(response);
60          }
```

```
61          httpRequest.send(JSON.stringify(request.body));
62      });
63  }
64
65  /**
66   * Expands the uriTemplate uri using the defined variables in
       parameters,
67   * according to RFC6570. Note that it is only implemented for the
       uriTemplates
68   * supported by HeadREST.
69   * @param uri A string with the uriTemplate to be expanded
70   * @param parameters An object with the pair (variable, value)
71   * @returns The expansion of uriTemplate uri using the values in
       parameters
72   */
73  function _expand(uri, parameters) {
74      var i = 0;
75      while ((i = uri.indexOf("{")) != -1) {
76          var j = uri.indexOf("}", i);
77          if (uri.charAt(i + 1) == "?")
78              var substitution = "?" + uri.substring(i + 2,
       j).split(",").filter(p => parameters.hasOwnProperty(p))
79                      .map(p => p + "=" +
       parameters[p]).map(_convertValue).join("&");
80          else
81              var substitution = uri.substring(i + 1,
       j).split(",").filter(p => parameters.hasOwnProperty(p))
82                      .map(p =>
       parameters[p]).map(_convertValue).join(",");
83          uri = uri.substring(0, i) + substitution + uri.substring(j +
       1);
84      }
85      return uri;
86  }
87
88  /**
89   * Converts a value into a string, and changes the necessary
       characters to
90   * percentage encoding so it can be a valid URI literal
91   * @param str The value to be converted
92   * @returns The value converted
93   */
94  function _convertValue(str) {
95      str = str + "";
96      var newStr = "";
97      for (var i = 0; i < str.length; i++) {
```

```
98          c = str.charAt(i);
99          if (c == "%" && /[0-9A-F]{2}/.exec(str.substring(i + 1, i +
        3)))
100             newStr += c;
101         else if (c.charCodeAt() < 32 || c == " " || c == '"' || c ==
        "'" || c == "%" || c == "<" || c == ">" ||
102                 c == "\\" || c == "^" || c == "`" || c == "{" || c
        == "|" || c == "}")
103             newStr += "%" + c.charCodeAt();
104         else
105             newStr += c;
106     }
107     return newStr;
108 }
109
110 /**
111  *
112  */
113 function _convertHeader(header) {
114     if (header === undefined)
115         return header;
116     if (header.hasOwnProperty("basicAuthorization"))
117         header.authorization = "Basic " +
        btoa(header.basicAuthorization);
118     return header;
119 }
```

Bibliography

[1] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for javascript. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 428–452, Berlin, Heidelberg, 2005. Springer-Verlag.

[2] Joop Aué, Maurício Aniche, Maikel Lobbezoo, and Arie van Deursen. 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-SEIP '18, pages 13–22, New York, NY, USA, 2018. ACM.

[3] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 364–387, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[5] M. Berzish, V. Ganesh, and Y. Zheng. Z3str3: A string solver with theory-aware heuristics. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 55–59, Oct 2017.

[6] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend : learn how to implement a DSL with Xtext and Xtend using easy-to-understand examples and best practices*. Packt Publishing, Birmingham, 2016.

[7] Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu, and David Langworthy. Semantic subtyping with an smt solver. *SIGPLAN Not.*, 45(9):105–116, September 2010.

[8] Api blueprint. Documentation. `https://apiblueprint.org/documentation/`, last accessed on 2018-11-19.

[9] Nuno Burnay, Antónia Lopes, and Vasco T. Vasconcelos. Safe(REST)Script. In *Actas do 11º Encontro Nacional de Informatica, INFORUM 2019*, Guimarães, Portugal, 2019.

[10] Avik Chaudhuri. Flow: Abstract interpretation of javascript for type checking and beyond. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 1–1, New York, NY, USA, 2016. ACM.

[11] J. R. V. Dantas, H. A. Lira, B. d. A. Muniz, T. M. Nunes, and P. P. M. Farias. Semantic web services discovery adopting serin. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, pages 387–394, Oct 2015.

[12] Dart. Dart programming language. `https://www.dartlang.org/`, last accessed on 2018-11-20.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] Arman Dezfuli-Arjomandi. Restyped. `https://github.com/rawrmaan/restyped`, last accessed on 2018-11-27.

[15] Arman Dezfuli-Arjomandi. Restyped axios. `https://github.com/rawrmaan/restyped-axios`, last accessed on 2018-11-27.

[16] Arman Dezfuli-Arjomandi. 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/`, last accessed on 2018-11-27.

[17] Eclipse. Xtext - language engineering made easy. `https://www.eclipse.org/Xtext/`, last accessed on 2018-12-11.

[18] Ralf S. Engelschall. Ecmascript 6 — new features: Overview & comparison, 2015. `http://es6-features.org/#ClassDefinition`, last accessed on 2018-11-20.

[19] Rest Api Example. Dummy sample rest api. `http://dummy.restapiexample.com/`, last accessed on 2018-12-19.

[20] Facebook. Graph api. `https://developers.facebook.com/docs/graph-api/`, last accessed on 2018-11-19.

[21] Fábio Ferreira. Automatic test generation for restful apis. Master's thesis, Faculdade de Ciências da Universidade de Lisboa, 2017.

[22] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, RFC Editor, June 2014.

[23] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, May 2002.

[24] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. AAI9980887.

[25] Robert W. Floyd. *Assigning Meanings to Programs*, pages 65–81. Springer Netherlands, Dordrecht, 1993.

[26] M. Fokaefs and E. Stroulia. Using wadl specifications to develop and maintain rest client applications. In *2015 IEEE International Conference on Web Services*, pages 81–88, June 2015.

[27] Tim Freeman and Frank Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. ACM.

[28] Zheng Gao, Christian Bird, and Earl T. Barr. To type or not to type: Quantifying detectable bugs in javascript. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 758–769, Piscataway, NJ, USA, 2017. IEEE Press.

[29] Yeting Ge and Leonardo de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 306–320, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[30] Google. Gmail api. `https://developers.google.com/gmail/api/`, last accessed on 2018-11-19.

[31] J. Gregorio, R. Fielding, M. Hadley, M. Nottingham, and D. Orchard. URI Template. RFC 6570, RFC Editor, March 2012.

[32] John Gruber. Markdown, 2004. `https://daringfireball.net/projects/markdown/`, last accessed on 2018-11-19.

[33] Said Hayani. Here are the most popular ways to make an http request in javascript, 2018. `https://medium.freecodecamp.org/here-is-the-most-pop`

ular-ways-to-make-an-http-request-in-javascript-954ce8c95aaa, last accessed on 2018-11-26.

[34] Susumu Hayashi. Logic of refinement types. In *Proceedings of the International Workshop on Types for Proofs and Programs*, TYPES '93, pages 108–126, Berlin, Heidelberg, 1994. Springer-Verlag.

[35] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[36] C.A.R. Hoare, Jayadev Misra, Gary Leavens, and Natarajan Shankar. The verified software initiative: A manifesto. *ACM Comput. Surv.*, 41, 10 2009.

[37] Tony Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, January 2003.

[38] Facebook Inc. Flow: A static type checker for javascript. https://flow.org/, last accessed on 2018-11-21.

[39] GitHub Inc. Projects | the state of the octoverse. https://octoverse.github.com/projects#languages, last accessed on 2018-12-26.

[40] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648, RFC Editor, October 2006.

[41] The jQuery Foundation. jquery. https://jquery.com/, last accessed on 2018-11-26.

[42] JSHint. Jshint, a static code analysis tool for javascript. https://jshint.com/about/, last accessed on 2018-11-23.

[43] K. Rustan M. Leino. Specification and verification of object-oriented software. In M. Broy, W. Sitou, and T. Hoare, editors, *Engineering Methods and Tools for Software Safety and Security*, pages 231–266. IOS Press, 2009.

[44] K. Rustan M. Leino and Rosemary Monahan. Dafny meets the verification benchmarks challenge. In Gary T. Leavens, Peter O'Hearn, and Sriram K. Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, pages 112–126, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[45] Rustan Leino. This is boogie 2. Microsoft Research, June 2008.

[46] Brian LeRoux. Wtfjs. dotJS 2012, https://www.dotconferences.com/2012/11/brian-leroux-wtfjs.

[47] Guy Levin. The rise of rest api, 2015. https://blog.restcase.com/the-rise-of-rest-api/, last accessed on 2018-11-16.

[48] Microsoft. Language server protocol. https://microsoft.github.io/language-server-protocol/, last accessed on 2018-12-11.

[49] Microsoft. Office 365 apis. https://docs.microsoft.com/en-us/previous-versions/office/office-365-api/, last accessed on 2018-11-19.

[50] Microsoft. Type script - javascript that scales. https://www.typescriptlang.org/index.html, last accessed on 2018-11-20.

[51] Martin Nordio, Cristiano Calcagno, and Carlo Alberto Furia. Javanni: A verifier for javascript. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, FASE'13, pages 231–234, Berlin, Heidelberg, 2013. Springer-Verlag.

[52] Ingy döt Net Oren Ben-Kiki, Clark Evans. Yaml ain't markup language version 1.2, 2009.

[53] Terence Parr. Antlr parser generator. https://www.antlr3.org/, last accessed on 2018-12-11.

[54] Cesare Pautasso. *RESTful Web Services: Principles, Patterns, Emerging Technologies*, pages 31–51. Springer New York, New York, NY, 2014.

[55] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. "big"' web services: Making the right architectural decision. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 805–814, New York, NY, USA, 2008. ACM.

[56] David J. Pearce and Lindsay Groves. Whiley: A platform for research in software verification. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering*, pages 238–248, Cham, 2013. Springer International Publishing.

[57] Sebastián Peyrott. A brief history of javascript, 2017. https://auth0.com/blog/a-brief-history-of-javascript/, last accessed on 2018-11-23.

[58] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[59] RAML. Welcome - raml. https://raml.org/, last accessed on 2018-11-19.

[60] J. Reschke. The 'Basic' HTTP Authentication Scheme. RFC 7617, RFC Editor, September 2015.

[61] Leonard Richardson, Mike Amundsen, and Sam Ruby. *RESTful Web APIs*. O'Reilly Media, Inc., 2013.

[62] Telmo Santos. Code generation for restful apis in headrest. Master's thesis, Faculdade de Ciências da Universidade de Lisboa, 2018.

[63] Marius Schulz. Typescript vs. flow, 2017. `https://blog.mariusschulz.com/2017/01/13/typescript-vs-flow`, last accessed on 2018-11-21.

[64] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252, RFC Editor, June 2014.

[65] SmartBear. Rest request methods. `https://www.soapui.org/learn/api/rest-request-method-verbs.html`, last accessed on 2018-12-13.

[66] Kwangwon Sun and Sukyoung Ryu. Analysis of javascript programs: Challenges and research trends. *ACM Comput. Surv.*, 50(4):59:1–59:34, August 2017.

[67] Swagger. What is openapi? `https://swagger.io/docs/specification/about/`, last accessed on 2018-11-20.

[68] Richard N. Taylor. The role of architectural styles in successful software ecosystems. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 2–4, New York, NY, USA, 2013. ACM.

[69] Brian Terlson. Ecmascript 2018 language specification. `https://www.ecma-international.org/ecma-262/9.0/index.html`, last accessed on 2018-06-26.

[70] Write the Docs. Api blueprint. `http://www.writethedocs.org/guide/api/apiblueprint/`, last accessed on 2018-11-19.

[71] Peter Thiemann. Towards a type system for analyzing javascript programs. In *Proceedings of the 14th European Conference on Programming Languages and Systems*, ESOP'05, pages 408–422, Berlin, Heidelberg, 2005. Springer-Verlag.

[72] Eric Tholomé. A well earned retirement for the soap search api, 2009. `http://googlecode.blogspot.com/2009/08/well-earned-retirement-for-soap-search.html`, last accessed on 2018-11-19.

[73] Mark Utting, David J. Pearce, and Lindsay Groves. Making whiley boogie! In Nadia Polikarpova and Steve Schneider, editors, *Integrated Formal Methods*, pages 69–84, Cham, 2017. Springer International Publishing.

[74] Vasco T. Vasconcelos, Francisco Martins, Antónia Lopes, and Nuno Burnay. *HeadREST: A Specification Language for RESTful APIs*, pages 428–434. Springer International Publishing, Cham, 2019.

[75] Vasco T. Vasconcelos, Francisco Martins, Antónia Lopes, Fábio Ferreira, Telmo Santos, and Nuno Burnay. Confident. `http://rss.di.fc.ul.pt/tools/confident/`, last accessed on 2018-11-29.

[76] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement types for typescript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 310–325, New York, NY, USA, 2016. ACM.

[77] W3C. Web services description language, 2001. `https://www.w3.org/TR/2001/NOTE-wsdl-20010315`, last accessed on 2018-11-20.

[78] W3C. Web application description language, 2009. `https://www.w3.org/Submission/wadl/`, last accessed on 2018-11-20.

[79] Bruce W. Weide, Murali Sitaraman, Heather K. Harton, Bruce Adcock, Paolo Bucci, Derek Bronish, Wayne D. Heym, Jason Kirschenbaum, and David Frazier. Incremental benchmarks for software verification tools and techniques. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, pages 84–98, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[80] Erik Wittern, Annie Ying, Yunhui Zheng, Jim A. Laredo, Julian Dolby, Christopher C. Young, and Aleksander A. Slominski. Opportunities in software engineering research for web api consumption. In *Proceedings of the 1st International Workshop on API Usage and Evolution*, WAPI '17, pages 7–10, Piscataway, NJ, USA, 2017. IEEE Press.

[81] Erik Wittern, Annie T. T. Ying, Yunhui Zheng, Julian Dolby, and Jim A. Laredo. Statically checking web api requests in javascript. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 244–254, Piscataway, NJ, USA, 2017. IEEE Press.

[82] Matt Zabriskie et al. axios. `https://github.com/axios/axios`, last accessed on 2018-11-27.