# API Parameter Recommendation Based on Documentation Analysis

by

Yuan Xi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

## Author's Declaration

## Abstract

Application Programming Interfaces (APIs) are widely used in today's software development, as they provide a easy and safe way to build more powerful applications with less code. However, learning how to use an API function correctly can sometimes be difficult. Software developers may spend a lot of time to learn a new library before they can become productive. When an unfamiliar API is to be used, they usually have to chase down documentation and code samples to figure out how to use the API correctly. This thesis proposes a new approach based on documentation analysis, helping developers learn to use APIs by recommending likely parameter candidates. Our approach analyzes the documentation information, extracts possible candidates from code context, and gives them as parameter suggestions.

To test the effectiveness of our approach, we process the documentation of 5 popular JavaScript libraries, and evaluate the approach on top 1,000 JavaScript projects from GitHub. We used 1,681 instances of API function calls for testing in total. On average, over 60% of the time the correct parameter is in the suggestion set generated by our approach.

## Acknowledgments

Firstly, I want to thank my supervisors–Professor Lin Tan, Professor Michael Godfrey, and Professor Meiyappan Nagappan for their patience and help to my research work. They work hard to provide a good research environment for every team member and give valuable suggestions when we meet problems. I sincerely appreciate the opportunity to work with them.

I want to thank for my parents and my girlfriend, who always love and support me. Without their support, I cannot finish my study successfully.

Thanks to all my friends I met in Waterloo. I enjoy my graduate study time with all of them.

This thesis is dedicated to the ones I love and the ones who love me.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

To improve software productivity, today's programs use Application Programming Interfaces (APIs) extensively, because APIs provide the reuse of the code and can release the project developers from some trivial and repetitive work. Thanks to the success of software communities and platforms such as Github, developers today can find a large number of libraries which provide various APIs with plenty of functionalities and are suitable for different development environments.

There are many advantages for software developers to use existing APIs from well-known libraries and frameworks rather than write their own code with the same functionalities. Using APIs can not only reduce repetitive work, but also make programs more robust, since API developers are always trying their best to improve the qualities and reliability of their APIs.

Although the utiliazation of APIs can lessen the workload of developers and provide various functionalities, it still may introduce new problems; for example, a library upgrade may cause backward compatibility issues [38], and API dependency chains could bring a lot of setup problems to developers before they can actually use the APIs [21]. One main challenge is that correctly and efficiently using APIs from unfamiliar libraries and frameworks is nontrivial. Because there are numerous APIs providing different functionalities, it is very common for developers to encounter unfamiliar APIs in their work. Working with complex APIs in an unfamiliar library presents many barriers: programmers have to choose not only the right method to call, but also the correct parameters for a method call in an API usage. When programmers want to use a new API function, they often need to carefully read the documentations and inspect code examples to learn the actual

Table 1.1: Statistics on API Function Declarations and Invocations

| Project | Parameterized Declaration | Non-parameterized Declaration | Parameterized Invocation | Non-parameterized Invocation |
|---|---|---|---|---|
| Eclipse 3.6.2 | 64% | 36% | 57% | 43% |
| Tomcat 7.0 | 49% | 51% | 60% | 40% |
| JBoss 5.0 | 58% | 42% | 60% | 40% |
| average | 57% | 43% | 59% | 41% |

meaning of each parameter, and search the entire context code to see which variables fit the arguments.

In order to help programmers use unfamiliar APIs better, a number of techniques have been proposed [44], [11], [45]. Among these, Buse et al. [11] and Zhong et al. [45] investigate finding examples or usage patterns to guide developers to correctly use the APIs. Another kind of approach is a code completion system or API suggestion system, which promptly provides developers with programming suggestions, such as which API functions to call and which expressions to use as parameters.

Most existing API suggestion techniques focus on telling developers which is the right API method to call [42], [19]. However, previous research has also emphasized the importance of helping developers choose the right parameters for an API method call [10], [27].

Zhang et al. [44] investigate the statistics of API function declarations and invocations in the code of Eclipse 3.6.2 [1], Tomcat 7.0 [5], and JBoss 5.0 [3]. Table 1.1 displays their findings. According to Table 1.1, for both API declarations and API invocations, API functions with one or more parameters are more common than that without any parameters. Thus, helping developers choose the right API functions is not enough, and recommending the right parameters can be another non-trivial topic.

Nowadays libraries usually provide documentation for programmers to learn how to use them, and these are usually highly reliable because they are provided by the developers of the libraries, who fully understand how to use the APIs correctly. Previous work shows that API documentations have a significant impact on API usability [14]. When having to use an unfamiliar library, the first thing for most programmers to do is to search for the corresponding documentations and read them.

In this paper, we propose and evaluate an approach to let computers learn the documentation and suggest the correct arguments when using a certain API. It will help developers save a lot of time and reduce bugs when they need to use some unfamiliar APIs.

## 1.1 Research Contributions

In this thesis, I make the following contributions:

- We collect two data sets: one data set of JavaScript API documentation and one of real-world JavaScript projects. The API documentation data set consists of five popular JavaScript third-party libraries, and the project data set consists of top 1,000 JavaScript projects based on their star ratings. These two data sets can be used for future research on API suggestions and documentation analysis.
- We propose an approach to generate parameter suggestions for API function calls based on API documentation analysis. The approach does not require too many code examples, and can be generalized to other programming languages.
- We conduct experiments evaluating the performance of our API parameter suggestion approach.

## 1.2 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 and Chapter 3 discuss the background and related work respectively. Chapter 4 describes our approach for choosing the right parameters for an API method. Chapter 5 shows the setup of experiments to evaluate the proposed approaches. We show and analyze experimental results in Chapter 6. In Chapter 7, we present the threats to validity in our work. Chapter 8 summarizes our work, and sketches possible future research in this area.

# Chapter 2

# Background

This section provides the background of statically typed languages and dynamically typed languages, API documentation, the API library used in this research, and the overview of the research problem.

## 2.1 Statically Typed Languages and Dynamically Typed Languages

### 2.1.1 Statically Typed Languages

A language is statically typed if the type of a variable is known at compile time, and will not change after being declared. For some languages this means that you as the programmer must specify what type each variable is (e.g., Java, C); other languages offer some form of type inference, the capability of the type system to deduce the type of a variable (e.g.: OCaml, Haskell, Scala, Kotlin).

A more formal way to define statically typed language is that a programming language is considered as statically typed if it does type checking at compile-time. This process verifies the type safety of a program based on analysis of a program's source code. If a program passes a static type checker, then the program is guaranteed to satisfy some set of type safety properties for all possible inputs [43].

The main advantage here is that many kinds of checking can be done by the compiler, and therefore a lot of trivial bugs can be caught at a very early stage [6]. However, many languages with static type checking provide a way to bypass the type checker. Some languages allow programmers to choose between static and dynamic type safety. For example, C# distinguishes between statically-typed and dynamically-typed variables. Uses of the former are checked statically, whereas uses of the latter are checked dynamically. Other languages allow writing code that is not type-safe; for example, in C, programmers can freely cast a value between any two types that have the same size, effectively subverting the type concept.

Statically typed languages come out earlier than dynamically typed languages. The first statically typed language is Fortran [8], which is developed by IBM in 1957, and it is also known as the first high-level programming language.

### 2.1.2 Dynamically Typed Languages

A language is dynamically typed if the type is associated with run-time values, and not named variables/fields/etc. This means that the type of a variable may change after being declared. For example, JavaScript is a dynamically typed language, and in JavaScript, you can assign a numeric value to a variable when being declared, and assign a string value to it later. Typically you as a programmer do not have to specify types every. Here are some famous dynamically typed languages: JavaScript, Perl, Ruby, Python.

A dynamically typed language performs type checking at runtime. Implementations of dynamically type-checked languages generally associate each runtime object with a type tag (i.e., a reference to a type) containing its type information [43]. Most type-safe languages include some form of dynamic type checking, even if they also have a static type checker. This is because that many useful features or properties are difficult or impossible to verify statically. Most scripting languages have this feature as there is no compiler to do static type-checking anyway, but you may find yourself searching for a bug that is due to the interpreter misinterpreting the type of a variable. Luckily, scripts tend to be small so bugs have not so many places to hide.

By definition, dynamic type checking may cause a program to fail at runtime. In some programming languages, it is possible to anticipate and recover from these failures. In others, type-checking errors are considered fatal.

Most dynamically typed languages do allow you to provide type information, but do not require it. One language that is currently being developed, Rascal, takes a hybrid approach allowing dynamic typing within functions but enforcing static typing for the function signature.

The first dynamically typed language is Lisp, which was invented by John McCarthy in 1958 [20]. Lisp is the second-oldest high-level programming language in widespread use today, and Fortran is the only older one, by one year in 1957.

## 2.2 API Documentation

Since APIs are designed to be consumed, it is important to make sure that the clients, or consumers, are able to quickly learn an API and understand what they can do with it. Most of consumers will never read the source code of the API libraries they use. It is always too tidious for them to know how to implement it. Instead, they want to understand how to use the API quickly and efficiently, which is where API documentation comes into the picture.

API documentation is a technical content deliverable, containing instructions about how to effectively use and integrate with an API. It is a concise reference manual containing all the information required to work with the API, with details about the functions, classes, return types, arguments and more, supported by tutorials and examples.

Figure 2.1 gives an example of an online API documentation.

## 2.3 Lodash: A Modern JavaScript Utility Library

Lodash is a JavaScript library that provides utility functions for common programming tasks. It evolves from Underscore.js and now receives maintenance from the original contributors to Underscore.js.

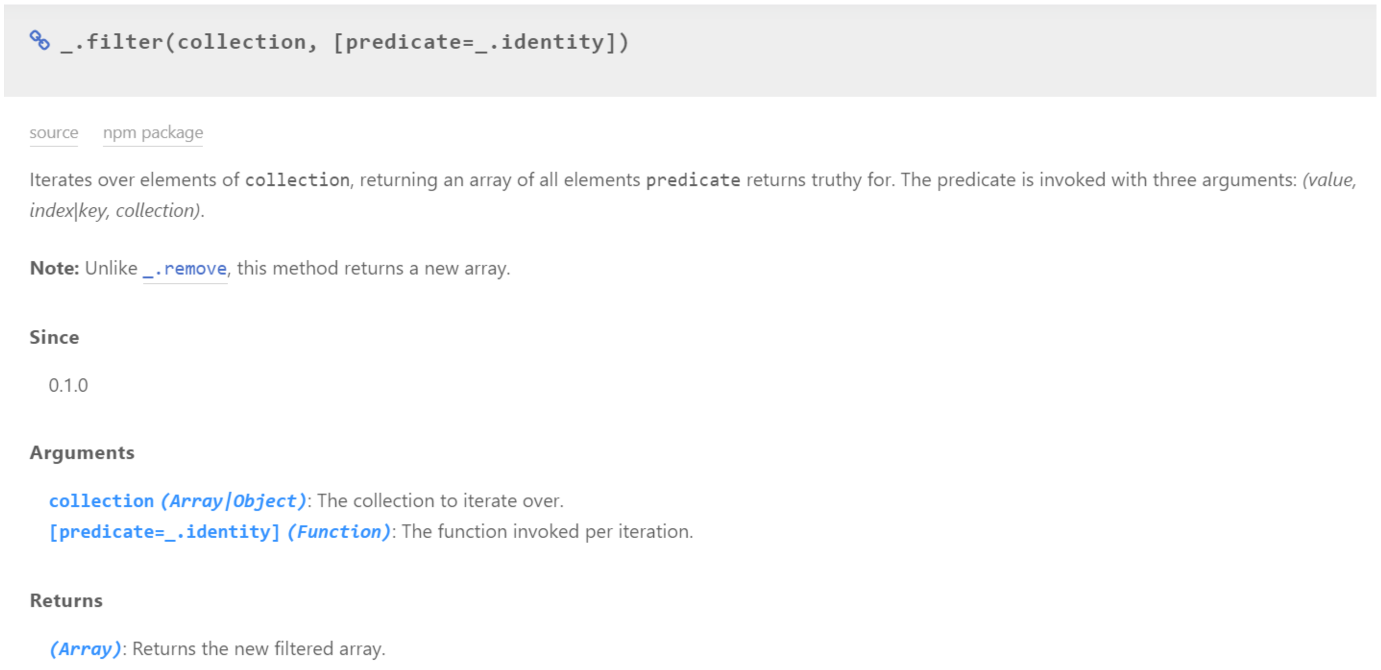
Lodash is one of the most popular JavaScript libraries now. Its number of weekly downloads is around 15,000,000 on NPM platform.

Table 2.1 shows some basic information about Lodash.

## 2.4 API Parameter Suggestion

This section illustrates API parameter suggestion problem with a use case example.

Imagine that a developer, Pat is writing a JavaScript project, and learning to use the JavaScript library Lodash. Pat is currently working on a new feature for their website that



```
_.filter(collection, [predicate=_.identity])
```

source npm package

Iterates over elements of `collection`, returning an array of all elements `predicate` returns truthy for. The predicate is invoked with three arguments: *(value, index|key, collection)*.

**Note:** Unlike `_.remove`, this method returns a new array.

**Since**

0.1.0

**Arguments**

`collection` *(Array|Object)*: The collection to iterate over.
`[predicate=_.identity]` *(Function)*: The function invoked per iteration.

**Returns**

*(Array)*: Returns the new filtered array.

Figure 2.1: Lodash Online Documentation for API Function "filter"

Table 2.1: Statistics on API Function Declarations and Invocations

| Name | Lodash |
|---|---|
| Original author | John-David Dalton |
| Initial release | April 23, 2012 |
| Current version | 4.17.10 |
| Written in | JavaScript |
| License | MIT |
| Website | lodash.com |
| Lines of codes | 17,105 |
| Number of API elements | 287 |

will display the usernames of all active users. Pat currently has an array of users, where each user is an object with attributes "name" and "gender".

```
var users = [
    { ‘name’: ‘Jack’, ‘gender’: ‘Male’ },
    { ‘name’: ‘Susan’, ‘gender’: ‘Female’ },
    { ‘name’: ‘Bob’, ‘gender’: ‘Male’ },
    { ‘name’: ‘Alex’, ‘gender’: ‘Non-binary’}
];
```

Now Pat wants to only keep all the male users from the list, and Pat knows that the API function `filter` Lodash library will be helpful, so Pat starts writing:

```
var male_users = filter(
```

However, Pat forgets what are the parameters for the function `filter`. Therefore, Pat opens a web browser and begins to search for the correct way o filter data using Lodash. Pat searches for ”Lodash” in Google, opens the home page of texttlodash.com, locates the page containing API documentation, and finds hundreds of Lodash API functions. It takes a few minutes for Pat to locate the documentation for the API method `filter` (shown as Fig 2.1). Pat learns from the documentation that the filter method takes two parameters: a collection and a predicate. The collection can be either an array or an object, and the predicate is a function used to filter the array. Having found the needed information, Pat returns from the journey back to editing the JavaScript document, where Pat completes the method call.

Consider how the above use case would be different if Pat had a plugin to automatically read the Lodash documentation and provide intelligent parameter suggestions:

1. Pat writes `var male_users = filter(`
2. The variable `users` pops up as a suggestion for the first parameter
3. Pat chooses `users` and press ENTER
4. The function `isMale(user)` pops up as a suggestion for the second parameter
5. Pat choose `isMale(user)` and presses ENTER

With only 5 steps, an unfamiliar API function call is completed, and John never loses focus on his current task.

The following chapters will introduce some related works and show how our documentation-based API parameter suggestions work.

# Chapter 3

# Related Work

## 3.1 API Usage Patterns

API usage patterns are patterns that guide consumers how to use APIs to achieve their goals. They show the sequences of API function calls for developers. For example, if a developer want to write something into a file, an API usage pattern will tell him/her which API function should be called first to open the file, then which API function should be called to write text content, and which function should be invoked finally to close the file.

In 2005, Holmes et al. [18] proposed Strathcona that helps developers by automatically locating source code examples that are relevant to their work. In 2009, Zhong et al. [45] proposed an approach called MAPO that mines API usage patterns and uses mined patterns as an index for recommending associated code snippets to aid programming. Buse et al. [11] tried to build API usage patterns by synthesizing code examples from documentation and online question and answering forums like StackOverflow. Saied et al. [31] proposed a technique for mining multi-level API usage patterns to exhibit the co-usage relationships between methods of the API of interest across interfering usage scenarios. In 2003, Montandon et al. [22] described a platform that instruments the standard Java-based API documentation format with concrete examples of usage and can also be applied to Android APIs. Nguyen et al. [25] proposed GrouMiner, a graph-based approach that represents usage patterns as graphs for mining. Wang et al. [39] presented UP-Miner to mine succinct and high-coverage patterns from source code by clustering API call sequences. In 2014, a new method called Baker was proposed to link up-to-date source code examples to API documentation [33]. In 2016, Fowkes et al. [16] designed a parameter-free probabilistic

API mining algorithm, PAM, that uses a novel probabilistic model based on a set of API patterns and a set of probabilities to infer the most interesting API patterns.

## 3.2 Software Text Analysis

Documentation contains a lot of useful information. Forward et al. [15] pointed out that documentation content is relevant and important in their research. They also found that good software documentation technologies should be more aware of professionals' requirements, opposed to blindly enforce documentation formats or tools.

A lot of useful technologies have been proposed to analyse software documentation. Rubio-González et al. [30] used static program analysis to examine mismatches between documented and actual error codes. Many techniques have been proposed to automatically analyze comments and detect inconsistencies between comments and source code [35], [36], [37]. Dalip et al. [13] proposed an approach that automatically estimates the quality of the documents in the digital library. *DMOSS* [12] is a toolkit that can systematically assess the quality of non source code text found in software packages. Schiller et al. [32] investigated code contracts of 90 C# open-source projects. Wong et al. [40] designed an approach called *DASE* to improve the performance of symbolic execution for automatic test generation and bug detection. *DASE* can automatically extract input constraints from documents of a software project, and use these constraints to find out core execution paths in the program. Blasi et al. [9] presents an approach, Jdoctor, that translates Javadoc comments into executable procedure specifications written as Java expressions. Yang et al. [41] designed D2Spec for extracting web API specifications from the documentation pages based on machine learning.

## 3.3 Code Completion System

Enhancing current completion systems to work more effectively with large APIs have been investigated in previous studies [24], [10]. These studies made use of database of API usage recommendation, type hierarchy, context filtering and API methods functional roles for improving the performance of API method call completion. Calcite is an Eclipse plugin that helps developers instantiate classes by adding Java API suggestions to the code completion menu [23]. Omar et al. [26] designed Graphite, an active code completion tool allowing Java library developers to introduce interactive and highly-specialized code generation interfaces directly into the editor. Robbes et al. [29] proposed an approach to improve code completion with program history. Ginzberg et al. [17] proposed an automatic code completion approach by developing an LSTM model. Asaduzzaman et al. [7] proposed a context sensitive code completion technique that uses, in addition to the aforementioned information, the context of the target method call. Recently, Pythia, a novel AI-assisted code completion system developed by Microsoft, has been available as part of Intellicode extension in Visual Studio Code IDE [34].

## 3.4 API Usage Recommendation

Most of the current API recommendation techniques focus on API usage recommendation, which tells the consumers which is the right API method to call [28], [42], [19]. These techniques explore the relationship among the API functions and the context of real world source code that invokes the APIs and make use of these relations to give recommendation. Rahman et al. [28] proposed RACK that recommends a list of relevant APIs for a natural language query for code search by exploiting keyword-API associations from the crowd-sourced knowledge of StackOverflow. LibraryGuru [42] recommends suitable Android APIs for given functionality descriptions. Ma et al. [19] proposed an approach *ServRel* to recommend relevant Web APIs based on the proposed service cooperative network for a target Web API.

## 3.5 API Parameter Recommendation

Another kind of techniques are used for helping developers choose the right parameters for an API method call. Previous researches show that compared with suggesting the right API method to call, software developers like a tool that can recommend the right parameter much more [10], [27]. However, there are only few studies on it. Zhang et al. [44] proposed an approach, *Precise* for recommending API parameters by mining existing code bases, and 53% of the recommendations are exactly the same as the actual parameters. However, *Precise* can only work for Java projects and requires existing code examples that use the APIs.

As discussed in Section 2.1, statically typed languages like Java and dynamically typed languages like JavaScript are different. *Precise* makes use of the property of statically typed languages that the type of a variable is fixed, and will not change after being declared. After getting the abstract syntax tree (AST) of the program, *Precise* can easily know the type of each API parameter and each variable. However, for dynamically typed languages, type information is not contained in the AST, so *Precise* cannot work for dynamically typed languages. In addition, *Precise* is based on machine learning technologies, which need lots of usage examples as training data. It will be difficult to extend *Precise* for the API libraries that are less popular or do not have enough code examples.

# Chapter 4

# Approaches

In this chapter, we describe our approach for giving suggestions for API parameters.

## 4.1 Overview

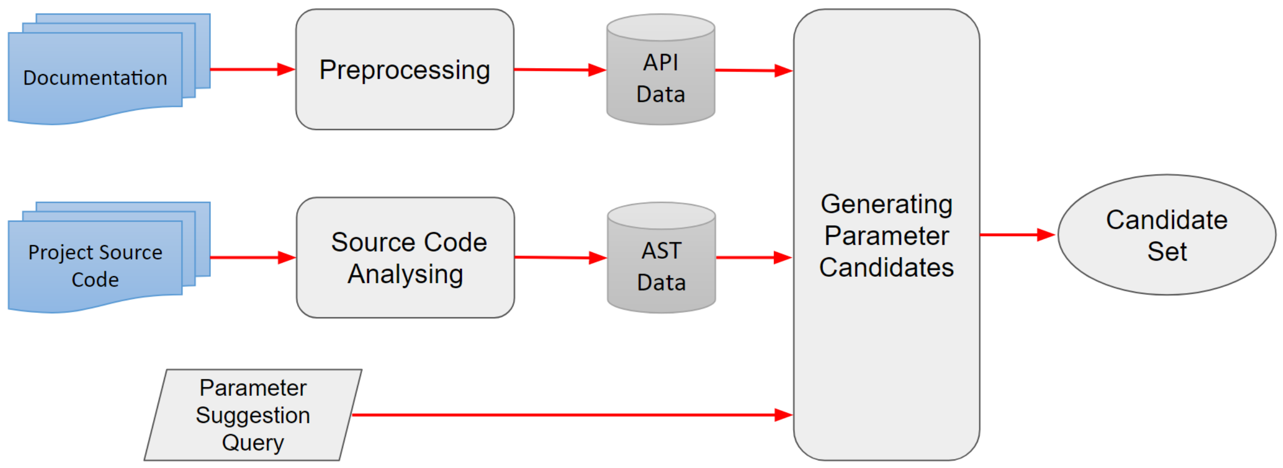


Figure 4.1: Workflow of API Parameter Recommendation

Figure 4.1 shows the workflow of the entire process. Our approach can be divided into three parts: preprocessing, analysing the project source code, and generating parameter candidates. First, we collect and preprocess the library documents to build an API database that stores the essential information of each API functions, such as method name, number of arguments, etc. Once the API database is set up, we need to analyse the source code for each parameter suggestion query. In this part, abstract syntax tree (AST) data will be generated for each source code file. The final step is to use the API data and AST data to generate parameter candidates for the suggestion query. This part involves two different approaches depending on whether the parameter in the query is non-function-type or function-type.

Section 4.2 describes the preprocessing part. Section 4.3 shows the detail of how to analyse the source code for each project that uses the APIs. Section 4.4 and Section 4.5 explains the candidate generating approaches for non-function-type parameter and function-type parameter respectively.

## 4.2 Preprocessing

Preprocessing is the first part of our approach. Figure 4.2 and Figure 4.3 demonstrate the workflow of the preprocessing step.

From the library documentation, we can get lots of information about an API element, including its arguments, what it returns, some description about the API, and some example code telling developers how to use the API in practice. This information is always provided, and we can use a web crawler to collect them from the online documents. We note that this collection step only needs to be performed once per library if we build these information into a database, then the documentation data can be reused for each API parameter suggestion query. Figure 4.2 shows how we build the database and what is contained in our database.

We study documents of many famous JavaScript libraries in our research. Although these documents are in different format, most of them are all well-structured. Some examples are shown as Figure 4.4 and Figure 4.5. Thus, although we have to write a different web crawler for each library, the task is relatively straightforward.

As shown in Figure 4.2, for each API element, we only store four basic kinds of information in our databse: argument, description, return, and example. Most documents such as Figure 4.4 and Figure 4.5 will provide this information. So this step can be used on most of popular libraries today.

Once we set up the database, we go to step 2, which is shown as Figure 4.3.

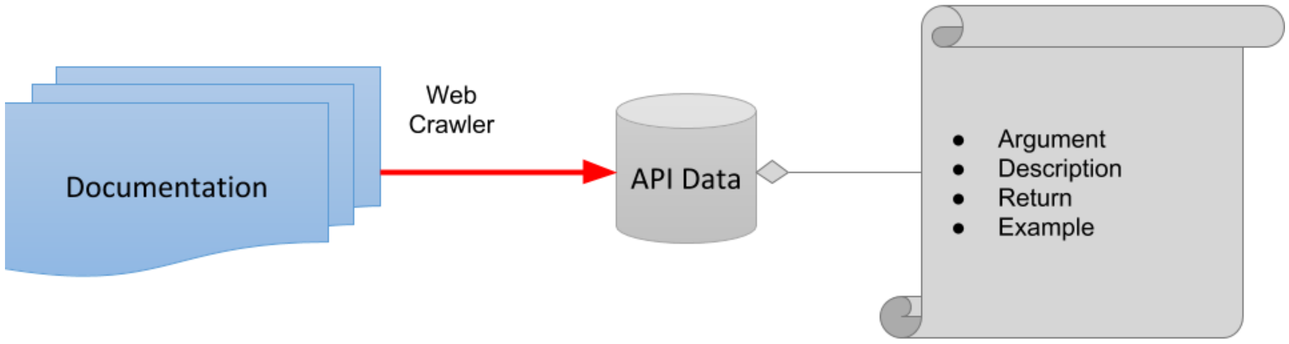


Figure 4.2: Preprocess: Step 1

Since we store argument information in our database, we can easily get the name and the type of the argument. In JavaScript, there are 6 basic types: Numeric, String, Boolean, Object, Array, and Function.

For our approach, we deal with Function-type arguments differently from other types, so here for each argument in each API function, we have to decide whether it should be function-type or non-function-type (Numeric, String, Boolean, Object, or Array) according to the documentation.

If the target argument is a non-function-type argument, we should use the approach for non-function-type parameter in Section 4.3 to give the suggestions.

If the target argument is a function-type argument, we should first take the name of the target argument and the description part of the API, and analyze the description part to see if there are any descriptions about how many parameters this function (the function-type argument) should have. We study a lot of popular JavaScript libraries, and find most of the documents will provide this kind of information although the ways of

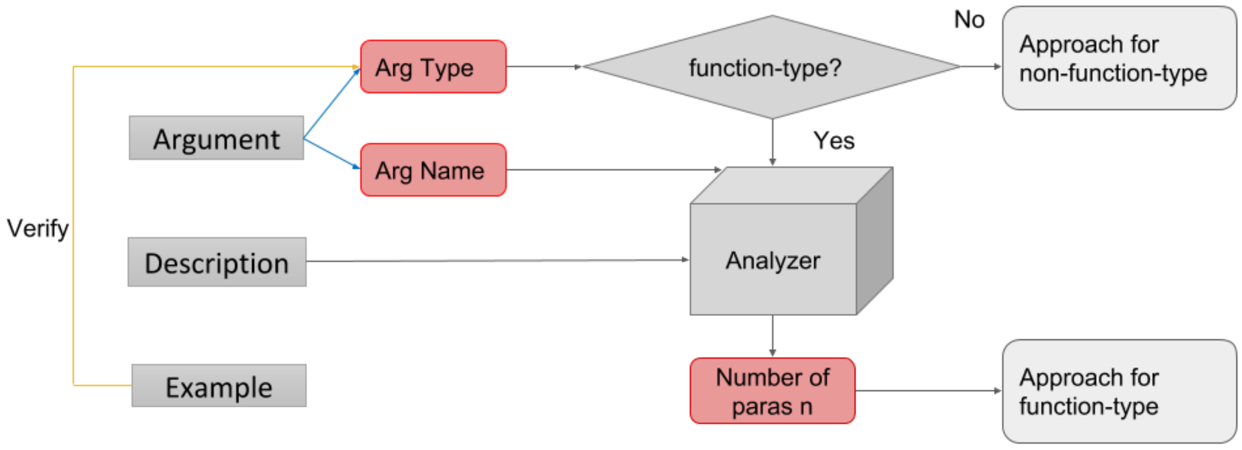
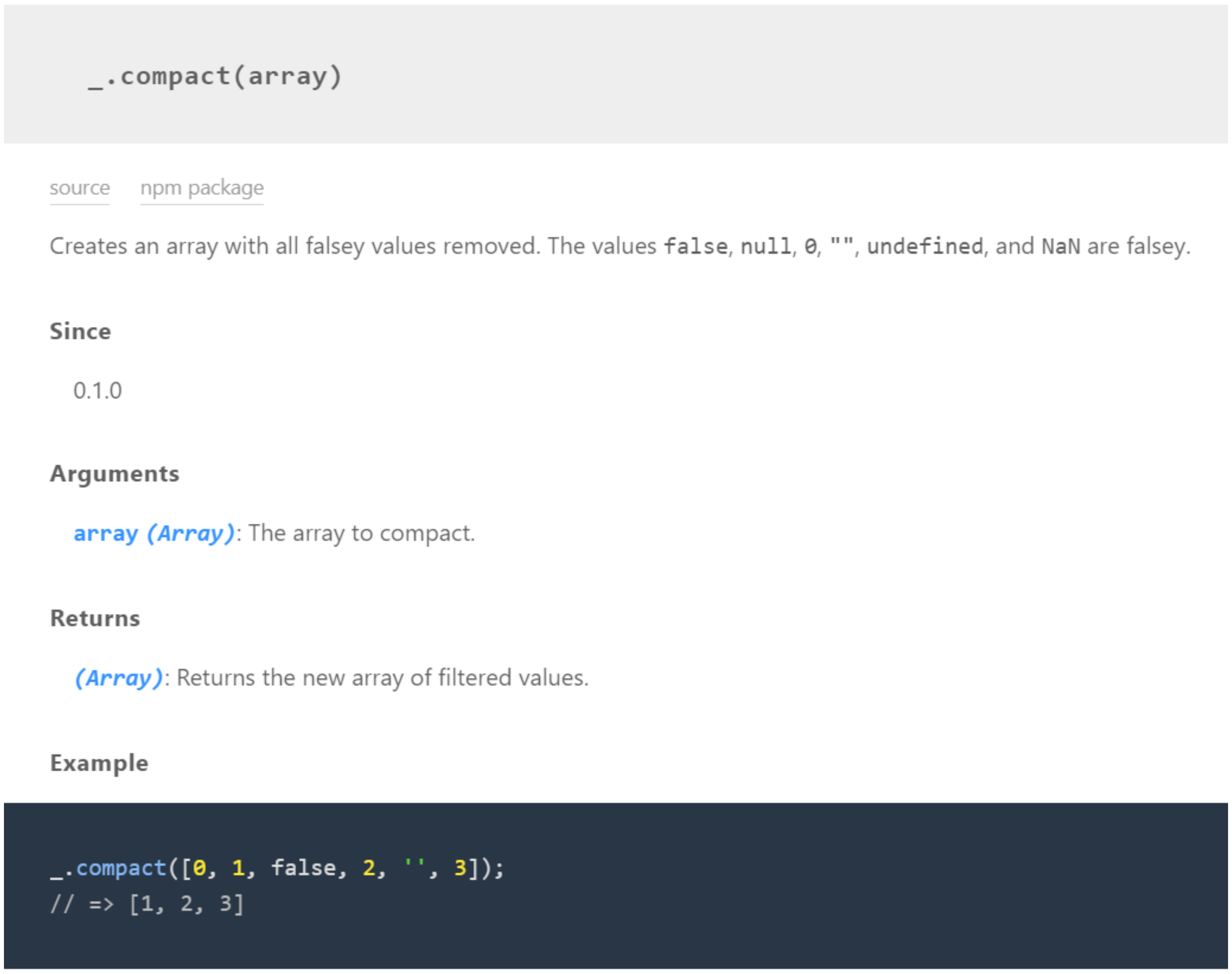


Figure 4.3: Preprocess: Step 2

their expressions may be different. Figure 4.6 shows the related description in Lodash document, and Figure 4.7 shows the related decription in AngularJS document. So for different libraries, we have to build a different analyzer to get how many parameters the function-type argument should have. Once we get this number of parameters, $n$, we use the approach for function-type parameter in Section 4.5 to give the suggestions.

We use the API documentation in Figure 4.6 as an example to illustrate the preprocess step. Using a web crawler, we can easily collect the arguments, description, return, and example of the API function *assignInWith*. This function has three arguments: *object*, *sources*, and *customizer*. The types of first two arguments are both Object, and the third argument is a Function. So the third argument is a function-type parameter, and the first two arguments are non-function-type. We check all the sentences in the description part, and the last two sentences both mention *customizer*. From the last sentence, using simple string match techniques, we get to know that *customizer* is a function that should have five arguments, so the number of parameters, $n$, of it will be 5. All the information of *assignInWith* will be stored in the API database.


```
_.compact(array)
```

source npm package

Creates an array with all falsey values removed. The values `false`, `null`, `0`, `""`, `undefined`, and `NaN` are falsey.

**Since**

0.1.0

**Arguments**

`array (Array)`: The array to compact.

**Returns**

`(Array)`: Returns the new array of filtered values.

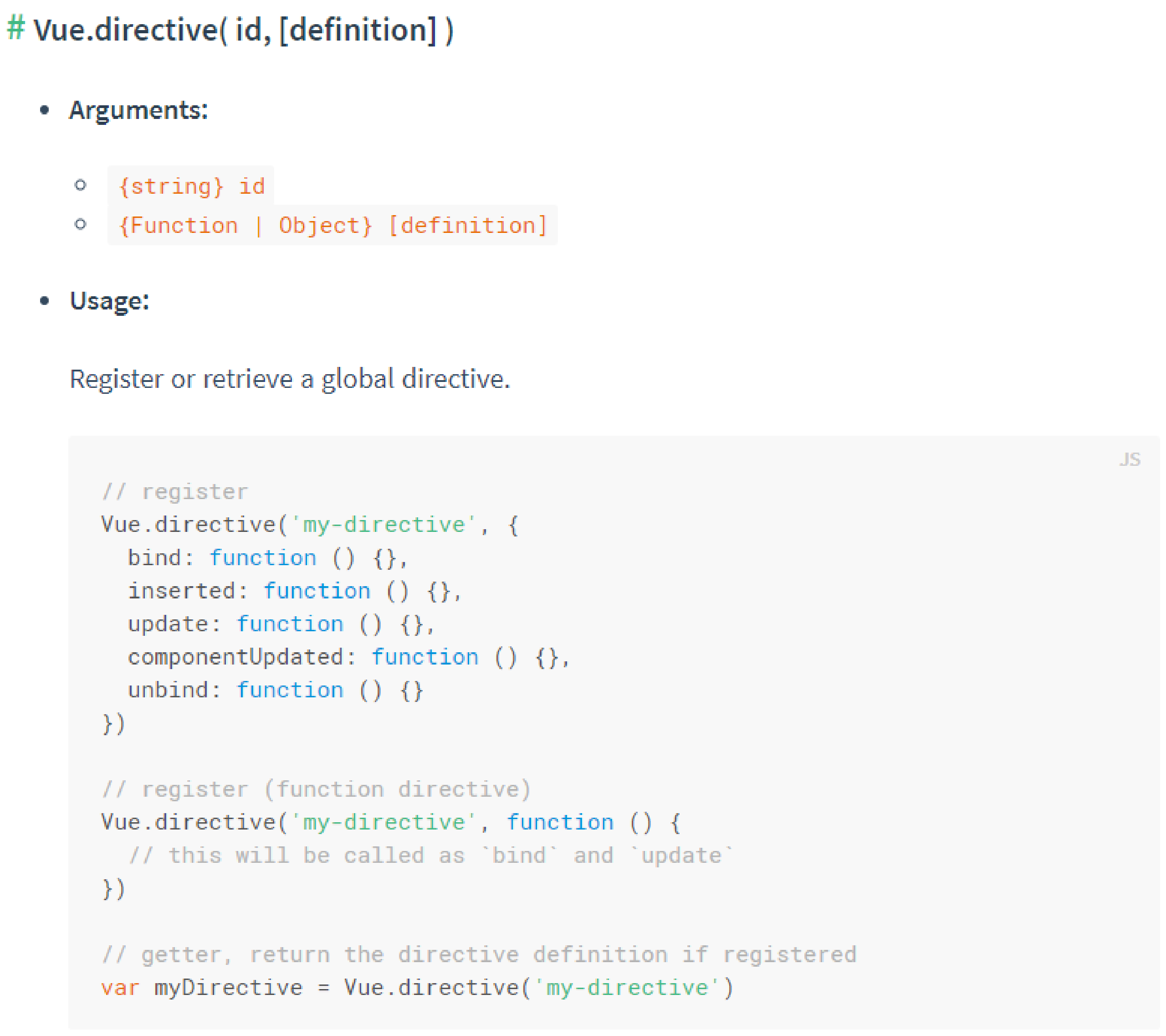
**Example**

```
_.compact([0, 1, false, 2, '', 3]);
// => [1, 2, 3]
```

Figure 4.4: Lodash Documentation Example

## 4.3 Analysing the Project Source Code

After setting up the database for the library, we use the information to give the API parameter suggestions. Given a project that invoke an API, we have to analyze the source code. Figure 4.8 shows the approach for analysing the source code of project which uses the APIs.


# Vue.directive( id, [definition] )

- **Arguments:**
  - `{string} id`
  - `{Function | Object} [definition]`
- **Usage:**

  Register or retrieve a global directive.

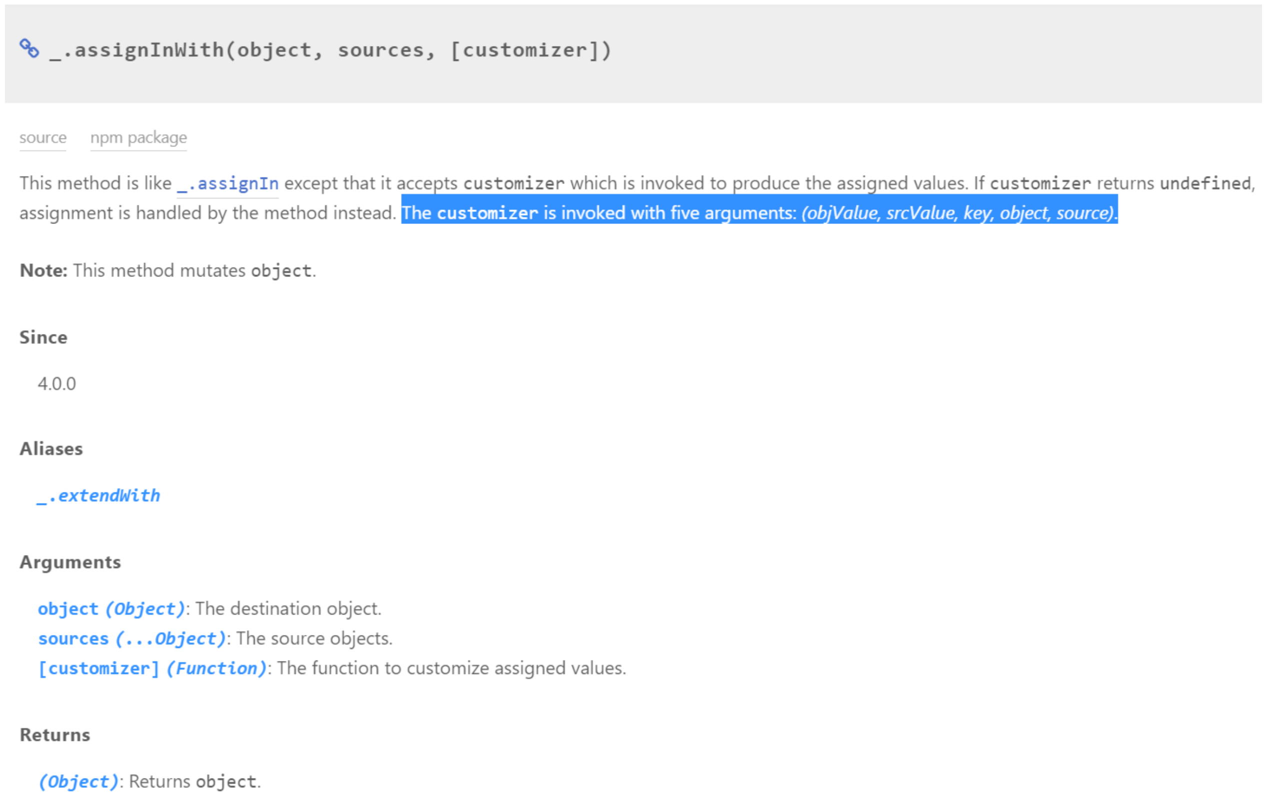
```
JS
// register
Vue.directive('my-directive', {
  bind: function () {},
  inserted: function () {},
  update: function () {},
  componentUpdated: function () {},
  unbind: function () {}
})

// register (function directive)
Vue.directive('my-directive', function () {
  // this will be called as `bind` and `update`
})

// getter, return the directive definition if registered
var myDirective = Vue.directive('my-directive')
```

Figure 4.5: Vue.js Documentation Example

For our approach, we only take the source code in the same file with the API call. We first need to use a JavaScript parser to parse the source code and get the abstract syntax tree (AST) of it. Once we have the AST, we use an AST parser to collect some



```
_.assignInWith(object, sources, [customizer])
```

source npm package

This method is like `_.assignIn` except that it accepts `customizer` which is invoked to produce the assigned values. If `customizer` returns `undefined`, assignment is handled by the method instead. The `customizer` is invoked with five arguments: *(objValue, srcValue, key, object, source)*.

**Note:** This method mutates `object`.

**Since**

4.0.0

**Aliases**

*`_.extendWith`*

**Arguments**

`object` *`(Object)`*: The destination object.
`sources` *`(...Object)`*: The source objects.
`[customizer]` *`(Function)`*: The function to customize assigned values.

**Returns**

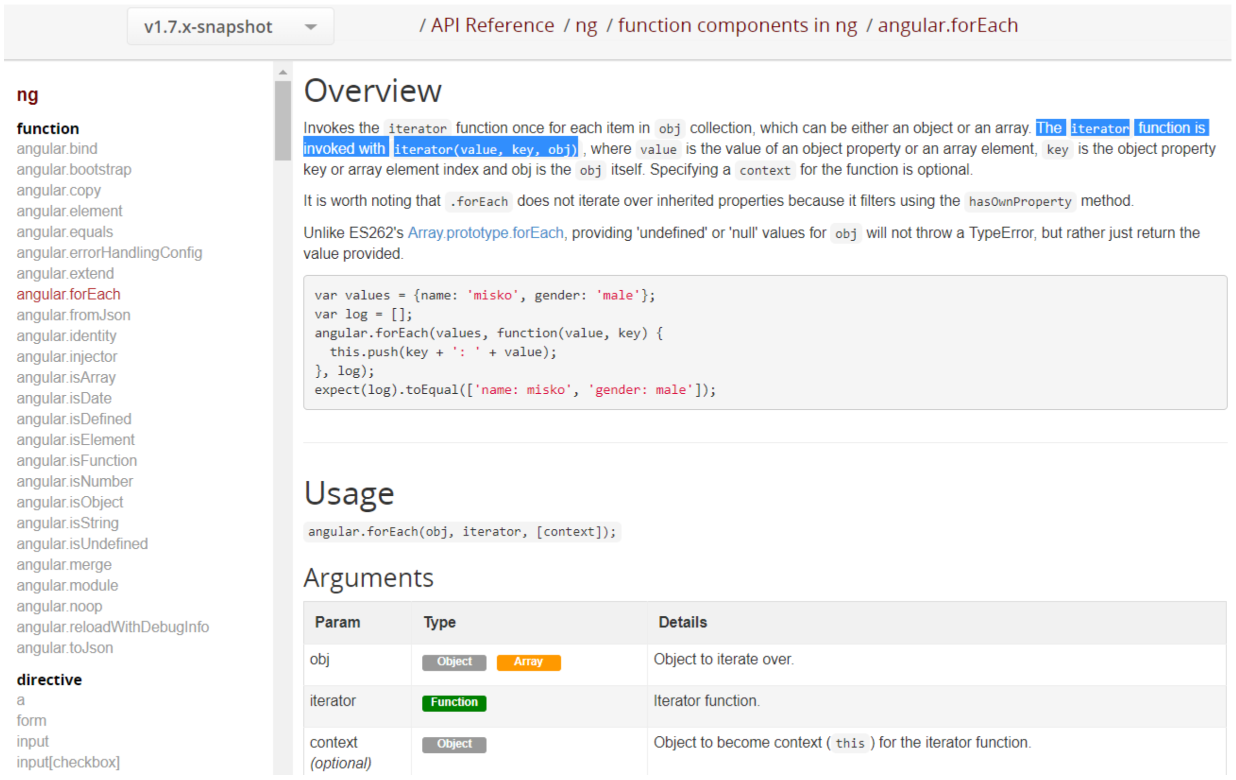
*`(Object)`*: Returns `object`.

Figure 4.6: Number of Arguments for Function-type Parameter in Lodash Documentation

important information, including all the APIs that are imported in the file, a set of variables that represent Functions, a set of variables that represent Numerics, a set of variables that represent Booleans, a set of variables that represent Strings, a set of variables that represent Objects, and a set of variables that represent Arrays. We can use these information and the information from the database in Section 4.2 to give the API parameter suggestions.

## 4.4 Approach for Non-function-type Parameter

Figure 4.9 demonstrates the approach for suggesting candidates for a non-function-type parameter.



Overview

Invokes the `iterator` function once for each item in `obj` collection, which can be either an object or an array. The `iterator` function is invoked with `iterator(value, key, obj)`, where `value` is the value of an object property or an array element, `key` is the object property key or array element index and obj is the `obj` itself. Specifying a `context` for the function is optional.

It is worth noting that `.forEach` does not iterate over inherited properties because it filters using the `hasOwnProperty` method.

Unlike ES262's Array.prototype.forEach, providing 'undefined' or 'null' values for `obj` will not throw a TypeError, but rather just return the value provided.

```
var values = {name: 'misko', gender: 'male'};
var log = [];
angular.forEach(values, function(value, key) {
  this.push(key + ': ' + value);
}, log);
expect(log).toEqual(['name: misko', 'gender: male']);
```

Usage

```
angular.forEach(obj, iterator, [context]);
```

Arguments

| Param | Type | Details |
|---|---|---|
| obj | Object Array | Object to iterate over. |
| iterator | Function | Iterator function. |
| context *(optional)* | Object | Object to become context ( `this` ) for the iterator function. |

Figure 4.7: Number of Arguments for Function-type Parameter in AngularJS Documentation

We can easily look up the type of the parameter we want to recommend in the database in Section 4.2, and we call the type $X$, which can be either one of Numeric, String, Boolean, Object, or Array. Because we classify the variables according to their types in Section 4.3, we can pick out a set of variable that represent type $X$, and these variables can be part of the candidate set.

We also need to check all the imported APIs in Section 4.2. We look up their return types in the database. If their return values are type $X$, then these API calls can also be part of the candidate set.

For each of the imported API, we should calculate the similarity between the target API and it. If we find a similar API to the target API, we can use the actual value of the similar argument as a suggestion for the target parameter. For example, we want to recommend a value for the first parameter of an API function, *pullAllWith*, and we check

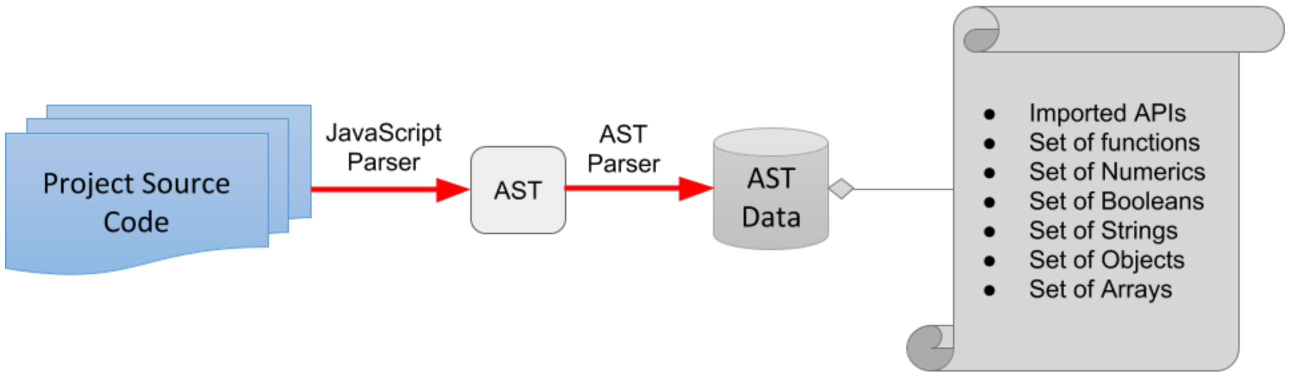


Figure 4.8: Analyse the Project Source Code

all the imported APIs and find one API function called *pullAllBy*. We look them up in the database, and find that not only their function names are similar, but also the first arguments of them have the same name and the same type, so we think these two APIs are similar, and we can use the actual value of the first parameter of *pullAllBy* as a suggestion for the first argument of *pullAllWith*. In this way, similar arguments from similar API functions can be used as part of the candidate set.

In addition, we use constants of type $X$ as our default options so that when our approach cannot find any variables that satisfy the requirements, we can still give some suggestions.

We use the following code snippet as an example to illustrate how the approach works.

```
var arr = [1];
var other = concat(arr, 2, [3], [[4]], ‘’);
var num = 5;
compact( ? );
```

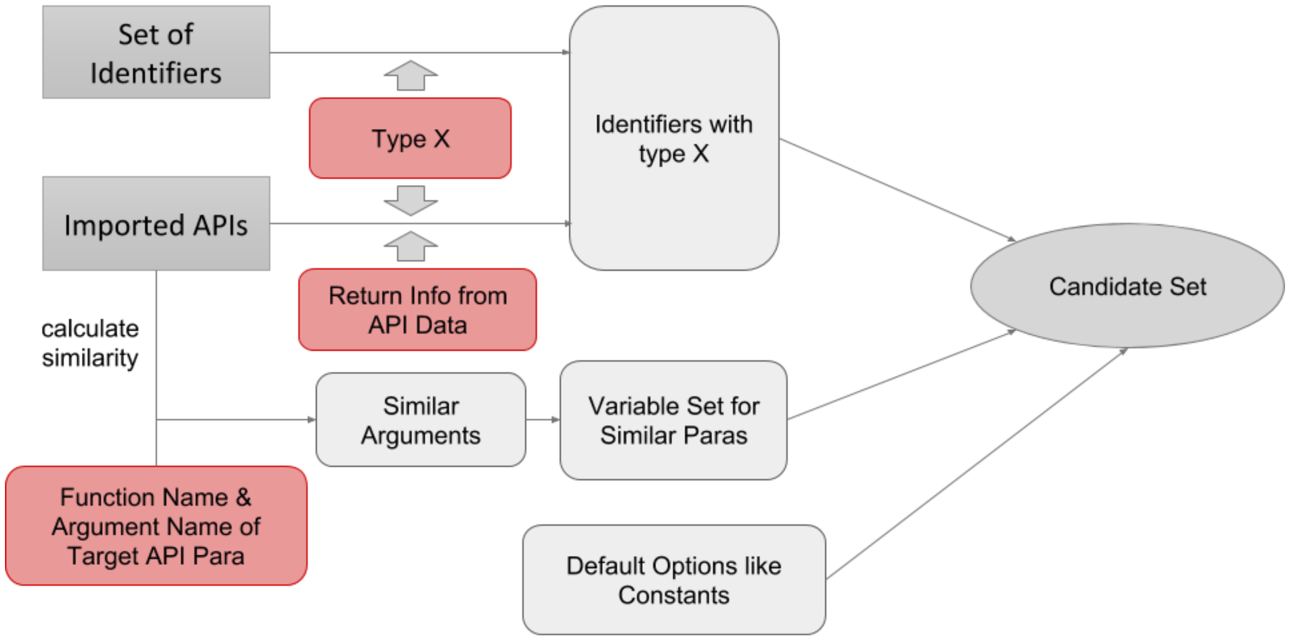

Set of Identifiers
Type X
Identifiers with type X
Imported APIs
Return Info from API Data
Candidate Set
calculate similarity
Similar Arguments
Variable Set for Similar Paras
Function Name & Argument Name of Target API Para
Default Options like Constants

Figure 4.9: Approach for Non-function-type Parameter

We need to generate the suggestion candidates for the first parameter of *compact*, the documentation of which can be found in Figure 4.4. From Figure 4.4, we know that the type of the first parameter is Array. After source code analysis, we can know that *arr* is an Array, and *num* is a Number. We also need to check another imported API function *concat*. It is also a Lodash API function, which returns an Array. Therefore, it is easy to infer that the type of *other* is Array. We should suggest an array constant as our default option as well, and we usually choose an empty array. So for this query, we will generate a candidate set that consists of *arr*, *other*, and an empty array.

## 4.5 Approach for Function-type Parameter

Figure 4.10 demonstrates the approach for suggesting candidates for a function-type parameter.

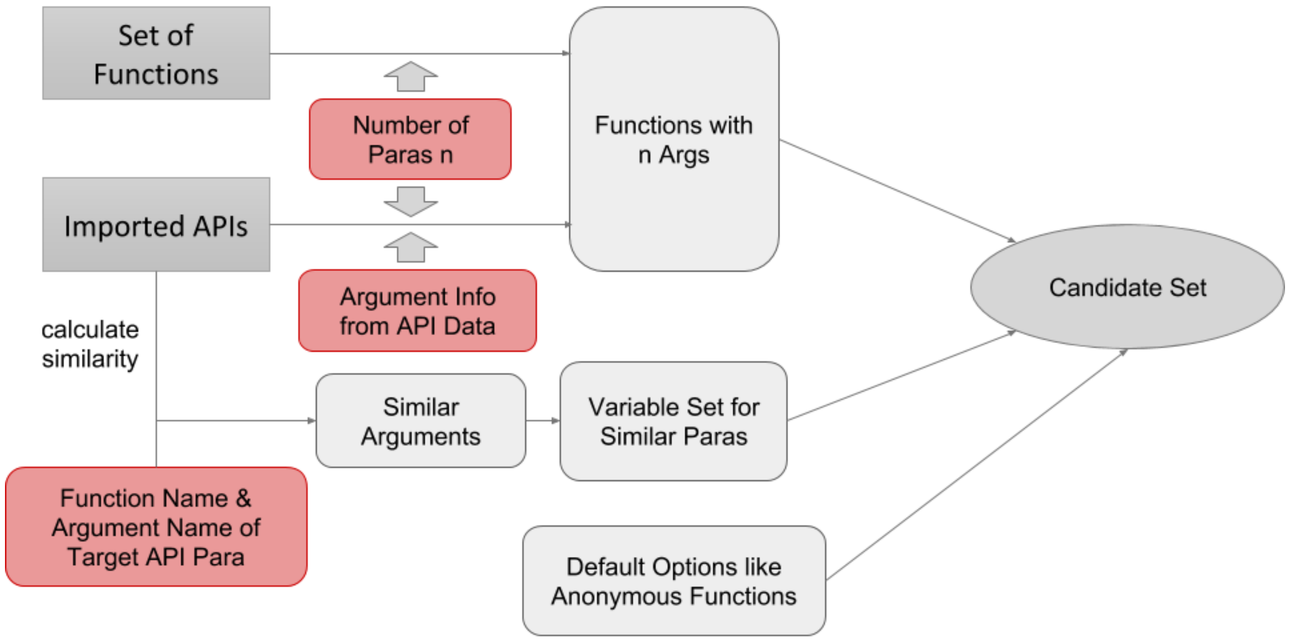

Set of Functions
Number of Paras n
Functions with n Args
Imported APIs
Argument Info from API Data
Candidate Set
calculate similarity
Similar Arguments
Variable Set for Similar Paras
Function Name & Argument Name of Target API Para
Default Options like Anonymous Functions

Figure 4.10: Approach for Function-type Parameter

The approach for Function-type parameter is similar to that for non-function-type parameter.

Because it is only for function-type parameter, we only need to pick the set of functions from the AST data in Section 4.3. As described in Section 4.2, we can know the number of parameters, $n$, for the function-type parameter. So we should check all the variables which represent functions to see if there are any functions called with $n$ parameters. These variables which represent functions with $n$ parameters should be part of the candidate set.

Similarly, We also need to check all the imported APIs in Section 4.2. We look up their argument information in the database. If they have n parameters, then these API calls can also be part of the candidate set.

And for each of the imported APIs, we calculate the similar parameters of similar API functions to our target API. The way to find similar arguments is similar to the corresponding descriptions in Section 4.4, so we do not repeat it here.

Since it is for function-type parameter, we use anonymous functions as our default options here.

We use the following code snippet as an example to illustrate how the approach works.

```
function mutate(o1, o2, mutation, cond) {
    if (cond) {
        return assignWith(o1, o2, mutation);
    } else {
        return assignInWith(o2, o1, ? );
    }
}
```

We need to generate the suggestion candidates for the third parameter of *assignInWith*, the documentation of which can be found in Figure 4.6. From Figure 4.6, we know that it is a function-type parameter, and it requires five parameters. After source code analysis, we can find two function-type variables: *mutate* and *assignWith*, but none of them have five parameters. Then we check other imported API functions and find *assignWith*. It takes three arguments and returns an Object. We cannot find any suggestion candidates until now. However, when we calculate the similarity between *assignWith* and *assignInWith*, we find they are similar APIs, and their third arguments are also similar. So we put the third argument of *assignWith*, *mutation* into the candidate set. In addition, We should suggest an anonymous function as our default option. So for this query, we will generate a candidate set that consists of *mutation*, and an anonymous function.

# Chapter 5

# Experimental Setup

In this section, we will explain how we conduct experiments to evaluate the performance of our approach.

## 5.1 Research Questions

The objective of our experiments is to investigate how useful our approach can be applied for generating parameter suggestions to API users. So we investigate two research questions and design the experiments aims to answer them:

- RQ1: How accurate are the parameter suggestions of the approach?
- RQ2: How many suggestions does the approach generate for input source code files?

The motivation for the first research question is to assess the accuracy of the parameter suggestion approach. It help us know if our approach can generate good suggestions for use cases in real projects.

The motivation for the second research question is to evaluate the efficiency of our parameter suggestion approach. Usually our approach gives more than one suggested variables for one parameter query. However, if our approach gives too many suggestions, it will be difficult for program developers to pick the right variable. Therefore, we want to know if number of suggestions that our approach generates is practical for most real-world situations.

Table 5.1: JavaScript Libraries Used as API Documentation Data Set

| Library | Version |
|---|---|
| Lodash | 4.17.10 |
| Vue.js | 2.5 |
| AngularJS | 1.7.8 |
| Async | 2.6.2 |
| Zlib | 1.0.5 |

## 5.2 Data Sets Collecting

To evaluate the accuracy and efficiency of our API parameter suggestion approach, we attempt to predict the parameters of different API functions from different libraries, using JavaScript source code from real-world open source projects.

### 5.2.1 API Documentation Data Set

The first step of the experiment is to collect the documentation information of five different third-party JavaScript libraries. The libraries are selected based on community popularity and frequency of use. The list of libraries selected is displayed in Table 5.1. For each library, the following information of each API method's documentation are collected and stored in a JSON file:

- Function name
- Function description
- Return type
- Return description
- Argument types
- Argument names
- Argument descriptions

- Number of parameters for any function-type parameters
- Method signature
- Code examples

There are two ways to collect these information.

For some libraries, a web crawler can be used to extract the necessary information from their online documentation if the documentation are well-structured. The web crawlers are written in Python and based on the Scrapy framework [4]. Because each library's online documentation is formatted differently, we have to write a different web crawler for each library.

Another way to collect the documentation information is to manually copy-and-paste this information. This is used when either the library contains only few methods or the online documentation is not well-structured. For libraries with few API functions, it would be more time-consuming to write a web crawler. For badly-structured online documentation, it is hard to write a web crawler that works perfectly, so manually collecting the information is easier and better. In an ideal world, this information might be easily extractable from a website that uses tags or makes the API details easy to access in some other way.

Here is an example of the JSON file [1] of the API function, *assignInWith*, the documentation of which is shown in Figure 4.6:

```
{
    "return": {
        "type": "Object",
        "description": "Returns object."
    },
    "description": "This method is like _.assignIn except that it
                     accepts customizer which is invoked to produce the
                     assigned values. If customizer returns undefined,
                     assignment is handled by the method instead. The
                     customizer is invoked with five arguments:
                     (objValue, srcValue, key, object, source).
```

[1]The example code is utf-8 encoded, so it is human unreadable.

```
                  Note: This method mutates object.",
    "arguments": [
        {
            "type": "Object",
            "name": "object",
            "description": "The destination object."
        },
        {
            "type": "Object",
            "name": "sources",
            "description": "The source objects."
        },
        {
            "type": "Function",
            "name": "customizer",
            "description": "The function to customize assigned values."
        }
    ],
    "signature": "_.assignInWith(object, sources, [customizer])",
    "example": "function\u00a0customizer(objValue,\u00a0srcValue)\u00a0
                {\n\u00a0\u00a0return\u00a0_.isUndefined(objValue)\u00a
                0?\u00a0srcValue\u00a0:\u00a0objValue;\n}\n\u00a0\nvar\
                u00a0defaults\u00a0=\u00a0_.partialRight(_.assignInWith
                ,\u00a0customizer);\n\u00a0\ndefaults({\u00a0'a':\u00a0
                1\u00a0},\u00a0{\u00a0'b':\u00a02\u00a0},\u00a0{\u00a0'
                a':\u00a03\u00a0});\n//\u00a0=>\u00a0{\u00a0'a':\u00a01
                ,\u00a0'b':\u00a02\u00a0}\n",
    "name": "assignInWith"
}
```

### 5.2.2 JavaScript Project Data Set

Once the documentation had been summarized for each API method, we need a large repository of JavaScript files to test against. Therefore, we choose to clone the top 1,000

JavaScript projects from GitHub ordered by their star ratings. In total, the top 1000 projects from GitHub contain over 200,000 JavaScript files.

Because not all these JavaScript files include the target API libraries in the previous data set, we can filter out the files that do not use any target APIs. We use a keyword search to locate every file that uses at least one of the target libraries. This can be done by using a UNIX *grep* command to locate all files containing import statements for target libraries. In the end, we found 1,023 files which make at least one call to a function contained in any of the target library APIs.

## 5.3 Generating Abstract Syntax Tree

According to our approach introduced in Section 4.3 (referring to Figure 4.8), all the JavaScript source code files from Section 5.2.2 must be converted into abstract syntax trees (AST).

We implement our JavaScript parser based on the Esprima framework [2] to generate the AST for each source code file and store it as a JSON file.

Given an example of JavaScript source code file as following:

```
function addOne(a) {
    return a + 1;
}
var b = addOne(3);
```

Here is the JSON file of the AST generated for the example above:

```
{
    "type": "Program",
    "body": [
        {
            "type": "FunctionDeclaration",
            "id": {
                "type": "Identifier",
            "name": "addOne"
        },
        "params": [
            {
                "type": "Identifier",
                "name": "a"
            }
        ],
        "body": {
            "type": "BlockStatement",
            "body": [
                {
                    "type": "ReturnStatement",
                    "argument": {
                        "type": "BinaryExpression",
                        "operator": "+",
                        "left": {
                            "type": "Identifier",
                            "name": "a"
                        },
                        "right": {
                            "type": "Literal",
                            "value": 1,
                            "raw": "1"
                        }
                    }
                }
            ]
        },
        "generator": false,
        "expression": false,
        "async": false
    },
    {
        "type": "VariableDeclaration",
        "declarations": [
                {
                    "type": "VariableDeclarator",
                    "id": {
                        "type": "Identifier",
                        "name": "b"
                    },
                    "init": {
                        "type": "CallExpression",
                        "callee": {
                            "type": "Identifier",
                            "name": "addOne"
                        },
                        "arguments": [
                            {
                                "type": "Literal",
                                "value": 3,
                                "raw": "3"
                            }
                        ]
                    }
                }
            ],
            "kind": "var"
        }
    ],
    "sourceType": "script"
}
```

After generating ASTs for the source code files, we note that only a fraction of each library's API functions are discovered in the GitHub projects we collected. The number of unique methods and number of method instances that were found for each library is shown in Table 5.2. In total, 43 unique functions from the 5 libraries are discovered.

Table 5.2: API Functions Found in the GitHub Projects

| Library | Total number of functions in library | Unique functions found | Function instances found |
|---|---:|---:|---:|
| Lodash | 574 | 10 | 875 |
| Vue.js | 10 | 10 | 561 |
| AngularJS | 9 | 9 | 200 |
| Async | 78 | 11 | 39 |
| Zlib | 7 | 3 | 6 |

### 5.3.1 API Parameter Suggestion

We implement an analyzer based on the our approach introduced in Section 4.4 and Section 4.5, which takes the AST file and the API documentation information as inputs, and outputs a CSV file which contains the parameter prediction results. Each row in the output file contains a set of suggestions for a single argument, the target API function, the target file, the argument number, the actual parameter value, the number of suggestions, and a list of suggestions.

Then we analyze the CSV file containing the predictions to answer our research questions:

- RQ1: How accurate are the parameter suggestions of the approach?
- RQ2: How many suggestions does the approach generate for input source code files?

In RQ1 we compute the accuracy of our prediction results. A given parameter prediction is considered successful if the actual value from the source code file is present in the list of suggestions, otherwise considered failed. To compute the accuracy for each API parameter prediction, we divide the number of successes by the total number of API function calls (the sum of successes and fails for each API function parameter).

In RQ2 we count the number of our suggestions for each API parameter prediction query. For each API parameter, we calculate the median and maximum number of suggestions.

# Chapter 6

# Experiment Results

Based on our experimental results, we are willing to answer the following research questions:

- RQ1: How accurate are the parameter suggestions of the approach?
- RQ2: How many suggestions does the approach generate for input source code files?

## 6.1 RQ1: How accurate are the parameter suggestions of the approach?

To calculate the accuracy of our API parameter suggestion approach, the actual parameter values used in the source code are compared to the list of suggestions. For each parameter, if the list of suggestions contains the actual value, then consider it as successful; if not then it fails. To compute the overall accuracy for each library, we divide the number of successes by the total number of API function calls of the library.

The detailed results for each target library are shown in Table 6.1, Table 6.2, Table 6.3, Table 6.4, and Table 6.5 respectively. The overall results are shown in Table 6.6.

The overall suggestion accuracy of our approach for the target libraries is 64.6%, and our approach achieves an accuracy over 50% for 4 out of 5 libraries. For the most successful library, Async, the accuracy is 81.3%.

However, when we look at the detailed results for each target library, we see a lot of predictions with 0.0% accuracy, which means our approach cannot give any correct

Table 6.1: Parameter Suggestion Accuracy for Lodash

| Function Name | Suggestion Accuracy (%) | | Number of API calls |
|---|---|---|---|
| | 1st Parameter | 2nd Parameter | |
| filter | 46.7 | 93.3 | 105 |
| reduce | 35.4 | 95.1 | 82 |
| some | 52.2 | 78.9 | 90 |
| times | 80.7 | 100.0 | 31 |
| transform | 80.0 | 100.0 | 5 |
| partition | 16.7 | 100.0 | 6 |
| find | 45.3 | 91.5 | 351 |
| map | 35.0 | 83.8 | 160 |
| remove | 45.5 | 63.6 | 11 |
| every | 67.7 | 70.6 | 34 |

Table 6.2: Parameter Suggestion Accuracy for Vue.js

| Function Name | Suggestion Accuracy (%) | | | Number of API calls |
|---|---|---|---|---|
| | 1st Parameter | 2nd Parameter | 3rd Parameter | |
| extend | 95.2 | - | - | 103 |
| filter | 66.7 | 0.0 | - | 12 |
| component | 90.4 | 32.5 | - | 114 |
| mixin | 76.9 | - | - | 26 |
| directive | 100.0 | 66.7 | - | 9 |
| compile | 100.0 | - | - | 2 |
| use | 4.6 | - | - | 176 |
| delete | 60.7 | 85.7 | - | 28 |
| set | 47.3 | 81.8 | 81.8 | 55 |
| nextTick | 77.8 | 72.2 | - | 36 |

suggestion for these parameters even once. This unexpected huge variance in accuracy for different parameters needs further investigation.

Table 6.3: Parameter Suggestion Accuracy for AngularJS

| Function Name | Suggestion Accuracy (%) | | Number of API calls |
|---|---|---|---|
| | 1st Parameter | 2nd Parameter | |
| toJson | 12.5 | 100.0 | 24 |
| forEach | 66.7 | 100.0 | 3 |
| module | 98.1 | 100.0 | 52 |
| equals | 0.0 | 0.0 | 3 |
| isDefined | 0.0 | - | 3 |
| isString | 0.0 | - | 1 |
| isUndefined | 0.0 | - | 2 |
| copy | 31.0 | 66.7 | 29 |
| element | 43.4 | - | 83 |

Table 6.4: Parameter Suggestion Accuracy for Async

| Function Name | Suggestion Accuracy (%) | | | | Number of API calls |
|---|---|---|---|---|---|
| | 1st Parameter | 2nd Parameter | 3rd Parameter | 4th Parameter | |
| times | 100.0 | 100.0 | 100.0 | - | 1 |
| eachSeries | 66.7 | 100.0 | 100.0 | - | 6 |
| parallel | 100.0 | 100.0 | - | - | 3 |
| series | 66.7 | 100.0 | - | - | 3 |
| waterfall | 100.0 | 50.0 | - | - | 2 |
| eachLimit | 40.0 | 66.7 | 100.0 | 100.0 | 15 |
| tryEach | 100.0 | 100.0 | - | - | 2 |
| map | 50.0 | 100.0 | 100.0 | - | 4 |
| each | 0.0 | 100.0 | 0.0 | - | 1 |
| timesLimit | 0.0 | 100.0 | 100.0 | 100.0 | 1 |
| whilst | 100.0 | 100.0 | 100.0 | - | 1 |

To find out why the accuracy for different parameters varies so widely, we manually check all the cases where the accuracy is 0.0%. After manual analysis, we find that the reasons for these failures can be classified into 4 types:

Table 6.5: Parameter Suggestion Accuracy for Zlib

| Function Name | Suggestion Accuracy (%) | | Number of API calls |
|---|---|---|---|
| | 1st Parameter | 2nd Parameter | |
| deflate | 100.0 | 0.0 | 1 |
| gzip | 66.7 | 0.0 | 3 |
| deflateRaw | 100.0 | 0.0 | 2 |

Table 6.6: Overall Parameter Suggestion Accuracy for Target Libraries

| Library | Overall Accuracy (%) |
|---|---|
| Lodash | 67.0 |
| Vue.js | 57.7 |
| AngularJS | 64.3 |
| Async | 81.3 |
| Zlib | 41.7 |
| Overall | 64.6 |

1. The parameter is imported from another file or library.
2. The parameter is contained within a dictionary.
3. The parameter is a property of an object.
4. It is difficult to infer the type of the right parameter from given code context.

The statistics of the reasons are displayed in Table 6.7. The major reasons are that the parameter is an object property, and that it is difficult to infer the type of the right parameter. To solve these problems, we need to improve the type analysis techniques on JavaScript source code, but this requires significant effort so we leave it as possible future work.

Our documentation based API parameter suggestion approach can give good suggestions in most cases. On average, the probability that the correct parameter is in our generated suggestion list is 64.6%.

Table 6.7: Reasons for Fail Cases

| Reason | Count | Percentage |
|---|---|---|
| Imported | 6 | 18.2% |
| Dictionary member | 1 | 3.0% |
| Object property | 13 | 39.4% |
| Type not detected | 13 | 39.4% |

## 6.2 RQ2: How many suggestions does the approach generate for input source code files?

We use this research question to investigate To answer RQ2, we compute the median and maximum number of suggestions for each API function parameters. Median numbers can reflect the efficiency of our approach in average cases, and maximum numbers can show the situation of the worst cases. All these results for different target libraries are displayed in Table 6.8, Table 6.9, Table 6.10, Table 6.11, and Table 6.12 respectively.

Table 6.8: Number of Parameter Suggestions Statistics for Lodash

| Function Name | Median of Suggestions | | Maximum of Suggestions | |
|---|---|---|---|---|
| | 1st Parameter | 2nd Parameter | 1st Parameter | 2nd Parameter |
| filter | 9 | 5 | 46 | 18 |
| reduce | 8 | 1 | 46 | 4 |
| some | 7 | 5 | 36 | 30 |
| times | 2 | 4 | 5 | 23 |
| transform | 7 | 1 | 10 | 1 |
| partition | 7 | 5 | 34 | 14 |
| find | 5 | 4 | 46 | 31 |
| map | 9 | 4 | 47 | 31 |
| remove | 6 | 3 | 15 | 8 |
| every | 13 | 8 | 45 | 31 |

Table 6.9: Number of Parameter Suggestions Statistics for Vue.js

| Function Name | Median of Suggestions | | | Maximum of Suggestions | | |
|---|---|---|---|---|---|---|
| | 1st Para | 2nd Para | 3rd Para | 1st Para | 2nd Para | 3rd Para |
| extend | 2 | - | - | 8 | - | - |
| filter | 1 | 3 | - | 2 | 16 | - |
| component | 1 | 10 | - | 9 | 52 | - |
| mixin | 1 | - | - | 3 | - | - |
| directive | 1 | 5 | - | 1 | 7 | - |
| compile | 1 | - | - | 1 | - | - |
| use | 4 | - | - | 55 | - | - |
| delete | 2 | 1 | - | 9 | 3 | - |
| set | 2 | 1 | 8 | 10 | 3 | 21 |
| nextTick | 2 | 2 | - | 4 | 3 | - |

Table 6.10: Number of Parameter Suggestions Statistics for AngularJS

| Function Name | Median of Suggestions | | Maximum of Suggestions | |
|---|---|---|---|---|
| | 1st Parameter | 2nd Parameter | 1st Parameter | 2nd Parameter |
| toJson | 6 | 6 | 21 | 9 |
| forEach | 2 | 4 | 2 | 7 |
| module | 2 | 1 | 6 | 5 |
| equals | 16 | 17 | 42 | 43 |
| isDefined | 13 | - | 14 | - |
| isString | 3 | - | 3 | - |
| isUndefined | 3 | - | 3 | - |
| copy | 13 | 29 | 57 | 35 |
| element | 2 | - | 7 | - |

From the detailed tables, we find that the median number of suggestions are usually smaller than or equal to 10 (an acceptable number of suggestions), which means our approach does not give too many suggestions overall. However, the maximum number of suggestions for each parameter can be much larger. It can be even as high as 57 for the

Table 6.11: Number of Parameter Suggestions Statistics for Async

| Function Name | Median of Suggestions | | | | Maximum of Suggestions | | | |
|---|---|---|---|---|---|---|---|---|
| | 1st Para | 2nd Para | 3rd Para | 4th Para | 1st Para | 2nd Para | 3rd Para | 4th Para |
| times | 4 | 14 | 14 | - | 4 | 14 | 14 | - |
| eachSeries | 3 | 13 | 13 | - | 4 | 13 | 13 | - |
| parallel | 3 | 10 | - | - | 8 | 18 | - | - |
| series | 10 | 21 | - | - | 18 | 30 | - | - |
| waterfall | 4 | 19 | - | - | 4 | 19 | - | - |
| eachLimit | 5 | 2 | 20 | 20 | 20 | 7 | 30 | 30 |
| tryEach | 1 | 11 | - | - | 1 | 11 | - | - |
| map | 10 | 11 | 11 | - | 11 | 11 | 11 | - |
| each | 1 | 4 | 4 | - | 1 | 4 | 4 | - |
| timesLimit | 4 | 5 | 29 | 29 | 4 | 5 | 29 | 29 |
| whilst | 8 | 8 | 8 | - | 8 | 8 | 8 | - |

worst case. So for this kind of cases with too many suggestions, our approach still needs to be refined.

We also draw a bar chart Figure 6.1 to display the distribution of numbers of suggestions. In Figure 6.1, blue bars shows the distribution for only successful cases in Section 6.1, and red bars shows the distribution for all the cases. The numbers on the horizontal axis represent the number of suggestions. We aggregate the numbers larger than or equal to 15 together as ">= 15". The height of each bar represents the percentage of number of suggestions occurs. For example, the first blue bar in Figure 6.1 means that 21.4% of parameter predictions give only 1 suggestion when considering only successful cases.

Comparing the blue and red bars, we do not see any significant difference. Over 60% predictions suggest at most 5 variables, and over 80% suggest at most 10 variables. Both charts show that our approach is much more likely to give a smaller amount of suggestions, which is more efficient and more helpful for developers. Therefore, in general our approach is efficient and usually do not generate excessive amount of suggestions.

Our documentation-based API parameter suggestion approach usually does not generate too many suggestions. On average, over 60% predictions suggest at most 5 variables, and over 80% suggest at most 10 variables.

Table 6.12: Number of Parameter Suggestions Statistics for Zlib

| Function Name | Median of Suggestions | | Maximum of Suggestions | |
|---|---|---|---|---|
| | 1st Parameter | 2nd Parameter | 1st Parameter | 2nd Parameter |
| deflate | 2 | 1 | 2 | 1 |
| gzip | 2 | 1 | 4 | 2 |
| deflateRaw | 11 | 6 | 11 | 6 |

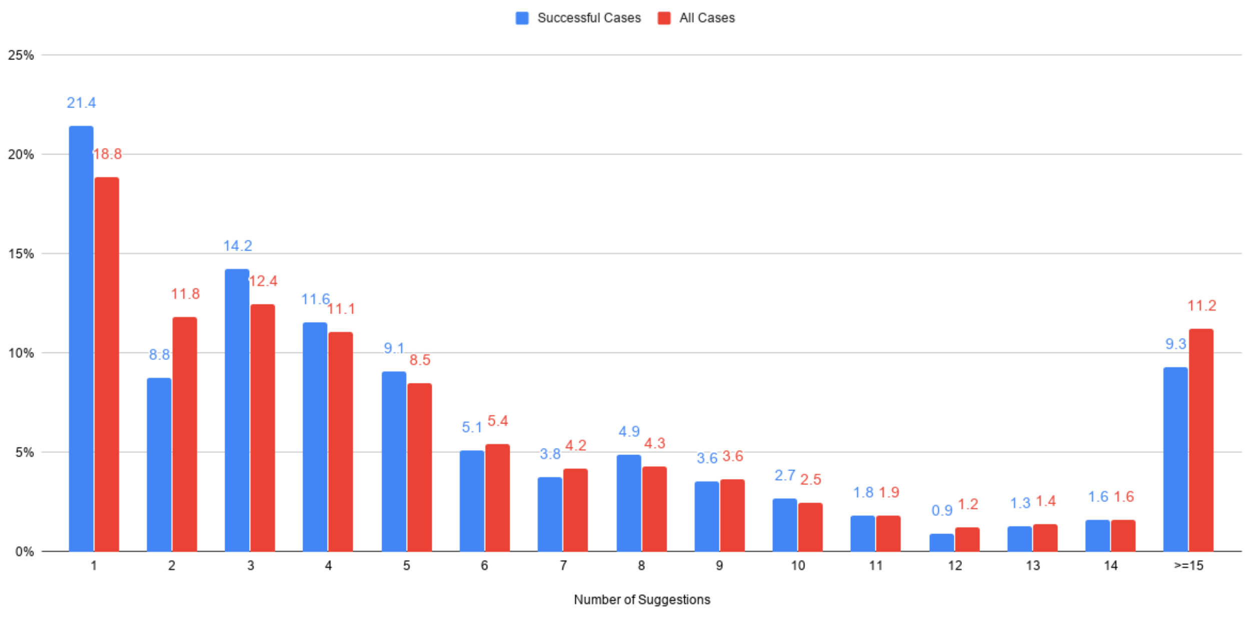


Figure 6.1: Distribution of Numbers of Suggestions

# Chapter 7

# Threats to Validity

## 7.1 Data Set Selection

In the data collection process, we collect the documentation information for 5 third-party JavaScript libraries, and 1,000 JavaScript projects from GitHub.

For this research, we focused on a single programming language: JavaScript. While our approach can be generalized to other languages since it does not require any JavaScript specific properties, we suspect that our experimental results from this thesis might be quite different for other programming languages.

Since our approach is based on documentation analysis, the quality of API documentation is an important factor that will impact our experimental results. Because we want our results to be more representative, we choose our target libraries based on their community popularity and frequency of use. However, it still causes some bias since most well-known and popular third-party libraries are also well-documented. It is hard to avoid, and what we can do is only to strike a balance.

We believe our sample of libraries is representative of most JavaScript APIs since we choose them based on their popularity. However, there are too many JavaScript APIs in real-world. Therefore, further research is required to get a more complete picture of typical API usage.

Similarly, although we tried to avoid bias when collecting testing projects, by picking the top 1,000 JavaScript projects on GitHub according to their star rating, there are much more projects in real-world, and not all of them are on GitHub. We also notice that not all of the API functions in the five libraries are covered in our testing projects. Therefore, further research on more real-world projects can be conducted to get more generalizable results.

## 7.2 API Documentation Information Collection

Threats may also come from our data mining and analysis techniques. To collect documentation information for each library, we used a combination of web crawler scripts and manual analysis. If there are any errors copying the documentation information, it would definitely influence the accuracy of our results. We examine the JSON files of manually created documentation information, making sure there is no such error.

# Chapter 8

# Conclusion and Future Works

## 8.1 Conclusion

APIs are highly beneficial to modern software development, as they free software developers from writing repetitive code, and provides more robust implementations. They improve the efficiency of software development. However, learning how to use an API effectively can be slow and tedious. When an unfamiliar API library is to be used for the first time, the developer must typically pause their current task to chase down documentation and code samples that describe the input, output, and intended use of the desired API methods. Since developers use APIs so frequently in their daily work, the time they spend searching for API information and learning has a considerable impact on their productivity.

Trying to solve this usability barrier caused by unfamiliar libraries, this thesis proposes an approach to predict the parameters when calling an API function. Our approach preprocesses API documentation, and help developers extract possible parameter candidates from the code context, and give them as suggestions.

Unlike other existing approach [44], our approach is based on API documentation analysis, so it does not require a lot example use cases to study from. Also [44] can only be used for Java projects, but our approach can be used for both statically-typed and dynamically-typed languages.

To evaluate the accuracy of our approach, we analyzed over 200,000 files from the top 1,000 JavaScript projects on GitHub. We parsed every file and predicted the parameters of every function belonging to any of 5 popular JavaScript libraries. 1,681 instances of target functions are discovered. We compare each prediction to the parameter that the developer actually used in the source code file to compute the accuracy. On average, the probability that the correct parameter is in our generated suggestion list is 64.6%.

## 8.2 Future Works

For future works, we plan to explore the following aspects:

**Applying to other programming languages:** Although we test our approach on JavaScript libraries and projects, it can be applied to other programming languages as well, because it does not require any language specific properties. We believe that this approach can also perform well on other languages, but future experiments are still needed to validate it.

**API documentation quality:** The quality of API documentation can influence the documentation based API suggestion approaches, so on the other hand, these documentation based approaches can help examining and improving the quality of API documentation.

# Bibliography

[1] Eclipse 3.6.2. `https://archive.eclipse.org/eclipse/downloads/drops/R-3.6.2-201102101200`.

[2] Esprima. `https://esprima.org`.

[3] JBoss 5.0. `https://docs.jboss.org/jbossas/docs/Server_Configuration_Guide/beta500/html/index.html`.

[4] Scrapy. `https://scrapy.org`.

[5] Tomcat 7.0. `http://tomcat.apache.org/tomcat-7.0-doc`.

[6] M. Abadi, L. Cardelli, B. Pierce, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268, Apr. 1991.

[7] M. Asaduzzaman, C. Roy, K. Schneider, and D. Hou. Cscc: Simple, efficient, context sensitive code completion. *Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014*, pages 71–80, 12 2014.

[8] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The fortran automatic coding system. In *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*, IRE-AIEE-ACM '57 (Western), pages 188–198, New York, NY, USA, 1957. ACM.

[9] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 242–253, New York, NY, USA, 2018. Association for Computing Machinery.

[10] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. ACM.

[11] R. P. L. Buse and W. Weimer. Synthesizing api usage examples. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 782–792, Piscataway, NJ, USA, 2012. IEEE Press.

[12] N. R. Carvalho, A. Simões, and J. J. Almeida. Open source software documentation mining for quality assessment. In *Advances in Information Systems and Technologies*, pages 785–794. Springer, 2013.

[13] D. H. Dalip, M. A. Gonçalves, M. Cristo, and P. Calado. Automatic assessment of document quality in web collaborative digital libraries. *J. Data and Information Quality*, 2(3):14:1–14:30, Dec. 2011.

[14] S. Endrikat, S. Hanenberg, R. Robbes, and A. Stefik. How do api documentation and static typing affect api usability? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 632–642, New York, NY, USA, 2014. ACM.

[15] A. Forward and T. C. Lethbridge. The relevance of software documentation, tools and technologies: A survey. In *Proceedings of the 2002 ACM Symposium on Document Engineering*, DocEng '02, pages 26–33, New York, NY, USA, 2002. ACM.

[16] J. M. Fowkes and C. A. Sutton. Parameter-free probabilistic api mining across github. In *FSE 2016*, 2016.

[17] A. Ginzberg, L. Kostas, and T. Balakrishnan. Automatic code completion. Technical report, Technical Report. Stanford CS224n Class Project, 2017.

[18] R. Holmes, R. J. Walker, and G. C. Murphy. Strathcona example recommendation tool. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 237–240, New York, NY, USA, 2005. ACM.

[19] S.-P. Ma, H.-J. Lin, C.-A. Yu, and C.-Y. Lee. Web api recommendation based on service cooperative network. pages 1922–1925, 05 2017.

[20] J. McCarthy. Lisp: A programming system for symbolic manipulations. In *Preprints of Papers Presented at the 14th National Meeting of the Association for Computing Machinery*, ACM '59, pages 1–4, New York, NY, USA, 1959. ACM.

[21] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller. Mining trends of library usage. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, IWPSE-Evol '09, pages 57–62, New York, NY, USA, 2009. ACM.

[22] J. Montandon, H. Borges, D. Felix, and M. Valente. Documenting apis with examples: Lessons learned with the apiminer platform. 10 2013.

[23] M. Mooty, A. Faulring, J. Stylos, and B. A. Myers. Calcite: Completing code completion for constructors using crowds. In *Proceedings of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC '10, pages 15–22, Washington, DC, USA, 2010. IEEE Computer Society.

[24] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 69–79, Piscataway, NJ, USA, 2012. IEEE Press.

[25] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, page 383–392, New York, NY, USA, 2009. Association for Computing Machinery.

[26] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active code completion. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 859–869. IEEE Press, 2012.

[27] M. Pradel and T. R. Gross. Detecting anomalies in the order of equally-typed method arguments. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 232–242, New York, NY, USA, 2011. ACM.

[28] M. M. Rahman, C. K. Roy, and D. Lo. Rack: Automatic api recommendation using crowdsourced knowledge. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 349–359. IEEE, 2016.

[29] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, page 317–326, USA, 2008. IEEE Computer Society.

[30] C. Rubio-González and B. Liblit. Expect the unexpected: Error code mismatches between documentation and the real world. pages 73–80, 12 2010.

[31] M. Saied, O. Benomar, H. Abdeen, and H. Diro. Mining multi-level api usage patterns. 03 2015.

[32] T. W. Schiller, K. Donohue, F. Coward, and M. D. Ernst. Case studies and tools for contract specifications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 596–607, New York, NY, USA, 2014. Association for Computing Machinery.

[33] S. Subramanian, L. Inozemtseva, and R. Holmes. Live api documentation. In *Proceedings of the 36th International Conference on Software Engineering*, pages 643–652. ACM, 2014.

[34] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2727–2735. ACM, 2019.

[35] L. Tan, D. Yuan, and G. Krishna. /*icomment: bugs or bad comments?*/. volume 41, pages 145–158, 01 2007.

[36] L. Tan, Y. Zhou, and Y. Padioleau. Acomment: Mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 11–20, New York, NY, USA, 2011. Association for Computing Machinery.

[37] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. @tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, page 260–269, USA, 2012. IEEE Computer Society.

[38] J. Visser, A. van Deursen, and S. Raemaekers. Measuring software library stability through historical version analysis. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, pages 378–387, Washington, DC, USA, 2012. IEEE Computer Society.

[39] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage api usage patterns from source code. pages 319–328, 01 2013.

[40] E. Wong, L. Zhang, S. Wang, T. Liu, and L. Tan. Dase: Document-assisted symbolic execution for improving automated software testing. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 620–631, Piscataway, NJ, USA, 2015. IEEE Press.

[41] J. Yang, E. Wittern, A. T. T. Ying, J. Dolby, and L. Tan. Towards extracting web api specifications from documentation. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 454–464, New York, NY, USA, 2018. Association for Computing Machinery.

[42] W. Yuan, H. H. Nguyen, L. Jiang, and Y. Chen. Libraryguru: Api recommendation for android developers. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, pages 364–365, New York, NY, USA, 2018. ACM.

[43] M. Zelkowitz. *Advances in Computers, Volume 77.* Academic Press, Inc., Orlando, FL, USA, 1st edition, 2009.

[44] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic parameter recommendation for practical api usage. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 826–836, Piscataway, NJ, USA, 2012. IEEE Press.

[45] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 318–343, Berlin, Heidelberg, 2009. Springer-Verlag.