Imperial College London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Static Analysis for Asynchronous JavaScript Programs

Author:
Thodoris Sotiropoulos

Supervisor:
Dr Benjamin Livshits

Submitted in partial fulfillment of the requirements for the MSc degree in Advanced Computing of Imperial College London

September 2018

Abstract

Asynchrony has become an inherent element of JavaScript, because it is a mean for improving the scalability and performance of modern web applications. To this end, JavaScript provides programmers with a wide range of constructs and features for developing code that performs asynchronous computations, including but not limited to timers, promises and non-blocking I/O. However, the data flow imposed by asynchrony is implicit, and is not always well-understood by the developers who introduce many asynchrony-related bugs to their programs. Even worse, there are few tools and techniques available for analysing and reasoning about such asynchronous applications. In this work, we address this issue by designing and implementing one of the first static analysis schemes capable of dealing with almost all the asynchronous primitives of JavaScript up to the 7th edition of the ECMAScript specification. Specifically, we introduce *callback graph*, a representation for capturing data flow between asynchronous code. We exploit callback graph for designing a more precise analysis that respects the execution order between different asynchronous functions. We parameterise our analysis with two novel context-sensitivity strategies and we end up with multiple analysis variations for building callback graph. Then we perform a number of experiments on a set of hand-written and real-world JavaScript programs. Our results show that our analysis can be applied on medium-sized programs achieving 79% precision. The findings further suggest that analysis sensitivity is able to improve accuracy up to 11,3% on average, without highly sacrificing performance.

Acknowledgments

I would like to thank my supervisor Dr Benjamin Livshits for giving me the opportunity to work with him on a challenging topic, and for all the interest, support, and encouragement that he expressed throughout the project.

Contents

1 Introduction **1**
- 1.1 Contributions . . . 2
- 1.2 Report Organisation . . . 3

2 Background **4**
- 2.1 The JavaScript Language . . . 4
  - 2.1.1 Features . . . 4
  - 2.1.2 Bugs . . . 5
- 2.2 Static Analysis for JavaScript . . . 6
  - 2.2.1 Basics . . . 6
  - 2.2.2 JavaScript-Specific Analysis . . . 9
- 2.3 The Asynchronous JavaScript . . . 14
  - 2.3.1 Sources of Asynchrony . . . 14
  - 2.3.2 Runtime Architecture . . . 17
  - 2.3.3 Concurrency Bugs . . . 18
  - 2.3.4 Detecting Concurrency Bugs . . . 20

3 Modelling Asynchrony **22**
- 3.1 The λ_q calculus . . . 22
  - 3.1.1 Syntax and Domains . . . 23
  - 3.1.2 Semantics . . . 25
  - 3.1.3 Modelling the Event Loop . . . 28
- 3.2 Modelling Asynchronous Primitives . . . 29
  - 3.2.1 Promises . . . 30
  - 3.2.2 Timers . . . 32
  - 3.2.3 Asynchronous I/O . . . 33
- 3.3 Comparison with Existing Models . . . 33

4 The Core Analysis **35**
- 4.1 Introduction to TAJS . . . 35
  - 4.1.1 Control-Flow Graph . . . 36
  - 4.1.2 The Analysis Framework . . . 36
  - 4.1.3 Computing the Solution . . . 38
- 4.2 Extending TAJS . . . 39
  - 4.2.1 Tracking Execution Order . . . 42
- 4.3 Callback Graph . . . 43
  - 4.3.1 Definition . . . 43
  - 4.3.2 Computing Callback Graph . . . 44

4.4 Analysis Sensitivity 48
4.4.1 Callback-Sensitivity 48
4.4.2 Sources of Imprecision 51
4.4.3 Context-Sensitivity Strategies 51
4.5 Implementation Details 53

5 Empirical Evaluation **55**
5.1 Experimental Setup 55
5.2 Micro Benchmarks 57
5.3 Macro Benchmarks 60
5.4 Case Studies 61
5.5 Conclusions 66
5.6 Threats to Validity 67

6 Conclusions & Future Work **68**
6.1 Summary 68
6.2 Future Work 69
6.2.1 Limitations & Optimisations 69
6.2.2 Further Research 70

Appendices **78**

A Semantics of λ_{q} **79**

B Performance of Micro Benchmarks **81**

C Ethics Checklist **82**

D Ethical and Professional Considerations **84**

Chapter 1

Introduction

JavaScript is an integral part of web development. Since its initial release in 1995, it has evolved from a simple scripting language, which was primarily used for interacting with web pages, into a complex and general-purpose programming language used for developing both client- and server-side applications. The emergence of Web 2.0 along with the dynamic features of JavaScript, which facilitate a flexible and rapid development, have led to the undeniable increase of its popularity. Indeed, according to the annual statistics provided by Github, which is the leading platform for hosting open source software, JavaScript is by far the most popular and active programming language for 2017 (Github, 2017).

Although the dominance of JavaScript is impressive, it has been widely criticised, as it poses a number of concerns as to the security and correctness of the programs (Richards et al., 2010). JavaScript is a language with a lot of dynamic and metaprogramming features, including but not limited to prototype-based inheritance, dynamic property lookups, implicit type coercions, dynamic code loading, etc. Many developers often do not understand or do not properly use these features, introducing errors to their programs, which are difficult to debug, or baleful security vulnerabilities. In this context, JavaScript has attracted many engineers and researchers over the past decade to: 1) study and reason about its peculiar characteristics, and 2) develop new tools and techniques, such as type analysers (Jensen et al., 2009; Lee et al., 2012; Kashyap et al., 2014), IDE and refactoring tools (Feldthaus et al., 2011; Feldthaus and Møller, 2013; Feldthaus et al., 2013; Gallaba et al., 2017), or bug and vulnerability detectors (Maffeis and Taly, 2009; Guarnieri and Livshits, 2009; Petrov et al., 2012; Mutlu et al., 2015; Davis et al., 2017; Staicu et al., 2018), to assist developers with the development and maintenance of their applications. Program analysis, and especially static analysis, which automatically computes facts about program's behaviour without actually running it, plays a crucial role in the design of such tools (Sun and Ryu, 2017).

Additionally, preserving the scalability of modern web applications has become more important than ever. To this end, in order to improve the throughput of web programs, JavaScript has started to adopt an event-driven programming paradigm. In this context, code is executed *asynchronously* in response to certain events. In recent years, this asynchrony has become a salient and intrinsic element of JavaScript, as many asynchrony-related features have been added to the language's core specification (i.e. ECMAScript), most notably promises (ECMA-262, 2015, 2018). Promises are special constructs which facilitate the development and organisation of asynchronous code by forming chains of asynchronous computation. Beyond that, many JavaScript applications are written to perform non-blocking

I/O operations. Unlike traditional statements, when we perform a non-blocking I/O operation, execution is not interrupted until that operation completes. For instance, a file system operation is done in an asynchronous way, allowing other code to be executed, while I/O takes place.

Like the other characteristics of JavaScript, asynchrony is not often well-understood by the programmers, as the large number of asynchrony-related questions issued in popular sites like `stackoverflow.com`[1] (Madsen et al., 2015, 2017), or the number of bugs reported in open source repositories (Wang et al., 2017; Davis et al., 2017) indicate. However, existing tools and techniques have limited (and in many cases no) support for asynchronous programs. In particular, existing tools mainly focus on the event system of client-side JavaScript application (Jensen et al., 2011; Park et al., 2016), and they lack support of the more recent features added to the language like promises.

In this work we tackle this issue, by designing and implementing a static analysis that deals with asynchronous JavaScript programs. For that purpose, we first define a model for understanding and expressing JavaScript's asynchronous constructs, and then we extend an existing analyser, namely *TAJS* (Jensen et al., 2009, 2010, 2011), by incorporating our model into the analyser. We propose a new representation, which we call *callback graph*, which provides information about the execution order of asynchronous code. Previous attempts on analysing asynchronous applications conservatively assumed that asynchronous functions are executed in any order (Jensen et al., 2011; Park et al., 2016). Callback graph, which is proposed in this report, tries to shed light on how data flow between asynchronous code is propagated. Contrary to previous works, we leverage callback graph and devise a more precise analysis which respects the execution order of asynchronous functions. Furthermore, we parameterise our analysis with two new sensitivity policies (context-sensitivity) that are specifically designed for asynchronous code. Specifically, asynchronous functions are distinguished based on the promise object on which they are registered or the next computation to which execution proceeds.

The ultimate goal of this report is to end up with a generic technique which can be seen as a stepping-stone to the analysis of asynchronous applications and facilitate the construction of new tools and techniques for JavaScript's asynchrony.

1.1 Contributions

This report makes the following five contributions:

- We propose a calculus, namely, λ_q for modelling asynchronous primitives in JavaScript language, including timers, promises, and asynchronous I/O operations. Our calculus is a variation of existing calculuses (Loring et al., 2017; Madsen et al., 2017), and provides constructs and domains specifically targeted for our analysis.
- We design and implement a static analysis that is capable of handling asynchronous JavaScript programs on the basis of λ_q. To the best of our knowledge, our analysis is the first to deal with JavaScript promises. Our work extends TAJS, an existing type analyser for JavaScript, and is designed as a data-flow inter-procedural analysis.

[1] `https://stackoverflow.com/`

- We propose *callback graph*, a representation which illustrates the data flow between asynchronous functions. Building on that, we propose a more analysis, namely, *callback-sensitive* analysis which internally consults callback graph to retrieve information about the temporal relations of asynchronous functions so that it propagates data flow accordingly.
- We design two novel context-sensitivity strategies which are specifically used for separating asynchronous functions.
- We evaluate the performance and precision of our analysis on a set of hand-written micro benchmarks and a set of real-word JavaScript modules. The evaluation includes a number of experiments with different parameterisations of our analysis. For the impatient reader, we find that our prototype can be applied on small and medium-sized asynchronous programs, and the analysis sensitivity is beneficial for improving analysis precision, as observed in both micro- and macro benchmarks.

1.2 Report Organisation

The rest of this report is organised as follows: we start in Chapter 2 with a brief overview of the background and related work. We present our calculus and how we express JavaScript's asynchronous primitives in terms of λ_q in Chapter 3. In Chapter 4, we discuss the details of our analysis: specifically, we summarise the necessary extensions made in TAJS, we introduce the concept of callback graph, and finally, we give a description of our new analysis sensitivities. In Chapter 5, we present the evaluation and the experimental results of our work. We wrap up with our conclusions and discussion about future work in Chapter 6.

Chapter 2

Background

In this chapter, we give a description of the background. In Section 2.1, we give a brief introduction to JavaScript language, and the main kind of errors that arise from its intricate features. In Section 2.2, we proceed to the material as to the static analysis techniques developed for JavaScript. Finally, Section 2.3 gives an introduction about asynchrony in JavaScript.

2.1 The JavaScript Language

JavaScript is a standardised language. Its specification, namely, *ECMAScript* (ECMA-262, 2018) informally describes JavaScript's features and characteristics. Its current edition is 9th (ECMA-262, 2018) which was published in 2018.

2.1.1 Features

One of the most important characteristics of JavaScript is that it is a dynamically typed language which means that the type of a program variable can change during execution. JavaScript supports two different types: primitives and objects. Primitives describe numbers, strings, booleans or symbols; everything else is an object. An object is expressed through key-value pairs which denote the set of the object's properties along with their values. Developers can dynamically change the structure of an object by adding new properties or removing existing ones. Beyond that, JavaScript heavily performs implicit type conversions between operands, depending on the context on which they are evaluated. For instance, when an object `x` is compared with a string, (e.g. `x == 'foo'`), `x` is implicitly converted into a string by calling function `toString()`. Another dynamic feature of JavaScript is dynamic code generation via functions like `eval` or other constructs like `Function`. The use of such functions has been widely criticised by the community as they have severe security impacts and hinder the effectiveness of static analysis techniques (Richards et al., 2011).

Also, unlike other object-oriented languages like Java or C++, JavaScript supports prototype-based instead of class-based inheritance. This means that objects inherit the structure and state of other objects whose template can potentially change at runtime. On the contrary, in a class-based model, objects are instantiated from classes which are static, i.e. their template cannot change at runtime. All objects in JavaScript have an internal property (i.e. it is not visible from developers[1]), which points to the prototype object from which they are derived.

[1] To be more precise, some implementations of JavaScript like those of Firefox or Microsoft provide methods

This internal property is different from the property "prototype" which can be explicitly found on constructor objects. The former is used to inspect the prototype chain during a property lookup, whereas the latter denotes the prototype object used to build new objects through keyword `new`.

Prototype-based model often leads to program behaviours that are not well-understood by the developers who are more familiar with the class-based model (Lee et al., 2012). For example, consider the following code:

```
1  function Foo() {
2    this.x = 1;
3  }
4  x = new Foo();
5  // we update the prototype of constructor.
6  Foo.prototype = {bar: 2};
7  y = new Foo();
8
9  y instanceof Foo      //true
10 x instanceof Foo;     // false
```

In this example, function `Foo` is used as a constructor to create two new objects, i.e. objects `x` and `y` at line 4 and 7 respectively. During function declaration, a prototype object is created at the background and is referenced by property `prototype` of object `Foo`. Since we change the prototype of `Foo` at line 6, the resulting objects `x` and `y` have different object types (lines 8, 9), even though they come from the same constructor.

JavaScript also adopts an event-driven programming paradigm. which enables the development of highly responsive applications (Zheng et al., 2011; Madsen et al., 2015; Davis et al., 2017). That is, functions can be executed asynchronously in response to certain events. At this point, we defer the discussion about event-driven and asynchronous JavaScript programs, as we will come back to it in a dedicated section (Section 2.3).

2.1.2 Bugs

Several prior studies (Ocariza Jr. et al., 2011, 2013) have examined various bugs occured in web applications, analysing their root causes. According to these studies, the most frequent bugs found in JavaScript code are property accesses of `undefined` or `null` variables, use of undefined functions (i.e. functions that are not declared anywhere in the code) or incorrect use of web APIs (e.g. built-in functions, DOM API, etc.) (Ocariza Jr. et al., 2011, 2013). It has been shown that these categories of bugs are highly associated with many dynamic features of JavaScript. For example, property accesses of `undefined` variables are correlated with the dynamic deletion of object members (Ocariza Jr. et al., 2011). At the same time, prototype-based inheritance seems to mislead developers who often access incorrect object properties (Ocariza Jr. et al., 2011). Also, the lack of a type system along with the subtle type coercions lead to surpising results for the programmers, e.g. adding `"1" + 0` results in a string with value `"10"` (Gao et al., 2017).

Another studies (Wang et al., 2017; Davis et al., 2017) have pointed out the lack of understanding that many developers have regarding the asynchrony-related features of JavaScript

to access this internal property.

which causes a lot of severe concurrency bugs. We will come back to these sutdies in Section 2.3.3.

2.2 Static Analysis for JavaScript

In this section, we focus on static analysis techniques. First, we describe the required background about static analysis. Then, we present some of the static analysis techniques that have been developed specifically for JavaScript programs along with their challenges.

2.2.1 Basics

The goal of static analysis is to automatically produce summaries about general properties of a program without actually running it (Emanuelsson and Nilsson, 2008). Examples of such properties are program points where a variable is used, the list of values that a variable may hold, list of methods invoked by a particular caller method, etc. By taking advantage of these summaries, a static analyser can detect various kinds of bugs. For instance, knowing where a variable is used and written, a static analyser is able to decide, if there is a read of an uninitialised variable. Beyond that, static analysis techniques have been adopted for program optimisation such as those employed by the optimiser of modern compilers (Cooper and Torczon, 2012). However, typically, static analysis is not able to reason about the functional properties of a program (Emanuelsson and Nilsson, 2008). For example, it cannot determine if there is a mistake in the implementation of an algorithm or not.

Abstractions

Since the program is not actually executed, analysis should be able to approximate the possible executions of the program in order to produce meaningful results. The are two different types of approximations: 1) *under-approximation*, and 2) *over-approximation*. When analysis under-approximates a program, it may miss some properties that exist in the real program, e.g. it may fail to detect a bug, etc. In this case, the analysis is called *complete*. On the other hand, when analysis over-approximates a program, it may capture some properties that are not present in the initial program, e.g. it may mistakenly report a bug, etc. This analysis is called *sound*. Depending on the context on which analysis is applied, someone may pursue soundness instead of completeness and vice versa. For example, a bug detector used in a safety-critical system should be sound, i.e. it should not miss any bug. On the other hand, an autocompletion tool used in an IDE should be complete, that is, it should not produce spurious suggestions to the developer.

A standard way to achieve such approximations is by abstracting the heap and executions of a program. Specifically, concrete values that program variables may hold along with the semantics of the operations applied on them are replaced with abstract ones. A typical way to abstract possible values of a variable is through lattices (Kam and Ullman, 1977). For instance, Figure 2.1 shows a lattice, which was inspired from the work of Jensen et al. (2009). This lattice abstracts program variables that contain integer values. Specifically, $\bot$ denotes that variable is not an integer, $\top$ describes that variable can be any integer, whereas properties `Negative`, `Zero` and `Positive` are self-explanatory. Although this lattice contains a large number of integers (i.e. all possible integers that can be represented by the program), multiple values are abstracted via least upper bound operation ($\sqcup$), e.g. $1 \sqcup 14 =$ `Positive`.

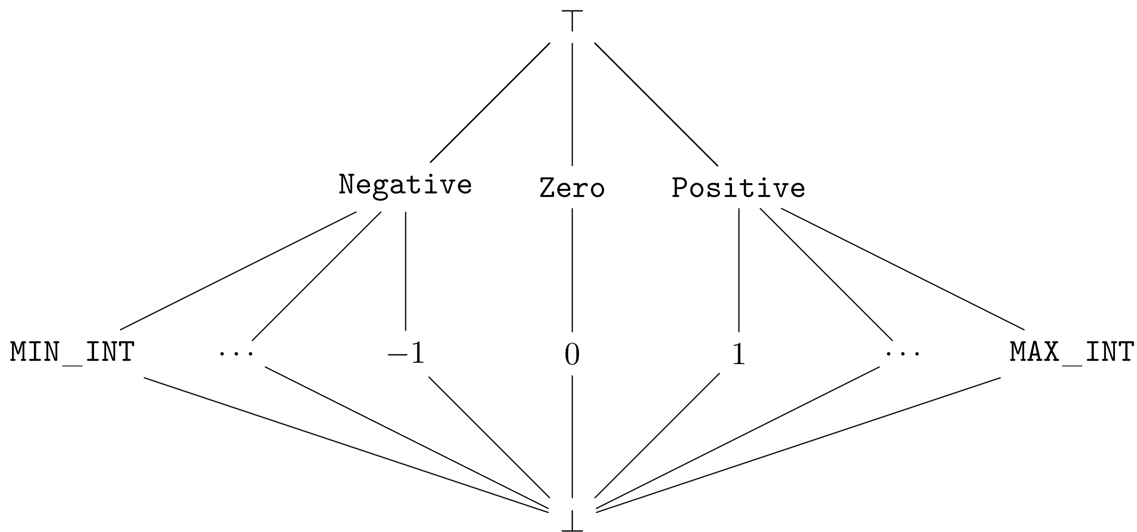


Figure 2.1: An example of a lattice used to abstract integers.

In this context, a simplified abstract heap of a program (whose variables contain only integers) is described by a map as follows:

$$\widehat{Heap} = Variable \rightarrow \widehat{Integer}$$

The semantics of the operations is also abstracted in order to operate on the abstract values. For instance, the abstract multiplication between two `Negative` integers will produce a `Positive`.

Precision & Sensitivities

A common issue that arises in static analysis is its precision. Recall that static analysis techniques approximate the behaviour of the programs, meaning that they may capture properties that are not present during the actual execution or may miss properties that can appear. For example, consider this code fragment:

```
1  if (x > 0) {
2    x = 1;
3    y = x;
4  } else {
5    x = 2;
6  }
```

Using the simplest form of analysis, we would conclude that x is either 1 or 2. In turn, we would propagate this property to variable y after the assignment at line 3. Thus, we would say that y is also either 1 or 2; something which is imprecise if we consider the actual flow of the program. Indeed, y can be 1, but the analysis also reports that y may be 2 which is impossible to happen. If the analysis infers properties that are not present in the actual program (like in the example above), we say that the analysis produces *false positives*. Similarly, if the analysis misses some of the existing properties (e.g. misses that x can be 2), we say that the analysis reports *false negatives*. The number of false positives (or false negatives) is an indicator of the precision of an analysis; the less false positives (or false negatives) an analysis reports, the more precise it is.

In the example above, we silently assumed a *flow-insensitive* analysis, i.e. an analysis that does not take into account the order of program statements. To improve the precision of the analysis of this example (i.e. to infer that `y` can be only 1), the control flow of the program should be considered. In this case, the analysis is called *flow-sensitive*.

There are many different flavours of sensitivity which aim to enhance the precision of the analysis. One common kind of sensitivity that is primarily used in inter-procedural analyses (i.e. analyses that involve more than one functions) is *context-sensitivity* (Emanuelsson and Nilsson, 2008). A context-sensitive analysis takes into account the context (e.g. local variables, arguments, program point of invocation, etc.) on which a method invocation is performed. For example, imagine the following code snippet:

```
1  function foo(x) {
2    if (x > 0) {
3      x = x + 1;
4    } else {
5      x = x - 1;
6    }
7    return x;
8  }
9
10 foo(1);
11 foo(-1);
```

A context-insensitive is not able to distinguish between the different invocations of function `foo()`. In other words, function `foo()` is analysed only once and capture all possible executions coming from lines 10 and 11. Therefore, analysis cannot determine the exact value of parameter `x`, leading to imprecision. A context-sensitive analysis, which uses call-sites (i.e. function's arguments) to distinguish information, inspects function `foo()` twice, i.e. one for `x` = 1, and one for `x` = -1. In this way, analysis gets exact results from `foo()` depending on the context of invocation.

There are many kinds of context-sensitivity. Depending on the characteristics of the programming language on which static analysis is applied, some approaches may suit better than others. For example, it has been shown that object-sensitivity (i.e. analysis separates invocations based on the value of the receiver) performs better than a simple call-site-sensitivity in object-oriented languages (Milanova et al., 2002; Lhoták and Hendren, 2008). Another example comes from the work of Madsen et al. (2015) who designed context-sensitivities specifically targeted for the analysis of event-driven Node.js applications.

Sensitivity is a crucial ingredient of an analysis, because it is a determining factor for the precision and performance (Smaragdakis et al., 2011). Typically, there is a trade-off between performance and precision, i.e. the more precise an analysis is, the more expensive it is (Emanuelsson and Nilsson, 2008). For instance, in our example above, context-sensitivity comes with a cost; we have to analyze function `foo()` more times to get more accurate results. This may be prohibitive in larger and real-word programs. Therefore, it is important to choose the right balance between them so that analysis is neither too imprecise (i.e. produces too many false positives or false negatives) nor too expensive (i.e. does not terminate within a reasonable time). However, it is important to note here that in higher-order programming languages like JavaScript, imprecision may lead to the degradation of the performance (Park et al., 2016). To illustrate this, consider the following code:

```
1 function foo(x) {
2   x("arg");
3 }
4
5 function bar(x) { /* code */ }
6 function baz(x) { /* code */ }
7 function qux(x) { /* code */ }
8
9 // code
```

Imagine an imprecise analysis which reports that parameter `x` may be one of `bar, baz, qux` when it analyses function `foo()`. In this case, the analysis should consider all possible invocations arising from the value of `x` (i.e. `bar`, `baz` and `qux`), when it inspects the call at line 2. A more precise analysis, which correctly determines that parameter `x` of function `foo()` can be *only* `bar`, avoids the analysis of spurious function calls (i.e. `baz` and `qux`).

2.2.2 JavaScript-Specific Analysis

The dominance of JavaScript makes the development of reliable and robust programs more vital than ever (Lee et al., 2012; Kashyap et al., 2014). As described in Section 2.1, JavaScript's dynamic nature such as dynamic update of objects' structure, implicit type conversions, prototype-based inheritance, dynamic code generation or event-driven programming often confuse developers who introduce bugs and security vulnerabilities to their applications. As an effort to provide developers with means, which aim to prevent such bugs, ensure the correctness of the programs and assist developers in their debugging efforts, much work has been done by the community recently, designing new types of analysis specifically targeted for JavaScript (Guarnieri and Livshits, 2009; Jensen et al., 2009; Lee et al., 2012; Wei and Ryder, 2014; Kashyap et al., 2014; Madsen et al., 2015; Davis et al., 2017).

Most notably, over the last ten years, significant progress in static analysis for JavaScript has been made (Guarnieri and Livshits, 2009; Jensen et al., 2009; Lee et al., 2012; Kashyap et al., 2014; Madsen et al., 2015), as new techniques have emerged which are able to handle a wide spectrum of JavaScript's features. As a result, static analysis is considered to be one of the most prevalent techniques developed for analysing JavaScript programs (Sun and Ryu, 2017). Its applications vary from program optimisation and refactoring (Feldthaus et al., 2011; Feldthaus and Møller, 2013) to bug detection and enforcement of security policies (Guarnieri and Livshits, 2009; Jensen et al., 2009; Bae et al., 2014; Kashyap et al., 2014; Madsen et al., 2015).

This advance is also facilitated by the recent attempts to formalise JavaScript. In particular, many researchers have tried to capture the quirky semantics of JavaScript by proposing formal models. For instance, Maffeis et al. (2008) presented one of the first formalisations of JavaScript by designing small-step operational semantics for a subset of the 3rd version of ECMAScript specification. In a subsequent work, Guha et al. (2010) expressed the semantics of 3rd edition of ECMAScript (they omitted `eval` though) through a different approach; they developed a lambda calculus called λ_{JS}, and provided a desugaring mechanism for converting JavaScript code into λ_{JS}. Later, Gardner et al. (2012) introduced a program logic for reasoning about client-side JavaScript programs which support ECMAScript 3. They presented big-step operational semantics on the basis of that proposed by Maffeis et al. (2008), and they introduced inference rules for program reasoning which are highly inspired from separation logic (Reynolds, 2002); a logic that is able to reason about the heap of the pro-

grams. Their rules are expressed through Hoare triples of the form: $\{P\}e\{Q\}$, meaning that "if P holds before the evaluation of e, then e does not produce any errors, and if it terminates, it will do the same when Q holds" (Gardner et al., 2012).

A number of new tools and techniques emerged by exploiting this formal description of JavaScript, including but not limited to analyses for detecting security vulnerabilities (Maffeis and Taly, 2009), techniques for verifying and reasoning about JavaScript programs (Fragoso Santos et al., 2017, 2018), new analyses that are capable of handling a bigger subset of the language (Madsen et al., 2015), etc. For instance, Madsen et al. (2015) extended λ_{JS} to model the event system of server-side applications and designed a static analysis to detect bugs in event-driven Node.js programs based on λ_{JS}. Beyond that, the semantics has provided the foundations for understanding and modelling more recent features of JavaScript (Loring et al., 2017; Madsen et al., 2017).

Techniques and Challenges

Despite its success, static analysis of JavaScript programs still faces a number of challenges which prevent its massive industrial adoption:

Dynamic features and complicated semantics. Static analysis of dynamic languages like JavaScript is not a trivial task. The bizarre features of JavaScript, which were described in previous sections, make the precise capture of the program behaviour extremely difficult (Guarnieri and Livshits, 2009; Jensen et al., 2009; Kashyap et al., 2014). Imprecise modelling of this eccentric semantics dooms analysis to producing meaningless results. On the other hand, attempts to consider all possible executions yield to an analysis that does not terminate. Hence, it is very important to devise the proper abstractions to handle these language features accurately without sacrificing performance.

Many works proposed in recent years have managed to effectively model some of these features. Specifically, Jensen et al. (2009) and Kashyap et al. (2014) abstract values using a tuple of lattices to address the dynamic typing of variables. For example, consider a tuple of lattices $\widehat{Integer} \times \widehat{String} \times \widehat{Bool}$ to abstract variables that may hold integers, strings, and booleans. An abstract value $(Pos, \text{“}foo\text{”}, \bot)$ means that variable may be a positive integer or a string "foo", but it cannot be a boolean. Beyond that, Jensen et al. (2009) models the dynamic update of object properties more precisely by exploiting a technique called recency abstraction (Balakrishnan and Reps, 2006), which minimises the imprecision coming from weak updates. A weak update is performed on statements of the form `x.f = v`, when `x` points to multiple objects. In this case, the new value `v` is joined with the previous values of `f` for all the objects to which `x` points. To address this issue, this technique keeps track of the most recent allocated object on which it performs a strong update, whereas it weakly updates the older objects. Another work of Wei and Ryder (2014) introduces a state-sensitive analysis; a variation of object-sensitive analysis (Recall Section 2.2.1). State-sensitivity is able to distinguish two function calls depending on the state of the receiver (i.e. set of object properties along with their values). Therefore, it improves the precision of the analysis, because every dynamic update, addition or deletion of object properties is tracked. For instance, consider the following code snippet (Wei and Ryder, 2014):

```
1  function Foo() { /* code */}
2  function Bar() { /* code */}
3
4  var x = {
5    baz: function(u, v) {
6      u.f = v;
7    },
8    f: new Foo()
9  }
10
11 x.bar(x.f, 10);
12 x.f = new Bar();
13 x.bar(x.f, 11);
```

Using a state-sensitive analysis, the two function calls at lines 11, 13 are separated, because the state of receiver `x` is changed at line 12. As a result, the analysis performs a strong update at line 6 for both calls, since property `f` points to a unique object when function `bar` is invoked (lines 11, 13).

An alternative approach followed by many works such as those of Guarnieri and Livshits (2009) and Kashyap et al. (2014) is to neglect features that are tricky to statically analyse. This approach is closely related to an emerging term in static analysis called *soundiness* (Livshits et al., 2015). Soundy analyses choose to purposely bypass some programming features, (e.g. because they do not appear very often), but they soundly analyse the remainder. Conceptually, they combine both soundness and unsoundness, aiming to perform effectively on the subset of the language. For example, Guarnieri and Livshits (2009), who proposed one of the first pointer analyses for JavaScript, precluded the use of `eval`-family functions from their analysis. Their analysis focused on widgets where the use of `eval` is not common, even though in other real-world applications, the use of such language constructs is prevalent (Richards et al., 2011). Also, Guarnieri and Livshits (2009) illustrated that a combination of dynamic features may harm the effectiveness of the analysis; therefore, extra care should be given. For example, simply restricting the explicit use of `eval` function does not prevent the dynamic use of it, as illustrated by the following piece of code (Guarnieri and Livshits, 2009):

```
1  var foo = this["eval"];
2  foo("inject code here");
```

Here, `this` points to the global object, and through a dynamic property lookup, `eval` function is loaded to variable `foo`. To this end, they instrumented runtime code to prevent such cases if the name of the property cannot be resolved statically.

Other works employed different approaches to handle such dynamic features. For instance, Park et al. (2013) developed a mechanism for converting programs that use `with` statement into with-free programs. Another example comes from the work of Jensen et al. (2012) who proposed a semantics-preserving program transformation to eliminate calls of `eval` function.

Environment and Libraries. JavaScript applications reside in a complex environment and they make a pervasive use of libraries and frameworks. For instance, client-side applications interact with HTML document and browser using the DOM and BOM API respectively and they receive inputs from user via events, e.g. mouse click, mouse move, etc. The code of these APIs is not available to the analyser, as they have been implemented natively. At the same time, applications often use complicated libraries like jQuery that hinder the perfor-

mance of the analysis or analyser can reason about. A simple workaround to analyse such applications is to create stubs for library and environment functions (Guarnieri and Livshits, 2009). These stubs reflect all possible results derived from the invocation of these functions, but the problem with this approach is the high rate of inaccuracy.

The most widespread method to deal with these issues, which is employed by many static analysers, is to provide models (Jensen et al., 2011; Kashyap et al., 2014; Park et al., 2016; Madsen et al., 2015). In general, modelling is a demanding and error-prone task though, which may harm the soundness of the analysis (Guarnieri and Livshits, 2009; Park et al., 2016). For example, designers have to decode vague and difficult to understand specifications (e.g. ECMAScript, DOM specification, etc.) in order to implement models properly (Park et al., 2016). However, modelling has a positive impact on the precision and the scalability of the analysis, if it is applied correctly (Park et al., 2016).

Much progress has been made for modelling the environment of client-side applications. Jensen et al. (2011) models HTML DOM by creating an hierarchy of abstract states which reflect the actual HTML object hierarchy. For instance, a `HTMLFormEelement` abstract object contains properties like `elements`, `action`, `method`, etc., and derives other properties from the hierarchy chain, e.g. from its parent `HTMLElement` and so forth. Before analysis begins, an initial heap is constructed which contains the set of the abstract objects corresponding to the HTML code of the page. Park et al. (2016) follow a similar approach for modelling HTML DOM. They also provide a more precise model which respects the actual tree hierarchy of the DOM. For example, their model distinguishes the two different `div` nodes at the following case, as the one is nested to another.

```
1 <div>
2   <div></div>
3 </div>
```

A semi-automatic approach to model APIs is proposed by Bae et al. (2014). Specifically, they automatically model libraries by taking advantage of APIs specification written in Web IDL[2]. Their approach first analyses APIs specification, and extracts information about functions' signature i.e. return type. Based on this specification, they model function calls by returning an abstract object with regards to the return type pointed by the specification. For example, if specification states that function `foo` returns a value of type `Bar`, their analysis generates an abstract object whose structure follows that of `Bar`. Note that every type, that is, the structure of the objects, (e.g. the set of properties that an object returned by the function must contain), is also described in the specification.

Madsen et al. (2013) sacrifice soundness for scalability. They technique they propose is able to analyse JavaScript applications even if they operate in complex environment and interact with native or large libraries. They employ a combination of a pointer and a use analysis for determining the values that an API call might return. Their analysis is flow- and context-insensitive and is evaluated on benchmarks that consist of up to 30,000 lines of library code. Their work demonstrate that soundness is not always a requirement for designing analysis that is both scalable and precise.

[2] `https://heycam.github.io/webidl/`

Tools

There are many static analysis frameworks developed for JavaScript. We discuss the most notable ones.

TAJS (Jensen et al., 2009) is a static analyzer for JavaScript which implements a classical dataflow analysis using monotone frameworks (Kam and Ullman, 1977) which is aimed to be sound. It uses a special lattice designed for JavaScript which handles the vast majority of JavaScript's features and its complicated semantics. Analysis is both flow- and context-sensitive. Initially, it begins on top of an approximated control-flow graph (CFG) which describes the intra-procedural flow of the program. Gradually, the inter-procedural flow is constructed on the fly when analysis encounters function calls or exception statements. Every node of CFG corresponds to a transfer function which computes the next abstract state resulted by the execution of this node. The aim of the analysis is to compute all reachable states given an initial state. By inspecting every state, TAJS is able to detect various type errors such as, use of a non-function variable as a function, property access of `null` or `undefined` variables, inconsistencies caused by implicit type conversions, and many others (Jensen et al., 2009). At the end of the analysis, TAJS determines the call graph of the program as well as an abstract representation of the heap.

TAJS has a good support for client-side JavaScript applications, as it provides complete models for DOM and BOM APIs (Jensen et al., 2011) and has been evaluated on many different versions of jQuery (Andreasen and Møller, 2014); its support for Node.js applications is limited though.

SAFE (Lee et al., 2012) is static analysis framework, which provides three different representations of JavaScript, that is, an AST, an IR and a CFG. This facilitates users to design and implement client analyses on the right level of abstraction. For instance, a client analysis that aims to find duplicate JavaScript code operates on AST (Lee et al., 2012). For this reason, analysis is implemented in phases: parsing (source code to AST), compilation (AST to IR), and building of CFG (IR to CFG). This modular architecture allows users to add extensions at whichever phase they want. SAFE implements a default analysis phase which is plugged after the construction of CFG. This analysis adopts a similar approach with that of TAJS, i.e. a flow- and context-sensitive analysis which operates on top of CFG. Similar to TAJS, SAFE provides models for DOM and BOM APIs (Park et al., 2016). It has been also used for detecting API misuses of platform libraries (Bae et al., 2014).

JSAI (Kashyap et al., 2014) implements an analysis through abstract interpretation framwork (Cousot and Cousot, 1977). Specifically, it employs a different approach compared to other existing tools. Unlike TAJS and SAFE, JSAI operates on top of AST rather than CFG; it is flow-sensitive though. To achieve this, abstract semantics is specified on a CESK abstract machine (Felleisen and Friedman, 1987), which provides small-step reduction rules and an explicit data structure (i.e. continuation) which describes the rest of computation, unwinding the flow of the program in this way. The rationale behind this design decision is that CFG of a JavaScript program is not known apriori, and cannot be computed precisely, leading to imprecision (Kashyap et al., 2014). JSAI also supports concrete semantics which were tested against a large number of JavaScript applications and have been proven to be sound (Kashyap et al., 2014). Analysis is configurable with different flavours of context-sensitivity which are plugged into the analysis through widening operator used in the fix-

point calculation (Hardekopf et al., 2014). Finally, JSAI provides partial models for DOM and browser APIs, and presents a proof of concept client analysis which simply counts the number of type errors in a program.

2.3 The Asynchronous JavaScript

Indisputably, the major domain of JavaScript is web applications (both client- and server-side). One of the non-functional requirements of this kind of applications is that they should be highly responsive in order to be able to serve a large number of requests and provide the best possible experience to their users. This requires that time-consuming operations do not block the execution of the main flow of the program. For example, when a JavaScript client-side code issues an HTTP request to a server, web UI should not be blocked: it should operate normally and allow user to perform other tasks. In programming languages like Java or C, one standard way to improve throughput in an application and run multiple expensive tasks in the background is using threads. However, JavaScript differs from these languages, as it adopts a single-threaded execution model. JavaScript achieves high throughput by embracing an event-driven programming paradigm. Specifically, code, namely, *callback*, is executed in response to events (e.g. mouse click, response arrived from a server, file read from disk, etc.). Also, an expensive task like an I/O operation (e.g. request to a server, etc.) is *asynchronous*, meaning that execution is not interrupted, waiting the operation to terminate. On the contrary, execution proceeds to the next tasks and programmer registers callbacks which are invoked later in the future when the expensive operation is complete. As we will see later, this asynchrony is the source of many concurrency errors due to the non-deterministic execution of asynchronous operations and event handling (Wang et al., 2017).

This section is dedicated to asynchrony of JavaScript applications. First, we present the main sources of asynchrony in JavaScript applications. Then, we illustrate how JavaScript engines like Node.js implements asynchrony. Finally, we discuss the various concurrency bugs which stem from asynchrony and what kind of analyses have been developed to detect such bugs.

2.3.1 Sources of Asynchrony

JavaScript heavily relies on callbacks to achieve asynchrony. Callbacks are just functions that are passed as arguments to other functions. According to a study of Gallaba et al. (2015), more than half of the callbacks found in a program are executed asynchronously which means that are *not* processed the time when they are passed to the callee function. The main sources where this asynchronous execution occurs in JavaScript applications are I/O operations, events, promises and timers.

I/O Operations

When a JavaScript program executes an I/O operation, it processes it asynchronously. Typical examples of I/O operations are network operations (e.g. HTTP requests to a server), or file system operations (e.g. read and write to/from a file). A popular example comes from client-side JavaScript as follows:

```
1 var request = new XMLHttpRequest();
2 request.onreadystatechange = function () {
3   // Code is executed when we receive the response the from server.
4 };
```

```
5 request.open("GET", "www.example.com");
6 request.send();
7 // other code
```

In this example, we issue an AJAX GET request to 'www.example.com'. When we submit this request to the server (line 6), the execution is not interrupted, but it continues processing the next code. For this reason, we register a callback function at line 2, which is called when client receives the response from the server.

The same pattern is used when we read a file in a Node.js application as follows:

```
1 var fs = require("fs");
2
3 fs.readFile("myfile", function (err, data) {
4   // Code is executed when read of file is completed.
5 });
6 // other code
```

Again, program reads file 'myfile' asynchronously, and when read is complete, the callback passed by the programmer at line 3 is executed.

Events

As it is already mentioned, developers can add callback functions in response to events coming from the external environment, e.g. mouse clicks, keyboard typing, etc. In particular, when an event is triggered from the interaction of a user with DOM, JavaScript engine executes the corresponding callback asynchronously. For example, consider this code:

```
1 <script>
2 function foo() {
3   // code
4 }
5 </script>
6 <button id="bar" onclick="foo()">Foo</button>
```

If user clicks the button with id "bar", callback `foo()` will be invoked asynchronously, i.e. at some point later in the future.

However, note that triggering an event through `dispatchEvent()` of client-side JavaScript (MDN, 2018) or `emit()` of Node.js (v10.3.0 Documentation, 2018) results in the *synchronous* execution of the registered callbacks. For example, imagine the following code:

```
1 var emitter = require("events");
2
3 var foo = new emitter();
4 foo.on("bar", function() {
5   // This code is executed synchronously
6   // when event 'bar' is triggered.
7 })
8 foo.emit("bar");
9 // other code
```

At line 4, we register a callback as a response to event "bar". When we trigger event "bar" at line 8, callback is executed synchronously, and thus, execution proceeds to the next code only if callback terminates.

Promises

The 6th edition of ECMAScript specification (ECMA-262, 2015) introduced a new language construct which is inherently related to asynchronous computation (Madsen et al., 2017), namely, *promises*. Promises are used to describe the state of (asynchronous) operations. According to ECMAScript specification (ECMA-262, 2015), the state of a promise object can be one of:

- *fulfilled:* the associated operation has been completed and the promise object tracks its resulting value.
- *rejected:* the associated operation has failed and the promise object tracks its erroneous value.
- *pending:* the associated operation has been neither completed nor failed.

Promises are particularly useful for asynchronous programming, because they facilitate the creation of chains of asynchronous computation (Madsen et al., 2017). One problem that arises with asynchronous operations and callbacks as presented previously is that it is not intuitive to the programmer how to enforce a sequence of such operations. For instance, imagine that we need to call an asynchronous function `async1`. After its completion we need to call another asynchronous function `async2`, and so forth. The standard way to enforce this order of computation is through nested callbacks as follows:

```
1  async1(function() {
2    // callback of async1
3    async2(function() {
4      // callback of async2
5      async3(function() {
6        // callback of async3
7        ..
8      })
9    })
10 });
```

However, this approach engenders a deep nesting which poses a number of maintainability issues (i.e. callback hell) (Gallaba et al., 2015, 2017).

Programmer can intuitively create such sequence of operations by calling `then()` of a promise object. This function receives two callbacks that are called asynchronously once the receiver object has been either fulfilled or rejected. These callbacks get one argument which is the value based on which promise is fulfilled or rejected. Also, `then()` allocates a new promise whose value with which is fulfilled or rejected depends on the return value of the callback registered during the invocation of `then()`. In this way, programmer enforces the order of operation by constructing a chain of promises as shown in Figure 2.2.

The code snippet above illustrates how we achieve an execution order between multiple asynchronous operations via promises. When an asynchronous operation finishes its job, it resolves the promise using the function `resolve()` (lines 3, 10, 16). The resolved value is propagated to the next operation through the argument of the callback passed to `then()` (i.e. `value` in our example). It is guaranteed that `async1` is executed before `async2`, and `async2` before `async3`, because the callbacks passed to `then()` are called only when the promise is fulfilled, i.e. the asynchronous operations associated with the promise is complete. Chaining promises also facilitates error propagation and error handling; something that is extremely

```
1  var promise = new Promise(function(resolve, reject) {
2    async1(function() {
3      resolve("async1 completed");
4    })
5  });
6
7  promise.then(function(value) {
8    return new Promise(function(resolve, reject) {
9      async2(function() {
10       resolve("async2 completed");
11     });
12   });
13 }).then(function(value) {
14   return new Promise(function(resolve, reject) {
15     async3(function() {
16       resolve("async3 completed");
17     });
18   });
19 });
```

Figure 2.2: An example of a promise chain.

difficult to be expressed through nested callbacks (Gallaba et al., 2017; Madsen et al., 2017).

A typical way to initialise a new promise is via promise constructor. The promise constructor creates a fresh promise, and expects a function which is named *promise executor*. Promise executors receive two internal functions as arguments, (i.e. they are not transparent to the developers), which are used to fulfill and reject the created promise respectively.

Timers

JavaScript also provides some mechanisms to schedule the execution of some code at some point later in the future. For example, function `setTimeout()` is used to schedule the execution of a callback when some period of time elapses, whereas function `setInterval()` executes a callback periodically. Note again that these callbacks are executed asynchronously, therefore, in the following code, the callback of `setTimeout()` is called after `someExpensiveJob()` finishes doing its work, even though it is scheduled for execution after 0 milliseconds.

```
1  setTimeout(function() {
2    // Code will be executed after
3    // someExpensiveJob() is complete
4  }, 0);
5  someExpensiveJob();
```

2.3.2 Runtime Architecture

JavaScript executes all the asynchronous callbacks through the *event loop*. The high-level idea of the event loop is that it is "perpetually" waiting for new events to come, and when an event arrives, the event loop synchronously executes its callbacks via a single thread. Figure 2.3 demonstrates the main architecture of the event loop. When the execution of all top-level code is complete, event loop fires and waits for events to be triggered. When an event takes place, JavaScript puts its associated callbacks (if any) into an event queue. In

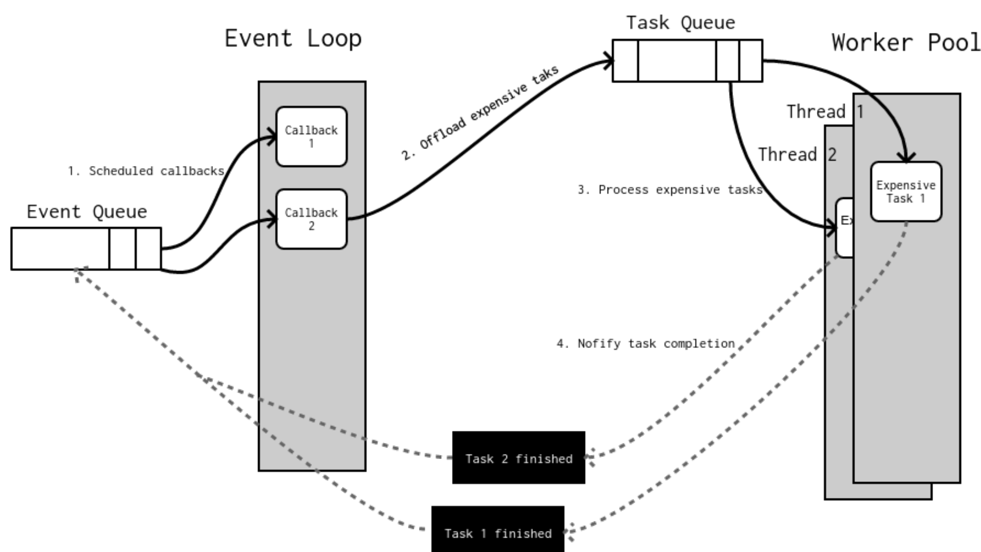


Figure 2.3: High-level architecture of the event loop (Wang et al., 2017; Davis et al., 2017). Event loop fetches scheduled callbacks from event queue (1) and processes them sequentially. Time consuming callbacks are assigned to a different queue (2) maintained by the worker pool. In turn, worker pool executes them concurrently via multiple threads (3), and upon completion, it notifies event loop via special events (4).

turn, event loop dequeues one callback at a time, and executes it using one single thread in a synchronous manner. Note that the execution of every callback is atomic, therefore it cannot be interleaved with other callbacks.

Since all callbacks in event loop are executed synchronously, there is an extra component in this architecture to avoid blocking the event loop due to the execution of time-consuming callbacks. Examples of such expensive callbacks are I/O operations (e.g. DNS queries, filesystem operations), or CPU-bound operations (e.g. crypto modules) (Node.js, 2018a). Specifically, when an expensive callback comes to place, the event loop defers its execution, and assigns it to a different queue (namely, task queue) maintained by the *worker pool*. Unlike event loop, execution inside the worker pool is multi-threaded, consisting of k threads (Node.js, 2018a). Thus, the worker pool fetches expensive tasks and executes them concurrently. After their execution, the worker pool notifies the event loop of their completion, through special events whose callbacks are again placed in the event queue.

Note that different JavaScript execution engines may use different policies for processing the scheduled callbacks at step (1). For example, client-side JavaScript processes load callbacks before any other callback (Jensen et al., 2011). On the other hand, Node.js executes callbacks through six different phases in a round robin fusion, e.g. it processes all callbacks related to phase for a while, it then proceeds to the next phase, and so on (Loring et al., 2017; Wang et al., 2017; Node.js, 2018b).

2.3.3 Concurrency Bugs

Although execution is single-threaded, concurrency issues still exist in JavaScript programs because of the non-deterministic processing of callbacks from event loop (Wang et al., 2017;

Davis et al., 2017). Specifically, even though some JavaScript engines support some sort of scheduling policies (Loring et al., 2017; Wang et al., 2017; Node.js, 2018b), in general, event loop processes callbacks non-deterministically (Davis et al., 2017). Also, recall from Figure 2.3 that worker pool processes time-consuming tasks concurrently, thus, there is not guarantee about the completion order of these tasks.

This non-determinism instigates a number of challenges to the developers. The misuse of the asynchrony-related features of JavaScript and the wrong assumptions about the execution order of callbacks can cause a number of concurrency bugs. Several previous studies (Zheng et al., 2011; Hong et al., 2014; Wang et al., 2017; Davis et al., 2017) have classified these bugs into two major categories based on their pattern: 1) *order violation*, and 2) *atomicity violation*.

Order Violation. An order violation occurs when two operations are supposed to be executed in a specific order, but this order is not guaranteed by the runtime. For example, consider a classical example already presented in the literature (Madsen et al., 2015):

```
1  var fs = require("fs");
2
3  fs.writeFile("myfile", "data", function(err, data) {
4    // Callback of writeFile()
5  });
6  ..
7  fs.stat("myfile", function(err, stats) {
8    // Callback of stat()
9  });
```

In this code snippet, the intention of the programmer is to create a new file named ‘myfile’, and at future point, they want to retrieve the stat information of this file using function `stat()`. However, the functions `writeFile()`, and `stat()` are asynchronous, and their execution order is not deterministic. If `stat()` is executed before `writeFile()`, programmer will receive a “File not found” error. However, recall that programmers can enforce the right sequence of operations, via nested callbacks or promises (Section 2.3.1).

Atomicity Violation. An atomicity violation happens when two operation are supposed to be atomic, (i.e. the intention is to be processed together without the interleaving of any other operation) but this atomicity is not enforced. An example of a code which is prone to atomicity violation is the following:

```
1  var x;
2  var data = {};
3
4  function foo() {
5    x += 1;
6    someAsyncOperation(function(asyncData) {
7      data[x] = asyncData;
8    });
9  }
```

The program above maintains two global variables, that is, a counter `x`, and an object `data`. The program also has a function `foo()`. When it is called, it fist increments the counter by one (line 5), and then performs an asynchronous operation (line 6). When this asynchronous operations terminates, its callback puts the resulting data to the global object `data`

based on the value of the counter `x` (line 7). The callback of the asynchronous operation and function `foo()` are not atomic though. Most notably, if we call function `foo()` twice, it can be the case that the update of counter `x` performed by the second call at line 5 precedes the execution of the callback of the first call. In other words, we have the following erroneous sequence of operations: $foo_1 \rightarrow foo_2 \rightarrow callback_1 \rightarrow callback_2$, whereas the indented order is: $foo_1 \rightarrow callback_1 \rightarrow foo_2 \rightarrow callback_2$. The consequence of the wrong order is that we eventually miss the data coming from the first callback.

Notice that a notable difference between order and atomicity violations is that the former need only one call to manifest themselves, while the latter require multiple calls (recall, we assumed that function `foo()` is invoked twice) (Davis et al., 2017). Also, the recent studies of Wang et al. (2017) and Davis et al. (2017) showed that atomicity violations is the most prevalent bug pattern in Node.js applications.

Other bug patterns. (Wang et al., 2017) also identified one more bug pattern which is caused by the scheduling policies implemented in Node.js event loop. The name of this bug is *starvation* and it occurs when a callback or a task blocks the execution of other callbacks. For instance, consider the following piece of code (Loring et al., 2017):

```
1  var fs = require("fs");
2
3  function foo() {
4    process.nextTick(foo);
5  }
6  fs.writeFile("myfile", "data", function() {
7    // Never executed.
8  });
9  foo();
```

When we execute `foo()`, `process.nextTick()` schedules its callback immediately regardless of the presence of other callbacks in the event queue. Since we recursively re-schedule `foo()` via `process.nextTick()`, callback of `writeFile()` will never be executed due to the high priority of `process.nextTick()`.

2.3.4 Detecting Concurrency Bugs

The consequences of such concurrency bugs are pernicious for the vast majority of applications (Wang et al., 2017; Davis et al., 2017). Specifically, these bugs can have a really negative impact on the reliability of the software: they lead to software crashes, incorrect outputs, and inconsistent states (Zheng et al., 2011; Wang et al., 2017; Davis et al., 2017). Even client-side applications, where a large number of these bugs result in innocent UI errors, are often negatively affected if these bugs concern browser storage (Mutlu et al., 2015). Hence, many techniques have been developed as an effort to detect such bugs in JavaScript programs.

Client-side JavaScript. The initial focus of researchers was on detecting concurrency bugs on client-side JavaScript. Zheng et al. (2011) proposes one of the first bug detectors which employs a static analysis for identifying concurrency issues in asynchronous AJAX calls. The aim of their analysis is to detect data races between the code which pre-processes an AJAX request and the callback invoked when the response of the server is received. In other words, if there is an update of a global variable (or a variable which is dependent on a global) in

the code which pre-processes the request, and the same variable is used in the response callback, their analysis reports a possible error. Their approach took advantage of call graphs produced by other core analyses, and on top of that, they defined Datalog rules in order to infer data races.

In a subsequent work, Petrov et al. (2012) have adopted a dynamic analysis to detect data races in web applications. They first proposed a *happens-before* relation model, to capture the order of execution between different operations that are present in a client-side application. Example of such operations are load of HTML elements, execution of scripts, event handling and callback of timers, i.e. `setTimeout()`, `setInterval()`. Their model defines precise relations between those operations. For example, two inline scripts (i.e. their code is inside the HTML page) are executed in the order they appear. Using this model, their analyses reports data races, if two operations access the same variable (at least one of them writes to it), and there is not any happens-before relation between them. However, their approach introduced a lot of false positives, at the most data races did not lead to severe concurrency bugs.

Mutlu et al. (2015) has combined both a dynamic and static analysis and primarily focused on detecting data races that have severe consequences on the correctness of applications, reporting data races that mainly affect browser storage. Initially, they collect the execution traces of an application, and then, they apply a dataflow analysis on these traces which inspects different executions of event callbacks or asynchronous operations, targeting to find data races. Their analysis keeps a state which is map of variables' locations to their values. The states derived from the multiple executions of a callback are merged. If a sensitive location (e.g. cookie) contains multiple values, then their tools flags this location as a data race, because its value is dependent on the schedule of the callback (i.e. different schedules produced different values). Their approach effectively managed to report a very small number of false positives (Mutlu et al., 2015).

Server-side JavaScript. Little work has been done so far regarding the development of concurrency bug detection tools for server-side applications (Wang, 2017). A very recent tool is proposed by Davis et al. (2017), which builds `Node.fz`; a fuzzer for Node.js applications. The key idea of their approach is to shuffle the callbacks located at the event and task queues of event loop and worker pool respectively (Recall Figure 2.3). In this way, they emulate different sequences of arrivals of events and they also explore different execution orders of tasks assigned to the worker pool. `Node.fz` does not produce false positives, however, it may miss possible bugs as its effectiveness is closely dependent on the input data of the application.

Building tools for detecting concurrency bugs in this kind of applications is a promising area though, because recent studies pointed out the high severity of these bugs(Wang et al., 2017; Davis et al., 2017). Beyond that, the works of Loring et al. (2017) and Madsen et al. (2017), who presented new JavaScript semantics for modelling event loop and asynchronous operations, give a lot of opportunities for better reasoning about JavaScript's asynchrony and building new analyses and tools on top of them.

Chapter 3

Modelling Asynchrony

The focus of our work is to develop a static analysis technique which is able to handle and reason about the asynchrony in JavaScript. As a starting point, we first need to define a model for the asynchronous constructs found in JavaScript such as promises, timers and asynchronous I/O. The goal of this model is to provide us with the foundations for gaining a better understanding of these features and ease the design of a static analysis for asynchronous JavaScript programs. This model is expressed through a calculus called $\lambda_{\texttt{q}}$; an extension of the core calculus $\lambda_{\texttt{js}}$ for JavaScript developed by Guha et al. (2010). Note that our model is inspired from the previous works of Loring et al. (2017); Madsen et al. (2017) who also extended $\lambda_{\texttt{js}}$ to deal with asynchronous primitives and promises respectively. Contrary to their works, our model keeps track the effects of uncaught exceptions when they are triggered in the execution of callbacks. The $\lambda_{\texttt{q}}$ calculus is the touchstone of the analysis introduced in Chapter 4, where we devise the domains and the semantics that over-approximates our model. Note that our calculus $\lambda_{\texttt{q}}$ is designed to model almost all asynchronous primitives of JavaScript up to the 7th edition of the ECMAScript specification (i.e. promises, timers, asynchronous I/O). However, we do not handle `Promise.all()` function, and `async/await` keywords which were introduced in the 8th version of ECMAScript (ECMA-262, 2018).

In this chapter, we first introduce the $\lambda_{\texttt{q}}$ calculus; a variation of $\lambda_{\texttt{async}}$ and $\lambda_{\texttt{p}}$ (Loring et al., 2017; Madsen et al., 2017). We present its syntax, its concrete domains, and its semantics (Section 3.1). Then, we provide models for our asynchronous constructs in terms of $\lambda_{\texttt{q}}$ (Section 3.2). Finally, we highlight the main differences of $\lambda_{\texttt{q}}$ with the existing models (Section 3.3).

3.1 The $\lambda_{\texttt{q}}$ calculus

Our calculus, as the previous works of Loring et al. (2017); Madsen et al. (2017), introduces new special language elements for treating asynchrony. The key component of our model is called *queue object* and it is closely related to JavaScript promises. Specifically, a queue object (like a promise) tracks the state of an asynchronous job, and it can be in one of the following states: 1) *pending*, 2) *fulfilled* or 3) *rejected*. A queue object may have registered callbacks which are executed depending on its state, i.e. there are two distinct lists for storing callbacks: the first one holds the callbacks which are triggered once the queue object is fulfilled, while the second list contains the callbacks which are called upon the rejection of the queue object. Initially, a queue object is in a pending state. A pending queue object can transition to a fulfilled or a rejected queue object. A queue object might be fulfilled or

rejected with a given value which is later passed as argument in the registered callbacks. Once a queue object is either fulfilled or rejected, its state is final and cannot be changed. We keep the same terminology as promises, so if a queue object is either fulfilled or rejected, we call it *settled*.

In the following sections, we first present the concrete domains and the syntax of $\lambda_{\mathtt{q}}$ which introduces new expressions for dealing with the state of queue objects. In turn, we describe the semantics of the new language constructs.

3.1.1 Syntax and Domains

$$
\begin{array}{lcl}
v \in Val & ::= & \ldots \\
 & | & \bot \\
\\
e \in Exp & ::= & \ldots \\
 & | & \texttt{newQ()} \\
 & | & e.\texttt{fulfill}(e) \\
 & | & e.\texttt{reject}(e) \\
 & | & e.\texttt{registerFul}(e, e, \ldots) \\
 & | & e.\texttt{registerRej}(e, e, \ldots) \\
 & | & \texttt{append}(e) \\
 & | & \texttt{pop()} \\
 & | & \bullet \\
\\
E & ::= & \ldots \\
 & | & E.\texttt{fullfill}(e) \mid v.\texttt{fulfill}(E) \\
 & | & E.\texttt{reject}(e) \mid v.\texttt{reject}(E) \\
 & | & E.\texttt{registerFul}(e, e, \ldots) \mid v.\texttt{registerFul}(v, \ldots, E, e, \ldots) \\
 & | & E.\texttt{registerRej}(e, e, \ldots) \mid v.\texttt{registerRej}(v, \ldots, E, e, \ldots) \\
 & | & \texttt{append}(E) \\
\\
u \in Fun & = & \text{the set of functions} \\
\\
f \in F & = & Fun \cup \{\texttt{defaultFulfill}, \texttt{defaultReject}\}
\end{array}
$$

Figure 3.1: Syntax and evaluation contexts of $\lambda_{\mathtt{q}}$.

Figure 3.1 illustrates the syntax of $\lambda_{\mathtt{q}}$. For brevity, we present only the new constructs added to the language. Specifically, we add eight new expressions:

- `newQ()`: This expression creates a new queue object in a pending state with no callbacks associated with it.
- $e_1.\texttt{fulfill}(e_2)$: This expression fulfills the receiver (i.e. expression e_1) with the value of e_2.
- $e_1.\texttt{reject}(e_2)$: This expression rejects the receiver (i.e. expression e_1) with the value of e_2.

- e_1`.registerFul`$(e_2, e_3, \dots)$: This expression registers a new callback e_2 to the receiver. This callback is executed only when the receiver is *fulfilled*. This expression also expects another queue object which is passed as the second argument, i.e. e_3. This queue object will be fulfilled with the return value of callback e_2. This allows us to model chains of promises where a promise is resolved with the return value of the callback of another promise (Recall Section 2.3.1). This expression also expects optional parameters (i.e. expressed through "...") with which e_2 is called if the queue object is fulfilled but not with a certain value.
- e_1`.registerRej`$(e_2, e_3, \dots)$: The same as e`.registerFul`$(\dots)$ but this time the given callback is executed once the receiver is *rejected*.
- `append`(e): This expression appends queue object e to the top of the current queue chain. As we shall see later, the top element of a queue chain corresponds to the queue object that is needed to be rejected if the execution encounters an uncaught exception.
- `pop()`: This expression pops the top element of the current queue chain.
- The last expression $\bullet$ stands for the event loop.

Observe that we use evaluation contexts (Felleisen et al., 2009; Guha et al., 2010; Madsen et al., 2015; Loring et al., 2017; Madsen et al., 2017) to express how the evaluation of an expression proceeds. The symbol E denotes which sub-expression is currently being evaluated. For instance, E`.fulfill`(e) describes that we evaluate the receiver of `fulfill`, whereas v`.fulfill`(E) implies that the receiver has been evaluated to a value v, and the evaluation now lies on the argument of `fulfill`. Beyond those expressions, the syntax of $\lambda_{\mathtt{q}}$ introduces two functions, namely, `defaultFulfill` and `defaultReject`. The semantics of these functions is presented in the following section. Also, $\lambda_{\mathtt{q}}$ introduces a new value, that is, $\perp$. This value differs from `null` and `undefined`, and is used to express the absence of value.

$$
\begin{aligned}
a \in \mathit{Addr} &= \{l_i \mid i \in \mathbb{Z}^*\} \cup \{l_{time}, l_{io}\} \\
\pi \in \mathit{Queue} &= \mathit{Addr} \hookrightarrow \mathit{QueueObject} \\
q \in \mathit{QueueObject} &= \mathit{QueueState} \times \mathit{Callback}^* \times \mathit{Callback}^* \times \mathit{Addr} \\
s \in \mathit{QueueState} &= \{\mathtt{pending}\} \cup (\{\mathtt{fulfilled}, \mathtt{rejected}\} \times \mathit{Val}) \\
\mathit{clb} \in \mathit{Callback} &= \mathit{Addr} \times F \times \mathit{Val}^* \\
\kappa \in \mathit{ScheduledCallbacks} &= \mathit{Callback}^* \\
\kappa \in \mathit{ScheduledTimerIO} &= \mathit{Callback}^* \\
\phi \in \mathit{QueueChain} &= \mathit{Addr}^*
\end{aligned}
$$

Figure 3.2: Concrete domains of $\lambda_{\mathtt{q}}$.

Figure 3.2 presents the domains introduced in the semantics of $\lambda_{\mathtt{q}}$. In particular, a queue is a partial map of addresses to queue objects. An address is indicated by symbol l_i, where i is a positive integer. Notice that the set of the addresses also includes two special reserved addresses, i.e. l_{time}, l_{io}, which are used to store the queue objects responsible for keeping the state of callbacks related to timers and asynchronous I/O respectively (Section 3.2 explains

how these JavaScript features are modelled). A queue object is described by its state (recall that a queue object is either pending, fulfilled with a value or rejected with a value), a sequence of callbacks which are executed on fulfillment, and a sequence of callbacks which are called on rejection. The last element of a queue object is an address which corresponds to another queue object which is dependent on the current, i.e. they are settled whenever the current queue object is settled, and with exactly the same state. We create such dependencies when we settle a queue object with another queue object. In this case, the receiver is dependent on the queue object used as argument.

Moving to the domains of callbacks, we see that an element of that domain is a tuple of an address, a function to be executed, and a list of values (i.e. arguments of function). Note that the first component of this domain denotes the address of the queue object which is going to be fulfilled with the return value of the function. In a list of callbacks, namely $\kappa \in ScheduledCallbacks$, we keep the order in which callbacks are scheduled. Note that we maintain one more list of callbacks (i.e. $\tau \in ScheduledTimerIO$) where we store callbacks registered in the queue objects located at the special addresses l_{time}, l_{io}. We defer the discussion about why we keep two separate lists until Section 3.1.3.

A queue chain $\phi \in QueueChain$ is a sequence of addresses. In a queue chain, we store the queue object that is needed to be rejected if there is an uncaught exception in the current execution. Specifically, when we encounter an uncaught exception, we inspect the top element of the queue chain, and we reject it. If the queue chain is *empty*, we propagate the exception to the call stack as normal. Recall from Figure 3.1 that there is a special syntax for adding and removing elements from the queue chain explicitly.

3.1.2 Semantics

Equipped with the appropriate definitions of the syntax and domains, in Figure 3.3, we present the small-step semantics of $\lambda_{\texttt{q}}$ which is an adaptation of (Loring et al., 2017; Madsen et al., 2017). Note that we demonstrate the most representative rules of our semantics; some rules are omitted for brevity. The reader is referred to Appendix A for the rest of the rules of $\lambda_{\texttt{q}}$. For what follows, the binary operation denoted by symbol $\cdot$ means the addition of an element to a list, while the operation indicated by $::$ stands for list concatenation.

The rules of our semantics adopt the following form:

$$\pi,\ \phi,\ \kappa,\ \tau,\ E[e] \rightarrow \pi',\ \phi',\ \kappa',\ \tau',\ E[e']$$

That form expresses that a given queue π, a queue chain ϕ, two sequences of callbacks, (i.e. κ, τ), and an expression e in evaluation context E lead to a new queue π', a new queue chain ϕ', two new sequences of callbacks (i.e. κ' and τ' respectively), and a new expression e' in the same evaluation context E, assuming that the expression e is reduced to e' (i.e. $e \hookrightarrow e'$). This behaviour is described by the `[e-context]` rule.

The `[newQ]` rule creates a new queue object and adds it to the queue using a fresh address. This new queue object is initialised in a pending state, and it does not have any callbacks related to it.

The `[fulfill-pending]` rule demonstrates the case when we fulfill a pending queue object with value v which is not the $\bot$ element, and does *not* correspond to a queue object. In

$$\frac{e \hookrightarrow e'}{\pi,\ \phi,\ \kappa,\ \tau,\ E[e] \rightarrow \pi',\ \phi',\ \kappa',\ \tau',\ E[e']}\ \text{[e-context]}$$

$$\frac{fresh\,\alpha \quad \pi' = \pi[\alpha \mapsto (\mathtt{pending},\ [],\ [],\ \bot)]}{\pi,\ \phi,\ \kappa,\ \tau,\ E[\mathtt{newQ}()] \rightarrow \pi',\ \phi,\ \kappa,\ \tau,\ E[\alpha]}\ \text{[newQ]}$$

$$\frac{\begin{array}{c} v \neq \bot \quad (\mathtt{pending},\ t,\ k, l) = \pi(p) \quad v \notin dom(\pi) \quad t' = \langle(\alpha,\ f,\ [v],\ r) \mid (\alpha,\ f,\ a,\ r) \in t\rangle \\ \kappa' = \kappa :: t' \quad \chi = (\mathtt{fulfilled}, v) \quad \pi' = \pi[p \mapsto (\chi, [], [], l)] \quad p \neq l_{time} \wedge p \neq l_{io} \end{array}}{\pi,\ \phi,\ \kappa,\ \tau,\ E[p.\mathtt{fulfill}(v)] \rightarrow \pi',\ \phi,\ \kappa',\ \tau,\ E[l.\mathtt{fulfill}(v)]}\ \text{[fulfill-pending]}$$

$$\frac{\begin{array}{c} (\mathtt{pending},\ t,\ k, l) = \pi(p) \quad v \in dom(\pi) \\ p(v) = (\mathtt{pending},\ t',\ k', \bot) \quad \pi' = \pi[v \mapsto (\mathtt{pending},\ t',\ k',\ p)] \end{array}}{\pi,\ \phi,\ \kappa,\ \tau,\ E[p.\mathtt{fulfill}(v)] \rightarrow \pi',\ \phi,\ \kappa',\ \tau,\ E[\mathtt{undef}]}\ \text{[fulfill-pend-pend]}$$

$$\frac{(\mathtt{pending},\ t,\ k, l) = \pi(p) \quad v \in dom(\pi) \quad p(v) = ((\mathtt{fulfilled},\ v'),\ t',\ k', m)}{\pi,\ \phi,\ \kappa,\ \tau,\ E[p.\mathtt{fulfill}(v)] \rightarrow \pi,\ \phi,\ \kappa,\ \tau,\ E[p.\mathtt{fulfill}(\mathtt{v'})]}\ \text{[fulfill-pend-ful]}$$

$$\frac{\begin{array}{c} v = \bot \quad (\mathtt{pending},\ t,\ k, l) = \pi(p) \quad \kappa' = \kappa :: t \\ \chi = (\mathtt{fulfilled}, v) \quad \pi' = \pi[p \mapsto (\chi, [], [], l)] \end{array}}{\pi,\ \phi,\ \kappa,\ \tau,\ E[p.\mathtt{fulfill}(v)] \rightarrow \pi',\ \phi,\ \kappa',\ \tau,\ E[l.\mathtt{fulfill}(v)]}\ \text{[fulfill-pending-}\bot\text{]}$$

$$\frac{\pi(p) \downarrow_1 \neq \mathtt{pending}}{\pi,\ \phi\ ,\kappa,\ \tau,\ E[p.\mathtt{fulfill}(v)] \rightarrow \pi,\ \phi,\ \kappa,\ \tau,\ E[\mathtt{undef}]}\ \text{[fulfill-settled]}$$

$$\frac{\begin{array}{c} (\mathtt{pending}\ , t\ , k, l) = \pi(p) \quad t' = t \cdot (p',\ f,\ [n_1, n_2, \ldots, n_n],\ r) \\ \pi' = \pi[p \mapsto (\mathtt{pending},\ t',\ k, l)] \end{array}}{\pi,\ \phi,\ \kappa,\ \tau,\ E[p.\mathtt{registerFul}(f, p', r, n_1, n_2, \ldots, n_n)] \rightarrow \pi,'\ \phi,\ \kappa,\ \tau,\ E[\mathtt{undef}]}\ \text{[registerFul-pending]}$$

$$\frac{\begin{array}{c} p \neq l_{time} \wedge p \neq l_{io} \quad (s,\ t\ , k\ , \chi) = \pi(p) \quad s \downarrow_1 = \mathtt{fulfilled} \\ s \downarrow_2 \neq \bot \quad \kappa' = \kappa \cdot (p',\ f,\ [s \downarrow_2],\ r) \end{array}}{\pi,\ \phi,\ \kappa,\ \tau,\ E[p.\mathtt{registerFul}(f, p', r, n_1, n_2, \ldots, n_n)] \rightarrow \pi,'\ \phi,\ \kappa,\ \tau,\ E[\mathtt{undef}]}\ \text{[registerFul-fulfilled]}$$

$$\frac{\begin{array}{c} p \neq l_{time} \wedge p \neq l_{io} \quad (s,\ t\ , k\ , \chi) = \pi(p) \quad s \downarrow_1 = \mathtt{fulfilled} \\ s \downarrow_2 = \bot \quad \kappa' = \kappa \cdot (p',\ f,\ [n_1, n_2, \ldots, n_n],\ r) \end{array}}{\pi,\ \phi,\ \kappa,\ \tau,\ E[p.\mathtt{registerFul}(f, p', r, n_1, n_2, \ldots, n_n)] \rightarrow \pi,'\ \phi,\ \kappa,\ \tau,\ E[\mathtt{undef}]}\ \text{[registerFul-fulfilled-}\bot\text{]}$$

$$\frac{\begin{array}{c} p = l_{time} \vee p = l_{io} \quad (s,\ t\ , k\ , \chi) = \pi(p) \quad s \downarrow_1 = \mathtt{fulfilled} \\ s \downarrow_2 = \bot \quad \tau' = \tau \cdot (p',\ f,\ [n_1, n_2, \ldots, n_n],\ r) \end{array}}{\pi,\ \phi,\ \kappa,\ \tau,\ E[p.\mathtt{registerFul}(f, p', r, n_1, n_2, \ldots, n_n)] \rightarrow \pi,'\ \phi,\ \kappa,\ \tau,\ E[\mathtt{undef}]}\ \text{[registerFul-timer-io-}\bot\text{]}$$

$$\frac{p \in dom(\pi) \quad \phi' = p \cdot \phi}{\pi,\ \phi,\ \kappa,\ \tau,\ E[\mathtt{append}(p)] \rightarrow \pi,\ \phi',\ \kappa,\ \tau,\ E[\mathtt{undef}]}\ \text{[append]}$$

$$\frac{}{\pi,\ p \cdot \phi,\ \kappa,\ \tau,\ E[\mathtt{pop}()] \rightarrow \pi,\ \phi,\ \kappa,\ \tau,\ E[\mathtt{undef}]}\ \text{[pop]}$$

$$\frac{\phi = p \cdot \phi'}{\pi,\ \phi,\ \kappa,\ \tau,\ E[\mathbf{err}\ v] \rightarrow \pi,\ \phi',\ \kappa,\ \tau,\ E[p.\mathtt{reject}(v)]}\ \text{[error]}$$

Figure 3.3: The semantics of $\lambda_{\mathtt{q}}$

$$\frac{}{\pi,\ \phi,\ \kappa,\ \tau,\ E[\texttt{defaultFulfill}(v)] \rightarrow \pi,\ \phi,\ \kappa,\ E[v]}\ \text{[defaultFulfill]}$$

$$\frac{}{\pi,\ \phi,\ \kappa,\ \tau,\ E[\texttt{defaultReject}(v)] \rightarrow \pi,\ \phi,\ \kappa,\ E[\mathbf{err}\ v]}\ \text{[defaultReject]}$$

Figure 3.4: The semantics of `defaultFulfill`, `defaultReject` functions.

particular, we first change the state of the receiver object from "pending" to "fulfilled". In turn, we update the already registered callbacks (if any) by setting the value v as the only argument of them. Then, we add the updated callbacks to the list of scheduled callbacks κ (assuming that the receiver is neither l_{time} nor l_{io}). Also observe that the initial expression is reduced to $l.\texttt{fulfill}(v)$, that is, if there is a dependent queue object l, we also fulfill that queue object with the initial value v.

The `[fulfill-pend-pend]` describes the scenario of fulfilling a pending queue object p with another pending queue object v. In this case, we update queue π by making queue object p to be dependent on v. This means that whenever p is settled whenever v is settled, and with exactly the same way. Notice that both both p and v remain pending.

The `[fulfill-pend-ful]` rule demonstrates the case when we try to fulfill a pending queue object p with another queue object v which is in a fulfilled state. Then, p simply resolves with the value with which v is fulfilled. This is expressed by the resulting expression $p.\texttt{fulfill}(v')$.

The `[fulfill-pending-⊥]` rule captures the case when we fulfill a queue object with a $\perp$ value. This rule is the same with `[fulfill-pending]` however, this time we do not update the arguments of the already registered callbacks of the queue object (if any).

The `[fulfill-settled]` rule illustrates the case when we try to fulfill a settled queue object. Notice that this rule neither affects the queue π nor the lists of scheduled callbacks κ and τ.

The `[registerFul-pending]` rule adds the given callback f to the list of callbacks that should be executed once the queue object p is fulfilled. Note that this rule also associates this callback with the queue object p' given as the second argument that should be fulfilled upon the termination of f. Also, this rule adds any extra arguments passed in `registerFul` as arguments of f.

The `[registerFul-fulfilled]` rule adds the given callback f to the list κ (assuming that the receiver is neither l_{time} nor l_{io}). The value with which the receiver is fulfilled is used as the only argument of the given function. As the previous rule, it relates the given queue object p' with the execution of the callback. Notice that this time we *ignore* any extra arguments passed in `registerFul`, as the value with which the queue object p is fulfilled is *not* $\perp$.

The `[registerFul-fulfilled-⊥]` rule describes the case where we register a callback f on a queue object which is fulfilled with a $\perp$ value. Unlike the `[registerFul-fulfilled]` rule, this rule does *not* neglect any extra arguments passed in `registerFul`. In particular, it sets those arguments as parameters of the given callback. This distinction allows us to pass

$$\frac{\kappa = (q, f, a) \cdot \kappa' \qquad \phi = [] \qquad \phi' = q \cdot \phi}{\pi\ , \phi,\ \kappa,\ \tau,\ E[\bullet] \rightarrow \pi,\ \phi',\ \kappa',\ \tau,\ q.\texttt{fulfill}(E[f(a)]); \texttt{pop}(); \bullet} \text{[event-loop]}$$

$$\frac{\begin{array}{cc}\texttt{pick}\ (q, f, a)\ \texttt{from}\ \tau & \tau' = \langle \rho \mid \forall \rho \in \tau.\ \rho \neq (q, f, a) \rangle \\ \phi = [] & \phi' = q \cdot \phi\end{array}}{\pi\ , \phi,\ [],\ E[\bullet] \rightarrow \pi,\ \phi',\ [],\ \tau',\ q.\texttt{fulfill}(E[f(a)]); \texttt{pop}(); \bullet} \text{[event-loop-timers-io]}$$

Figure 3.5: The semantics of the event loop

arguments explicitly to a callback. Most notably, these arguments are not dependent on the value with which a queue object is fulfilled or rejected.

The `[registerFul-timer-io-⊥]` rule is exactly the same as the previous one, but this time we deal with queue objects located either at l_{time} or at l_{io}. Thus, we add the given callback f to the list τ instead of κ.

The `[append]` rule appends an element p to the front of the current queue chain. Note that this rule requires the element p to be a queue object (i.e. $p \in dom(\pi)$). On the other hand, the `[pop]` rule simply removes the top element of the queue chain.

The `[error]` rule demonstrates the case when we encounter an uncaught exception and the queue chain is not empty. In that case, we do not propagate the exception to the caller, but we pop the queue chain and get the top element. In turn, we reject the queue object p which is specified in that top element. In this way, we capture the concrete behaviour of uncaught exceptions when they are triggered during the execution of an asynchronous callback or a promise executor.

Finally, Figure 3.4 illustrates the semantics of the special functions `defaultFulfill`, and `defaultReject`. It is easy to see that the rules are fairly simple: the `defaultFulfill`(v) is reduced to v, while `defaultReject`(v) evaluates to `err` v. Both functions do not have any side-effects to the queue, queue chain, or the lists of scheduled callbacks κ and τ.

3.1.3 Modelling the Event Loop

A reader might wonder why do we keep two separate lists, i.e. list τ for holding callbacks coming from l_{time} or l_{io} queue objects, and list κ for callbacks of any other queue object. The intuition behind this design choice is that it is actually convenient for us to correctly model the concrete semantics of the event loop. In particular, the implementation of the event loop assigns different priorities to the callbacks depending on their kind (Loring et al., 2017; Node.js, 2018b). For example, a callback of a promise object is executed before any callback of a timer or an asynchronous I/O operation. The following code illustrates that scenario:

```
1  setTimeout(function foo() {
2    // executed after callback qux()
3  });
4
5  fs.readFile("file", function bar() {
6    // executed after callback qux()
7  })
8
```

```
9  Promise.resolve("foo").then(function baz() {
10   // This callback is executed first.
11 }).then(function qux() {
12   // This callback is executed second.
13 })
```

Initially, (assuming that the I/O operation is complete) we have three callbacks ready for execution (i.e. `foo()`, `bar()`, and `baz()`). The event loop *deterministically* picks `baz()`, because callbacks coming from promises have a higher priority than any other callbacks. After the execution of `baz()`, we schedule callback `qux()`. In turn, the next iteration of the event loop selects `qux()` for execution, even though it was scheduled much more after callbacks `foo()` and `bar()`. The reason for that is again the same: the event loop favours callbacks of promises over callbacks of timers and asynchronous I/O.

Note that in the next iteration of the event loop, i.e. after the termination of `qux()`, we do not know which callback (i.e. `foo()` or `bar()`) the event loop is going to execute. Specifically, the Node.js implementation of the event loop schedules the non-promises callbacks in different phases, where each phase is executed *preemptively* (Loring et al., 2017; Node.js, 2018b). For example, there is a phase, where the event loop executes callbacks associated with timers, whereas there is a similar phase for I/O related callbacks.

In this context, Figure 3.5 demonstrates the semantics of the event loop. The `[event-loop]` rule simply pops the first scheduled callback from list κ. Then, we get the queue object specified in that callback and we attach it to the front of the queue chain. Adding the queue object q to the top of the queue chain allows us to reject that queue object, if there is an uncaught exception during the execution of f. In this case, the evaluation of `fulfill` will not have any effect on the already rejected queue object q. Observe how the event loop is evaluated (i.e. $q.\texttt{fulfill}(f(a)); \texttt{pop}(); \bullet$). Specifically, once we execute callback f and fulfill the dependent queue object q with the return value of that callback, we evaluate `pop()`, that is, we pop the top element of the queue chain before we evaluate the event loop again. This is an invariant of the semantics of the event loop: every time we evaluate the event loop, the queue chain is *always* empty.

The `[event-loop-timers-io]` rule handles the case when the list κ is empty. In other words, that rule simply states that if there are not any callbacks, which come from common queue objects (i.e. queue objects which are not located at any of the special addresses l_{time}, l_{io}), inspect list τ, which holds scheduled callbacks coming from the queue objects located at l_{time} and l_{io}, and pick *non-deterministically* one of those. Selecting a callback non-deterministically allows us to over-approximate the concrete behaviour of the event loop regarding the different execution phases (Loring et al., 2017). Overall, that rule describes the scheduling policy presented in the work of Loring et al. (2017), where initially we look for any callbacks of promises, and if there exist, we select one of those. Otherwise, we choose any callback associated with timers and asynchronous I/O at random.

3.2 Modelling Asynchronous Primitives

In this section, we present how promises, timers, and asynchronous I/O can be expressed through our calculus λ_{q}.

3.2.1 Promises

The queue objects and their operations introduced in λ_q are very closely related to promises of the ECMAScript specification (ECMA-262, 2018). Therefore, the translation of promises operations into λ_q is straightforward. We model every property and method except for `Promise.all()`.

Promise Constructor

A model for the promise constructor is presented in the following code fragment:

```
1  function Promise(f) {
2    var promise = newQ();
3    append(promise);
4    f(p.resolve, p.reject);
5    pop();
6    return promise;
7  }
```

The promise constructor expects a function `f`, and it creates a new queue object via `newQ` construct. According to the ECMAScript (ECMA-262, 2018), the promise constructor gets the given function `f` and executes it *synchronously* by passing a pair of resolving functions as argument. Note that previous models (Loring et al., 2017) mistakenly assumed that `f` is executed asynchronously. Also, any uncaught exception that is triggered by the execution of `f` is not propagated to the call stack. In contrast, the constructor rejects the created promise with the uncaught exception. For that purpose, before we invoke function `f`, we append the created promise to the queue chain via `append(promise)` expression. After the execution of `f`, we pop that element through `pop()` at line 5. Also observe that we pass the resolving functions `p.fulfill` and `p.reject` as arguments in `f` as the ECMAScript states (ECMA-262, 2018).

Promise.resolve

The JavaScript `Promise.resolve()` function creates a new promise, and resolves it with the given value. This can be expressed via the following code:

```
1  Promise.resolve = function(value) {
2    var promise = newQ();
3    if (typeof value.then === "function") {
4      var t = newQ();
5      t.fulfill(⊥);
6      t.registerFul(value.then, t, promise.fulfill, promise.reject);
7    } else
8      promise.fulfill(value);
9    return promise;
10 }
```

According to the ECMAScript (ECMA-262, 2018), if the given `value` is a *thenable*, (i.e. an object which has a property named "then" and that property is a callable), the created promise resolves *asynchronously*. Specifically, the `value.then` is executed asynchronously, by passing functions `promise.fulfill` and `promise.reject` as arguments. Observe how the expressiveness of λ_q can model this source of asynchrony (lines 4–6). First, we create a fresh queue object `t`, and we fulfill it with $\perp$ value (lines 4, 5). Then, at line 6, we schedule function `value.then` by registering it on the newly created queue object. Notice that we also pass `promise.fulfill` and `promise.reject` as extra arguments. That means that those functions

will be the actual arguments of `value.then`, when it is called. On the other hand, if `value` is not a thenable, we synchronously resolve the created promise using `promise.fulfill` at line 8.

Promise.reject

The `Promise.reject()` function is similar to `Promise.resolve()`, however, this time we reject the freshly created promise.

```
1  Promise.reject = function(value) {
2    var promise = newQ();
3    promise.reject(value);
4    return promise;
5  }
```

Promise.prototype.then

```
1  Promise.prototype.then = function(f1, f2) {
2    if (typeof f1 !== "function")
3      f1 = defaultFulfill;
4    if (typeof f2 !== "function")
5      f2 = defaultReject;
6    var promise = newQ();
7    this.registerFul(f1, promise);
8    this.registerRej(f2, promise);
9    return promise;
10 }
```

The JavaScript method `then()` expects two optional callbacks: the first callback is executed when the receiver is fulfilled, and the second one is invoked when the receiver is rejected. If any of the given callbacks is not a function, `then()` registers the corresponding default resolving function (i.e. either `defaultFulfill()` or `defaultReject()`) on the receiver. In turn, it creates a fresh promise, and registers the given callbacks on the receiver (lines 6, 7). Observe that the newly created promise is settled based on the return value of the given callbacks (i.e. `promise` is the second argument of the `registerFul` and `registerRej` contructs).

Promise.prototype.catch

The `Promise.prototype.catch()` can be written as:

```
1  Promise.prototype.catch = function(f) {
2    return this.then(undefined, f);
3  }
```

Promise.race

`Promise.race()` gets an iterable of promises, and creates a new promise which resolves asynchronously according to the promise included in the given iterable which is firstly fulfilled or rejected. This is modelled as follows:

```
1  Promise.race = function(iter) {
2    var promise = newQ();
3
4    function race() {
5      for (elem in iterable) {
6        promise.fulfill(elem);
7      }
8    }
9
10   var t = newQ();
11   t.fulfill(⊥);
12   t.registerFul(race, t);
13   return promise;
14 }
```

As before, we create a fresh queue object at line 10 to asynchronously execute function `race()` (line 4) which is responsible for iterating over the sequence of promises and fulfilling the promise allocated at line 2 with the promise which is firstly fulfilled or rejected.

3.2.2 Timers

To model timers we follow similar approach to the work of Loring et al. (2017). Specifically, we start with a queue π, which contains a queue object q_{time} which is located at address l_{time}. The queue object q_{time} corresponds to $((\texttt{fulfilled}, \bot), [], [], \bot)$. Besides that, we extend the syntax of λ_q by adding one more expression as follows:

$$
\begin{array}{lll}
e \in Exp & ::= & \ldots \\
 & \mid & \texttt{addTimerCallback}(e_1, e_2, e_3, \ldots) \\
 & & \\
E & ::= & .. \\
 & \mid & \texttt{addTimerCallbackCallback}(E, e, \ldots) \mid \texttt{addTimerCallback}(v, \ldots, E, e, \ldots)
\end{array}
$$

The new expression adds a new callback e_1 to the queue object located at address l_{time}. That callback should be called with any optional parameters passed in `addTimerCallback`, namely, e_2, e_3, and so on. The exact semantics is presented below:

$$
\frac{q = \pi(l_{time})}{\pi,\ \phi,\ \kappa,\ \tau,\ \texttt{addTimerCallback}(f,\ n_1, \ldots) \rightarrow \pi,\ \phi,\ \kappa,\ q.\texttt{registerFul}(f,\ q,\ n_1, \ldots)}\ \text{[addTimerCallback]}
$$

The rule above retrieves the queue object q corresponding to the address l_{time} (Recall again that l_{time} can be found in the initial queue). Then, the given expression is reduced to $q.\texttt{registerFul}(f,\ q,\ n_1, \ldots)$. In particular, we add a new callback f to the queue object found at l_{time}. Observe that we pass the same queue object (i.e. q) as the second argument of `registerFul`. That means that the execution of f does not affect any queue object since q is already settled. Recall that according to the `[fulfill-settled]` rule (Figure 3.3), trying to fulfill (and similarly to reject) a settled queue object does not have any effect. Beyond that, since q is fulfilled with $\bot$, the extra arguments (i.e. $n_1, \ldots$) are also passed as of f.

The code snippet below demonstrates how we can model a timer, e.g. `setTimeout()` using the new language construct.

```
1 function setTimeout(f, t, n1,...) {
2   after t milliseconds elapses;
3   addTimerCallback(f, n1,...);
4 }
```

3.2.3 Asynchronous I/O

Modelling of asynchronous I/O has a high correspondence to that of timers. Specifically, the initial queue contains one more queue object $q_{io} = ((\texttt{fulfilled}, \bot), [], [])$ located at address l_{io} which is specifically targeted for asynchronous I/O operations. We then extend the syntax of λ_q as follows:

$$e \in Exp \quad ::= \quad \ldots \mid \texttt{addIOCallback}(e_1, e_2, e_3, \ldots)$$

$$E \quad ::= \quad .. \mid \texttt{addIOCallback}(E, e, \ldots) \mid \texttt{addIOCallback}(v, \ldots, E, e, \ldots)$$

The semantics of `addIOCallback` is exactly the same with that of `addTimerCallback`, however, this time, the queue object located at l_{io} is used by the reduced expression. Namely,

$$\frac{q = \pi(l_{io})}{\pi,\ \phi,\ \kappa,\ \tau,\ \texttt{addIOCallback}(f,\ n_1, \ldots) \rightarrow \pi,\ \phi,\ \kappa,\ q.\texttt{registerFul}(f,\ q,\ n_1, \ldots)} \text{[addIOCallback]}$$

To this end, an asynchronous I/O can be expressed as:

```
1 function asyncIO(f) {
2   data = ... // perform async I/O
3   addIOCallback(f, data);
4 }
```

3.3 Comparison with Existing Models

Our λ_q calculus has a lot of similarities with other existing models, namely, λ_{async}, and λ_p (Loring et al., 2017; Madsen et al., 2017) which we use as a base. However, there are two points where our model distinguishes from the existing works:

- The λ_q calculus models the effects of uncaught exceptions in the execution of callbacks and promise executors via the queue chain. Specifically, any uncaught exception occurred in the event loop is not propagated to the caller, so, the program does not terminate abnormally. Also, when we deal with promises, any uncaught exception is used to reject the next promise of the chain. Similarly, an uncaught exception in the execution of a promise executor leads to the rejection of the created promise.
- The λ_q calculus as λ_{async} (Loring et al., 2017) is designed to model not only promises but also timers and asynchronous I/O operations. However, contrary to our model, λ_{async} does not capture the case when we pass arguments to a callback explicitly. In other words, the arguments of a callback are independent of the value with which a queue object is fulfilled or rejected, like the example below:

```
setTimeout(callback, 0, "foo", "bar")
```

> Our model covers this case, by inspecting the value with which queue objects are settled: if it is $\bot$, we allow explicit arguments to be passed in a given callback. (Recall the difference between `[registerFul-fulfilled]` and `[registerFul-fulfilled-⊥]` rules).

Overall, our modifications enable us to model almost all the sources of asynchrony; some of them are not handled by the existing models. For instance, as presented in Section 3.2.1, $\lambda_{\mathtt{q}}$ is capable of modelling the case when we fulfill a promise with a thenable. Our calculus allows to express the fact that thenable is called asynchronously with arguments that are explicitly passed (i.e. `promise.fulfill`, `promise.reject`).

Chapter 4

The Core Analysis

We introduce a static analysis for asynchronous JavaScript programs by exploiting the λ_q calculus described in Chapter 3. The analysis is designed to be sound, thus, we devise abstract domains and semantics that over-approximate the behaviour of λ_q. Currently, there are few implementations for effectively dealing with all the asynchronous features of JavaScript, and previous efforts mainly focus on modelling the event system of client-side applications (Jensen et al., 2011; Park et al., 2016). To the best of our knowledge, there is not any implementation of a static analysis technique which is capable of handling promises which were introduced in the 6th edition of the ECMAScript specification (ECMA-262, 2015). Our work builds on top of an existing static analyser, namely, *TAJS* (Recall Section 2.2.2), which we extend with the abstract version of λ_q. To better understand how asynchronous callbacks are scheduled and finally executed by the event loop, our analysis introduces the notion of *callback graph*; a graph that captures the execution order between callbacks. By exploiting callback graph, we devise a more precise analysis that respects the execution order of callbacks along with three different flavours of context-sensitivity which are specifically targeted for the abstract model of λ_q.

In this chapter, we give an overview of the analysis adopted by TAJS (Section 4.1). In turn, we highlight all the necessary extensions made in TAJS so that it can analyse asynchronous programs (Section 4.2). Then, we introduce the definition of callback graph; we describe how the analysis computes it (Section 4.3), and we finally present different kinds of analysis sensitivity by leveraging callback graph and other characteristics of the analysis model (Section 4.4).

4.1 Introduction to TAJS

TAJS (Jensen et al., 2009) is a static analyzer which implements a classical inter-procedural analysis using monotone frameworks (Kam and Ullman, 1977). The main goal of TAJS is to detect various kinds of errors found in JavaScript programs, including, but not limited to, accessing a property of a `null` or `undefined` variable, calling a non-function variable, reading an absent variable, detecting intricate and error-prone type conversions, etc. (Jensen et al., 2009). In addition to bug detection, TAJS also computes call graph and all reachable states from a given initial state (Jensen et al., 2009).

A key question that arises is: why did we choose TAJS as a base of our approach? Our available options were TAJS, JSAI, and SAFE which are briefly described in Section 2.2.2.

We chose TAJS to extend, and there are many reasons for that. First, TAJS is a more mature project as it is a product of almost 10 years of development (Jensen et al., 2009). On the other hand, SAFE and JSAI are more recent tools. In this context, TAJS implements some important optimisations which enhance the performance (Jensen et al., 2010) and precision (Jensen et al., 2009) of the analysis, e.g. JSAI seems to produce more false positives compared to TAJS when they are evaluated on the same benchmarks (Kashyap et al., 2014). Also, TAJS has partial models for more recent features of JavaScript, and it provides the base for modelling Node.js modules and libraries as it supports `require` mechanism for importing other JavaScript files. TAJS also comes with an Apache Licence 2.0; a permissive licence which allows us to incorporate our modifications to the tool for private use.

In this section, we describe the core analysis employed by TAJS, providing the foundations for designing and presenting our extensions in future sections. This section does not include any new contribution and the presentation is mainly based on the previous existing works of Jensen et al. (2009, 2010). Note that we only present the material which is relevant to the context of our extensions. Therefore, we might omit some details of TAJS, which are not applicable to our analysis.

4.1.1 Control-Flow Graph

TAJS analysis is a flow-sensitive inter-procedural analysis, meaning that it operates on top of the control-flow graph (CFG) of the programs. The CFG representation is specifically designed for TAJS analysis (Jensen et al., 2009), and unsurprisingly, each node in the CFG represents program points, and edges stand for possible flow between different program points. The CFG representation organises code into functions, and each function consists of a set of basic blocks; every basic block is a set of CFG nodes where every node propagates flow to exactly one successor sequentially. Unlike nodes inside a basic block, the exit point of a basic block might propagate flow to zero or more basic blocks. Beyond that, each function has two different exit points: 1) a normal exit, and 2) an exceptional exit in order for analysis to distinguish flow that is triggered by exceptions. Also note that the top-level code is placed inside a special function called `<main>` and the entry point of this function corresponds to the program entry.

CFG defines two different kinds of edges: 1) intra-procedural edges which represent flow between nodes of the same function, and 2) inter-procedural edges which denote flow between functions exit points and call nodes (i.e. nodes which represent call statements). Note that intra-procedural edges are created during CFG construction, while inter-procedural edges are constructed during the course of the analysis (Jensen et al., 2009).

4.1.2 The Analysis Framework

The analysis of TAJS is based on monotone frameworks (Kam and Ullman, 1977). Given a program P, represented by a CFG as $P = N \times E$, where N is the set of nodes, and E is the set of edges, the instance of the analysis monotone framework is defined as follows (Jensen et al., 2010):

$$A = (P,\ F,\ C,\ Value,\ n_0,\ c_0,\ T)$$

where F is the set of all functions found in the program, C is the set of contexts derived from the context-sensitivity strategy adopted by the analysis, $Value$ is the basic lattice for

$$
\begin{aligned}
\alpha \in \mathit{AnalysisLattice} &= (C \times N \hookrightarrow \mathit{State}) \times \mathit{CallGraph} \\
g \in \mathit{CallGraph} &= \mathcal{P}(C \times N \times C \times F) \\
l \in L &= \{l_i \mid i \text{ is an allocation site}\} \\
\sigma \in \mathit{State} &= \mathit{Heap} \times \mathit{Stack} \\
h \in \mathit{Heap} &= L \hookrightarrow \mathit{Object} \\
o \in \mathit{Object} &= (\mathit{Pr} \hookrightarrow \mathit{Value} \times \mathit{Absent} \times \mathit{Attrs}) \times \mathcal{P}(\mathit{ScopeChain}) \\
\mathit{attr} \in \mathit{Attrs} &= \mathit{ReadOnly} \times \mathit{DontDelete} \times \mathit{DontEnum} \\
s \in \mathit{Stack} &= (V \to \mathit{Value}) \times \mathcal{P}(\mathit{ExecutionContext}) \\
\mathit{ctx} \in \mathit{ExecutionContext} &= \mathit{ScopeChain} \times L \times L \\
c \in \mathit{ScopeChain} &= L^* \\
v \in \mathit{Value} &= \mathit{Undef} \times \mathit{Null} \times \mathit{Bool} \times \mathit{Num} \times \mathit{Str} \times \mathcal{P}(L) \\
p \in \mathit{Pr} &= \text{set of properties}
\end{aligned}
$$

Figure 4.1: Abstract Domains of TAJS analysis (Jensen et al., 2009, 2010).

modelling values of program variables, n_0 stands for the CFG node of the program entry, c_0 is the context corresponding to the initial program point n_0 and T is the transfer function that is applied on every program location. Transfer function T is defined as (Jensen et al., 2010):

$$T = C \times N \to \mathit{AnalysisLattice} \to \mathit{AnalysisLattice}$$

Note that every pair $(c, n) \in C \times N$ represents a unique program point. Therefore, the transfer function T operates on every such pair, it takes as input an element $\alpha \in \mathit{AnalysisLattice}$ and produces a new element $\alpha \in \mathit{AnalysisisLattice}$. In other words, the analysis can be seen as a transition system, where the transfer function T modifies the element α according to the semantics of node $n \in N$ in context $c \in C$.

Figure 4.1 fully describes the domains of the TAJS analysis. Every element $\alpha \in \mathit{AnalysisLattice}$ consists of a map of every reachable program point $(c, n) \in C \times N$ to an abstract state $\sigma \in \mathit{State}$, and a call graph $g \in \mathit{CallGraph}$. A call graph is an element g of the powerset $\mathcal{P}(C \times N \times C \times F)$. Every tuple $(c_1, n, c_2, f) \in g$ means that there might be an invocation from node n in context c_1 to the function f in context c_2 (Jensen et al., 2010).

An abstract state consists of a heap $h \in \mathit{Heap}$ and a stack $s \in \mathit{Stack}$. An abstract heap is just a map of abstract addresses to objects. Note that the set of abstract addresses is simply defined as the set of allocation sites (i.e. the set of CFG nodes that may allocate a new object). An object $o \in \mathit{Object}$ is described as a partial map of property names to values along with two extra components (i.e. Absent, Attrs) which model metadata associated with a property. Specifically, Absent is a lattice which is used to model whether the property may be absent or not, whereas an element $\mathit{attr} \in \mathit{Attrs}$ models the attributes of a property according to the ECMAScript specification (Jensen et al., 2009; ECMA-262, 2018). Also, every object $o \in \mathit{Object}$ contains an element of a powerset $\mathcal{P}(\mathit{ScopeChain})$ to model the internal property `[[Scope]]` (Jensen et al., 2009; ECMA-262, 2018). A stack consists of a map, which associates every temporary variable with a value, along with an extra component to model the execution context on which the current function operates. The execution context follows the ECMAScript specification (Jensen et al., 2009; ECMA-262, 2018), thus, it is described by a scope chain (i.e. a sequence of addresses), an address corresponding to `this` object and an

address corresponding to the variable object.

Finally, $Value$ is a finite lattice which is composed by a number of other lattices; each of them models a different JavaScript type i.e. `null`, `undefined`, number, string, boolean or object reference. In this way, the analysis is able to abstract the dynamic typing of JavaScript. For example, the element ($\bot$, `null`, $\top$, $\bot$, $\bot$, $\{l_1, l_{15}\}$) $\in Value$ denotes that the variable may be `null`, any boolean (described by the third element which is $\top$) or an object coming from the allocation sites l_1 or l_{15} (Jensen et al., 2009).

4.1.3 Computing the Solution

Algorithm 1 The fixpoint iteration algorithm implemented in TAJS

Input: $A = (P,\ F,\ C,\ Value,\ n_0,\ c_0,\ T)$
Output: $\alpha \in AnalysisLattice$
1: $\alpha = \bot$
2: $W = \{c_0, n_0\}$
3: **while** $W \neq \emptyset$ **do**
4: pop a program point (c, n) from W
5: $\alpha = T(c, n)(\alpha)$
6: $\sigma = a \downarrow_1 (c, n)$
7: **for all** $n' \in$ `nextNodes`(P, n) **do**
8: $c' =$ `getContext`(n', σ)
9: **if** $(c', n') \notin \alpha \downarrow_1$ **then**
10: $W = W \cup \{(c', n')\}$
11: $\alpha \downarrow_1 (c', n') = \sigma$
12: **else**
13: $\sigma_{old} = \alpha \downarrow_1 (c', n')$
14: $\sigma_{new} = \sigma_{old} \sqcup \sigma$
15: **if** $\sigma_{new} \not\sqsubseteq \sigma_{old}$ **then**
16: $W = W \cup \{(c', n')\}$
17: $\alpha \downarrow_1 (c', n') = \sigma_{new}$
18: **end if**
19: **end if**
20: **end for**
21: **end while**

The output of the analysis is the set of all reachable states from a given initial state along with a call graph. This is represented by the final element $\alpha \in AnalysisLattice$, upon the termination of the analysis. More formally, the solution $\alpha \in AnalysisLattice$ of the analysis must satisfy the constraints of the following formula (Jensen et al., 2010):

$$\forall c \in C, n \in N : T(n, c)(\alpha) \sqsubseteq \alpha \tag{4.1}$$

where $\sqsubseteq$ stands for the partial order relation between two elements.

This solution of Formula 4.1 is computed via a fixpoint iteration algorithm, which is shown in Algorithm 1 (Jensen et al., 2010)[1]. Specifically, this algorithm uses a worklist W which contains all program points (i.e. pairs $(c, n) \in C \times N$) that should be analysed. Initially,

[1]We slightly change the presentation of the fixpoint algorithm which was initially presented in the work of (Jensen et al., 2010). Specifically, we put more details to the pseudocode to better understand the inner workings of the analysis.

W holds only the location which corresponds to the program entry (Algorithm 1, line 2). In turn, the algorithms extracts a program point (c, n) from W, and applies the transfer function on it, producing a new analysis lattice element α (Algorithm 1, line 5). Recall that the transfer function produces a new element $\alpha \in AnalysisLattice$ according to the abstract semantics of node n in context c. After that, the algorithm propagates the resulting state σ of the fresh $\alpha \in AnalysisLattice$ (Algorithm 1, line 6) to the successor nodes of the current node n with regards to the given CFG P (Algorithm 1, lines 7–18). In particular, as a starting point, the algorithm computes the context c' to form the next program location points where the state should be propagated (i.e. pairs (c', n'), where n' is one of the successor nodes of n). Note that the context c' is produced based on the node n', the context-sensitivity strategy with which the analysis is parameterised, and the state σ. As the algorithm progresses, there are two possible scenarios. First, the next program point (c', n') is not part of $\alpha\downarrow_1$, meaning that the analysis reaches a new program location. Then, the algorithm updates the map of reachable states and the worklist accordingly (Algorithm 1, line 9–11). Second, the next program point (c', n') is in the set of reachable points, so the algorithm inspects the current state corresponding to (c', n') (i.e. σ_{old}), and propagates the resulting state σ to σ_{old} via the least upper bound operation ($\sqcup$) at line 14, producing σ_{new} which is the merge of σ and σ_{old}. If there is a change (i.e. $\sigma_{new} \not\sqsubseteq \sigma_{old}$), the algorithm puts the program point (c', n') to W in order to analyse it again, and updates $\alpha\downarrow_1$ with the new state σ_{new} (Algorithm 1, lines 13–17). This procedure repeats until convergence, that is, W is empty.

4.2 Extending TAJS

$$
\begin{aligned}
l \in L_{async} &= L \cup \{l_{time}, l_{io}\} \\
\pi \in \widehat{Queue} &= L_{async} \hookrightarrow \mathcal{P}(\widehat{QueueObject}) \\
q \in \widehat{QueueObject} &= \widehat{QueueState} \times \mathcal{P}(\widehat{Callback}) \times \mathcal{P}(\widehat{Callback}) \times \mathcal{P}(L_{async}) \\
qs \in \widehat{QueueState} &= \{\texttt{pending}\} \cup (\{\texttt{fulfilled}, \texttt{rejected}\} \times Value) \\
clb \in \widehat{Callback} &= L_{async} \times L_{async} \times F \times Value^{*} \\
\kappa \in \widehat{ScheduledCallbacks} &= (\mathcal{P}(\widehat{Callback}))^{*} \\
\tau \in \widehat{ScheduledTimerIO} &= (\mathcal{P}(\widehat{Callback}))^{*} \\
\phi \in \widehat{QueueChain} &= (\mathcal{P}(L_{async}))^{*} \\
\sigma \in State &= Heap \times Stack \times \widehat{Queue} \times \widehat{ScheduledCallbacks} \times \widehat{ScheduledTimerIO} \times \widehat{QueueChain}
\end{aligned}
$$

Figure 4.2: Abstract domains of TAJS analysis that deal with asynchrony.

We extend TAJS analysis by introducing the abstract domains that deal with the asynchrony. These domains are adapted from Figure 3.2 so that they over-approximate the semantics of λ_{q}. The new domains are shown in Figure 4.2. Note that the domains not shown in that figure remain the same as Figure 4.1.

Abstract Queue

As a starting point, we extend the domain of abstract addresses; the new domain reflects the union of set L with the set which contains two internal addresses, i.e. l_{time}, l_{io}, corresponding to the addresses of the queue objects responsible for timers and asynchronous I/O respec-

tively. An abstract queue is defined as the partial map of abstract addresses to an element of the power set of queue objects. Therefore, an address may correspond to multiple queue objects. This abstraction over-approximates the behaviour of λ_q and allows us to capture all possible program behaviours that deal with queue objects. As a motivating example for this design choice, consider the following code fragment:

```
1 var x = new Promise(function(resolve, reject) {
2   if (AnyBool)
3     resolve("foo")
4   else
5     reject("bar")
6 })
```

In this example we create a new promise via the promise constructor. This promise may be either fulfilled with “foo” or rejected with “bar” depending on the value of variable `AnyBool` inside the `if` condition (line 2). If the exact value of `AnyBool` is unknown to the analysis (i.e. evaluates to $\top$), both paths should be taken into account (since the analysis is aimed to be sound). Therefore, the state of the analysis, and more specifically, the abstract queue needs to store the side-effects coming from both paths, i.e. the allocated address points to a fulfilled and a rejected queue object.

Abstract Queue Objects

An abstract queue object is represented by a tuple consisting of a queue state (observe that an abstract queue state has the same definition as λ_q), two sets of abstract callbacks (executed on fulfillment and on rejection respectively), and a set of abstract addresses (used to store the queue objects that are dependent on the current one). Notice how this definition differs from that of λ_q:

- First, we do not keep the registration order of callbacks, therefore, we convert the two lists of callbacks into two sets. The programming pattern related to promises underpins our design decision. Specifically, promises are commonly used as a chain; registering two callbacks on the same promise object is quite uncommon. Similar observations have been made for the event-driven programs by Madsen et al. (2015). That abstraction can negatively affect precision only when we register multiple callbacks on a *pending* queue objects. Recall from the semantics of λ_q (Figure 3.3) that when we register a callback on a settled queue object, we are able to precisely track its execution order since we add it to the list of scheduled callbacks.

- Second, we define the last component as a set of address; something that enables us to soundly track all possible dependent queue objects.

Abstract Callback

An abstract callback comprises two abstract addresses, one function, and a list of values which stands for the arguments with which the function is called. Note that the first address corresponds to `this` object, while the second one is the queue object that is fulfilled with the return value of the function.

Abstract List of Scheduled Callbacks

The abstract domains responsible for maintaining the callbacks which are scheduled for execution by the event loop, i.e. $\widehat{ScheduledCallbacks}$, and $\widehat{ScheduledTimerIO}$, are represented by a list of sets of callbacks. The ith element of a list denotes the set of callbacks that are executed after those placed at the $(i-1)^{th}$ position and before the callbacks located at the

$(i+1)^{th}$ position of the lists. The execution of callbacks of the same set is not known to the analysis; they can be called in any order. For example, consider the following sequence $[\{x\}, \{y,\ z\}, \{w\}]$, where $x, y, z, w \in \widehat{Callback}$. We presume that the execution of elements y, z succeeds that of x, and precedes that of w, but we cannot compare y with z, since they are elements of the same set; thus, y might be executed before z, and vice versa.

Note that a key requirement of the definition of those domains is that they should be finite so that the analysis is guaranteed to terminate. Keeping the lists of scheduled callbacks bound is tricky, because the same callback may be scheduled for execution multiple times, and therefore, it may be added to the list more than one time. As a motivating example, imagine the following code:

```
1  function foo() {
2    return Promise.resolve("foo")
3        .then(function bar(value) {
4            return value;
5        });
6  }
7
8  foo();
9  schedule a callback y
10 ...
11 foo();
12 schedule two callbacks z, w
13 ...
14 foo();
```

The first time we call function `foo()`, we schedule callback `bar()` through the invocation of `then()` at line 3. If we call function `foo()` again (line 11), the same callback is registered on the list; (i.e. the analysis is not able to distinguish them) because the function and its arguments remain the same, and the same queue object is fulfilled (as it originates from the same allocation site, that is, `then()` at line 3) when callback terminates. Calling function `foo()` n times results in the addition of the same callback n times. To guarantee the termination of the analysis, those lists compute the execution order of callbacks up to a certain limit n. The execution order of callbacks scheduled after that limit is not preserved, thus, they are all placed at the same set.

Now assume that in between the first and the second invocation of `foo()` at lines 8, 11, we schedule one more callback, i.e. $y \in \widehat{Callback}$, and in between the second and the third call of `foo()` at lines 11, 14, we schedule two callbacks, namely, $z, w \in \widehat{Callback}$ where z precedes w. The list of scheduled callbacks κ would be:

$$\kappa = [\{x\},\ \{y\},\ \{x\},\ \{z\},\ \{w\},\ \{x\}]$$

Knowing the first and the last occurrence of a callback in that sequence allows to determine the set of callbacks whose execution order is not known relatively to that callback. We exploit that information is Section 4.3, where we build a callback graph from a given list of scheduled callbacks.

Back to our example, since callback x can be found both before and after callbacks $\{y, z, w\}$, we presume that the execution order between pairs $\{(x, y), (x, z), (x, w)\}$ cannot be determined by the analysis. In other words, the analysis assumes that those pairs of callbacks are

executed in any order.

Abstract Queue Chain
The last component of our abstract domains is used to capture the effects of uncaught exceptions during the execution of callbacks and promise executors. It is defined as a sequence of sets of addresses. Based on the abstract translation of the semantics of λ_q, when the analysis reaches an uncaught exception, it inspects the top element of the abstract queue chain and rejects all the queue objects found in that element. If the abstract queue chain is empty, the analysis proceeds to the exceptional exit point of the current function as usual (Recall from Section 4.1.1 that every function has an exceptional exit point to propagate flow triggered by uncaught exceptions). Note that the queue chain is guaranteed to be *bound*. In particular, during the execution of a callback, the size of the abstract queue chain is *always 1*, because only one callback is executed at a time by the event loop. The only case when the size of the abstract queue chain is greater than 1 is when we have nested promise executors like the following scenario:

```
1 var x = new Promise(function() {
2     var y = new Promise(function() {
3         var z = new Promise(function() {
4             ...
5         })
6     })
7 });
```

However, since we cannot have an unbound number of nested promise executors, the size of the abstract queue chain is always finite.

Finally, we extend the definition of the abstract state which now consists of a heap, a stack, a queue, two sequences for storing the execution order of callbacks, (one for promises, and one for timers and asynchronous I/O), and a queue chain.

4.2.1 Tracking Execution Order

Promises

Estimating the order in which callbacks of promises are scheduled is straightforward, because it is a direct translation of the corresponding semantics of λ_q. In particular, there are two possible cases:

- *Settle a promise which has registered callbacks:* When we settle (i.e. either resolve or reject) a promise object which has registered callbacks, we schedule those callbacks associated with the next state of the promise by putting them on the tail of list κ. For instance, if we fulfill a promise, we append all the callbacks that are triggered on fulfillment on list κ. A reader might observe that if there are multiple callbacks registered on the promise object, we put them on the same set which is the element that is finally added to κ, since an abstract queue object does not keep the registration order of callbacks.

- *Register a callback on an already settled promise:* When we encounter a statement of the form `x.then(f1, f2)`, where `x` is a settled promise, we schedule either callback `f1` or `f2` (i.e. we add it to list κ) depending on the state of that promise, i.e. we schedule callback `f1` if `x` is fulfilled and `f2` if it is rejected.

Timers & Asynchronous I/O

A static analysis is not able to reason about the external environment. For instance, it cannot decide when an operation on a file system or a request to a server is complete. Similarly, it is not able to deal with time. For that purpose, we adopt a conservative approach for tracking the execution order between callbacks related to timers and asynchronous I/O. In particular, we assume that the execution order between those callbacks is unspecified, and thus, they are executed in any order. However, we *do* keep track the execution order between nested callbacks. For example, in the following code, the analysis knows that `baz()` is called after `bar()`, but it considers that pairs of `foo(), bar()`, and `foo(), baz()`, are invoked in any order.

```
1
2 setTimeout(function foo() {
3     // ...
4 });
5 setTimeout(function bar() {
6     setTimeout(function baz() {
7         // ...
8     });
9 });
```

4.3 Callback Graph

In this section, we introduce the concept of *callback graph*; an important component of our analysis that captures how data flow is propagated between different asynchronous callbacks. In Section 4.3.1, we formally define the notion of callback graph and then, we present how TAJS analysis is extended to construct callback graph for a given program. (Section 4.3.2)

4.3.1 Definition

A callback graph is defined as an element of the following power set[2]:

$$cg \in CallbackGraph = \mathcal{P}(Node \times Node)$$

where $n \in Node = C \times F$ is a node of a callback graph. Every element of a callback graph $(c_1, f_1, c_2, f_2) \in cg$, where $cg \in CallbackGraph$, means that function f_1 in context c_1 is executed *immediately before* function f_2 in context c_2. The above statement can be treated as the following expression: $f_1(); f_2();$

Definition 1 *Given a callback graph $cg \in CallbackGraph$, we define the binary relation $\rightarrow_{cg}$ on nodes of callback graph $n_1, n_2 \in Node$ as:*

$$n_1 \rightarrow_{cg} n_2 \Rightarrow (n_1, n_2) \in cg$$

Definition 2 *Given a callback graph $cg \in CallbackGraph$, we define the binary relation $\rightarrow^*_{cg}$ on nodes of callback graph $n_1, n_2 \in Node$ as:*

$$n_1 \rightarrow_{cg} n_2 \Rightarrow n_1 \rightarrow^*_{cg} n_2$$
$$n_1 \rightarrow^*_{cg} n_2 \wedge n_2 \rightarrow^*_{cg} n_3 \Rightarrow n_1 \rightarrow^*_{cg} n_3, \quad \text{where } n_3 \in Node$$

[2]We stick to the same notation as the definition of call graph.

Definition 1 and Definition 2 simply introduce the concept of *path* between two nodes in callback graph $cg \in CallbackGraph$. In particular, the relation $\rightarrow_{cg}$ denotes that there is path of length 1 between two nodes n_1, n_2, that is, $(n_1, n_2) \in cg$. On the other hand, the relation $\rightarrow^*_{cg}$ describes that there is a path of unknown length between two nodes. Relation $\rightarrow^*_{cg}$ is very important as it allows us to identify the *happens-before* relation between two nodes n_1, n_2, even if n_2 is executed long after n_1, that is $(n_1, n_2) \notin cg$. A significant property of a callback graph is that it does *not* have any cycles, i.e.

$$\forall n_1, n_2 \in Node.\, n_1 \rightarrow^*_{cg} n_2 \Rightarrow n_2 \not\rightarrow^*_{cg} n_1$$

Notice that if $n_1 \not\rightarrow^*_{cg} n_2$, and $n_2 \not\rightarrow^*_{cg} n_1$ hold, the execution order between n_1 and n_2 *cannot* be determined by the given callback graph, and therefore, we presume that n_1 and n_2 can be executed in any order.

4.3.2 Computing Callback Graph

Constructing Callback Graph

Algorithm 2 Algorithm for constructing a callback graph

Input: $\gamma \in (\mathcal{P}(\widehat{Callback}))^*$
Output: $cg \in CallbackGraph$
1: **function** BUILDCALLBACKGRAPH(γ)
2: $cg = \emptyset$
3: $p = \emptyset$
4: **for all** $i \in (1, \dots, n-1)$ **do**
5: $\mu = \emptyset$
6: **for all** $z \in \gamma_i$ **do**
7: $s =$ `getFunctionAndContext`(z)
8: **for all** $w \in \gamma_{i+1}$ **do**
9: $t =$ `getFunctionAndContext`(w)
10: **if** $\nexists n \in Node.\, t \rightarrow^*_{cg} n \land \nexists n' \in Node.\, n' \rightarrow^*_{cg} t$ **then**
11: $cg = cg \cup \{(s, t)\}$
12: **if** $p \neq \emptyset$ **then**
13: $cg = cg \cup \{(n, t) \mid n \in p\}$
14: $p = \emptyset$
15: **end if**
16: **else**
17: $S = \{n \mid \forall n \in Node.\, n \rightarrow_{cg} t\}$
18: $T = \{n \mid \forall n \in Node.\, t \rightarrow_{cg} n\}$
19: $cg = cg \setminus \{(t, n) \mid \forall n \in T\}$
20: $cg = cg \cup (S \times T)$
21: $\mu = \mu \cup \{t\}$
22: **end if**
23: **end for**
24: **end for**
25: $p = p \cup \mu$
26: **end for**
27: **end function**

A callback graph is constructed from a list of scheduled callbacks as shown in Algorithm 2. In particular, the algorithm expects a list of sets of abstract callbacks $\gamma \in (\mathcal{P}(\widehat{Callback}))^*$, and produces a callback graph $cg \in CallbackGraph$. Initially, the algorithm iterates over

every element of sequence γ and inspects the set of callbacks located at the ith and $(i+1)^{th}$ position respectively. Recall from Section 4.2 that according to the abstract semantics of $\gamma \in (\mathcal{P}(\widehat{Callback}))^*$, γ_i is the set of callbacks that are called before γ_{i+1}. Therefore, this information is reflected on the callback graph, by creating edges from every element of γ_i to every element of γ_{i+1} (in other words, we form $\rightarrow_{cg}$ relations). For that purpose, the algorithm iterates over every element of set γ_i and tries to connect it with every element of γ_{i+1} (Algorithm 2, lines 6–24). Specifically, firstly, the algorithm retrieves the function and the context $s, t \in Node$ from every abstract callback of γ_i, and γ_{i+1} respectively (Algorithm 2, lines 7, 9). Then, there are two possible cases:

Case 1 (lines 10–16): The target node t does not form any happens-before with any other node in the graph i.e. $\nexists n \in Node.\ t \rightarrow_{cg}^* n$, and $\nexists n' \in Node.\ n' \rightarrow_{cg}^* t$ This means that it is the first time we encounter target node t in list γ. Therefore, we simply add element $\{(s,t)\}$ to the current callback graph (Algorithm 2, line 11).

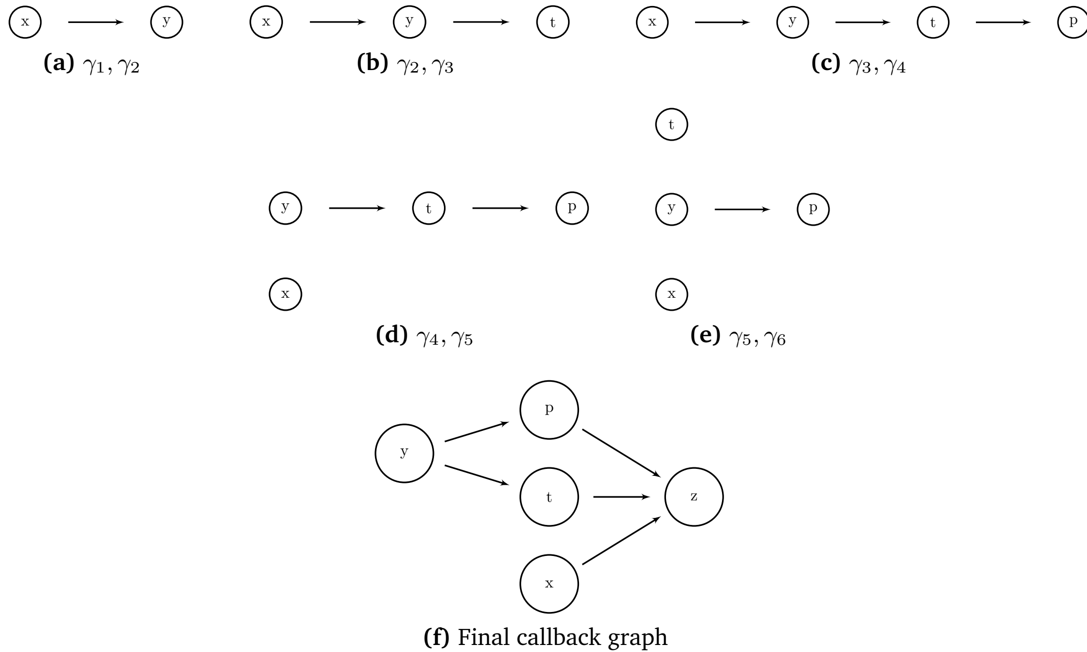
Case 2 (lines 16–22): The target node t is already found in the callback graph, and forms $\rightarrow_{cg}$ relations with other nodes In this case, we cannot add element (s,t) to the current callback graph, because such an addition creates a cycle. The presence of t implies that it has been previously processed in a position k, where $k < i$. In other words, the execution order between s and t is unspecified. Then, we first compute the set of nodes S (line 17) which have outgoing edges to t, i.e. $n \rightarrow_{cg} t$. After that, we compute the set of nodes T (line 18) with which t is directly connected, i.e. $t \rightarrow_{cg} n$. Using those sets, we perform two updates on callback graph:

- First, we remove all outgoing edges of target node t (Algorithm 2, line 19). Note that after that deletion, the following property holds: $\forall n \in Node.\ t \not\rightarrow_{cg}^* n$.
- Second, we create $\rightarrow_{cg}$ relations between every node of S, with every node of T (Algorithm 2, line 20). The reason for that update is that since all the elements of T have lost their predecessor which was t, (recall that we removed all the outgoing edges from t), we have to form happens-before relations with the predecessors of t, by connecting every predecessor of t (set S) with every element of successor of t (set T).

Also notice that we put node t to a separate set μ (Algorithm 2, line 21). The elements of this set are joint with that of p at line 25. Set p contains the nodes that we need to connect with the target nodes of the next iteration (Algorithm 2, lines 12–15) Since we remove all outgoing edges of node t at line 17, we have to form new $\rightarrow_{cg}$ relations with the next target nodes, i.e. those nodes located at the $(i+2)^{th}$ position of γ.

Example. Consider the following list of scheduled callbacks: $[\{x\}, \{y\}, \{t\}, \{p\}, \{x\}, \{t\}, \{z\}]$, where $x, y, t, p, z \in \widehat{Callback}$. The steps that demonstrate the building process of a callback graph from that list (with regards to Algorithm 2) are shown in Figure 4.3[3]. Each sub-figure illustrates the resulting callback graph after connecting the elements of γ_i with those of γ_{i+1}. First, observe the callback graph of Figure 4.3d, i.e. the callback graph computed after trying to connect γ_4 with γ_5. At that point, we want to connect callback p with x. However, by inspecting the graph of Figure 4.3c (which is the current graph of that step), we notice that x is already in the graph, therefore, we presume that the execution order between x and p

[3]For the purposes of demonstration, we label each node in the graph using the identifier of each abstract callback.



(a) γ_1, γ_2 (b) γ_2, γ_3 (c) γ_3, γ_4

(d) γ_4, γ_5 (e) γ_5, γ_6

(f) Final callback graph

Figure 4.3: Steps demonstrating the building process of a callback graph from list $[\{x\}, \{y\}, \{t\}, \{p\}, \{x\}, \{t\}, \{z\}]$. Each sub-figure illustrates the resulting callback graph after connecting the elements of γ_i with those of γ_{i+1}.

is not known. Therefore, we remove all the outgoing edges of x. Using the same reasoning, we remove all the outgoing edges of t at the next step, where we try to connect node x with t. Also, we connect node y, which is the predecessor of t, with p (this update corresponds to line 20 of Algorithm 2). After those modifications, we end up with the callback graph of Figure 4.3e. Note that at both cases, nodes t and x are stored to a special set in order to be connected with the next possible target nodes. In our case that target node is z: all nodes of set γ_6 along with x and t are connected with z, leading to the final graph of Figure 4.3f.

Figure 4.3f presents the final callback graph after processing all elements of list γ. By inspecting that graph, we make a lot of observations. First, we observe that there is not any path between the following pairs $\{(x,t), (t,p), (x,p), (y,x)\}$, thus, their execution order cannot be determined; in other words, they might be executed in any order. Indeed, this is also reflected on the given list, e.g. x appears both before and after callback y. Similar observations can be made for the rest of the pairs. Second, the graph correctly captures the temporal relation between nodes y, t, p, z, that is, t and p are called before z, and y is executed before t and p. Third, we also conclude that z is the last node that is invoked, because it does not have any outgoing edges and there is a path from any other node in the graph to z.

Extending Analysis Output

Callback graph (like call graph) is computed on the fly. To this end, the definition of $AnalysisLattice$ domain presented in Figure 4.1 is extended as follows:

$$\alpha \in AnalysisLattice = (C \times N \hookrightarrow State) \times CallGraph \times CallbackGraph$$

In other words, the output of the analysis is now a triple consisting of a map of all reachable

Algorithm 3 Updating callback graph during the analysis of the event loop.

Input: $\alpha \in AnalysisLattice, \sigma_{new} \in State, n \in N, c \in C$
1: $\kappa_{new} = \sigma_{new} \downarrow_4$
2: $\tau_{new} = \sigma_{new} \downarrow_5$
3: $\sigma = \alpha \downarrow_1 (c, n)$
4: $\kappa = \sigma \downarrow_4$
5: $\tau = \sigma \downarrow_5$
6: $cg = \alpha \downarrow_3$
7: $\kappa_d = \texttt{diff}(\kappa, \kappa_{new})$
8: **if** $\kappa_d \neq []$ **then**
9: $\quad cg' = \text{BUILDCALLBACKGRAPH}(\kappa_d)$
10: $\quad cg = \texttt{append}(cg, cg')$
11: **else**
12: $\quad \tau_d = \texttt{diff}(\tau, \tau_{new})$
13: $\quad$ **if** $\tau_d \neq []$ **then**
14: $\quad\quad cg' = \text{BUILDCALLBACKGRAPH}(\tau_d)$
15: $\quad\quad cg = \texttt{append}(cg, cg')$
16: $\quad$ **end if**
17: **end if**

program points to their states, a call graph, and a callback graph. In this way, the computed callback graph can be used by other client analyses as we shall discuss in Section 6.2.2.

Callback graph is built gradually as the analysis of the event loop progresses. In particular, during the analysis of the event loop, we update the current callback graph with regards to any new callbacks. Algorithm 3 explains how callback graph is incrementally constructed[4]. Specifically, when we analyze the event loop in a particular state σ_{new}, we get the corresponding lists of scheduled callbacks κ_{new}, τ_{new} from that state. Then by inspecting the element $\alpha \in AnalysisLattice$, we retrieve the current state of the event loop and its respective lists of callbacks (Algorithm 3, lines 3–5). Initially, the algorithm compares the elements κ_{new}, κ at line 7, getting the difference of those lists. If there is a change, (i.e. $\kappa_d \neq []$), it means that there are new scheduled callbacks since the last time when the event loop was analysed. For that reason, we build a callback graph from list κ_d by exploiting the Algorithm 2 (Recall Section 4.3.2), and then we append the resulting graph cg' to the current one at line 10 (i.e. `append` function connects all the sink nodes[5] of cg, with all the source nodes[6] of cg')

If the condition at line 8 does not hold, the algorithm follows the same procedure as before, but this time it uses lists τ_{new}, τ which hold callbacks associated with timers and asynchronous I/O (Algorithm 2, lines 11–16). Again, if there is any difference between τ_{new} and τ, the algorithm builds a callback graph from their difference and appends to it the current callback graph (Algorithm 3, lines 13–15). Notice that the behaviour of this algorithm correctly follows the semantics of λ_{q}, i.e. the promise-related callbacks are executed before the callbacks associated with timers and non-blocking I/O.

[4] This algorithm is part of the transfer function related to the event loop.
[5]nodes with only incoming edges.
[6]nodes with only outgoing edges.

```
1 var z = undefined;
2 var x = Promise.resolve()
3   .then(function bar() {
4     z = {foo: 1};
5     return 2;
6   })
7   .then(function baz(value) {
8     return value + z.foo;
9   });
```

Figure 4.4: Example which illustrates the importance of keeping track of the correct execution order between callbacks.

4.4 Analysis Sensitivity

4.4.1 Callback-Sensitivity

Knowing the temporal relations between asynchronous callbacks is an important fact of the analysis, as it enables us to correctly capture how data flow is propagated. Typically, a naive flow-sensitive analysis that exploits the CFG represents the event loop as as single program point with only one context corresponding to it; therefore, unlike traditional function calls, the analysis misses the happens-before relation between callbacks because they are triggered by the same program location (i.e. the event loop). Indeed, many previous works stuck to that and conservatively considered that all registered asynchronous callbacks can be executed in any order (Jensen et al., 2011; Park et al., 2016; Kashyap et al., 2014). However, such an approach may lead to imprecision and false positives.

Motivating Examples

Starting with a simple example, consider the code fragment of Figure 4.4. Initially, we schedule asynchronous callback `bar()` (line 3). While `bar()` executes, we initialise global variable `z` as `z = {foo: 1}` (line 4), and then, function returns with an integer (line 5). Upon exit of `bar()`, callback `baz()` is scheduled for execution. Observe that `baz()` accesses the property `foo` of global variable `z` at line 8. An analysis that misses the temporal relation between `bar()` and `baz()` will report a type error at line 8, because it assumes that `baz()` may be executed first, and thus, a property access of an undefined variable may occur at that line.

The second example comes from `honoka`[7], a real-world JavaScript module. Figure 4.5 shows a code snippet where a naive analysis reports a false positive (we omit irrelevant code for brevity). Line 1 performs an asynchronous request. In particular, function `asyncRequest()` returns a promise which is fulfilled when the asynchronous request succeeds. At lines 2–21, we create a promise chain. The first callback of this chain (lines 5–14) clones the response of the request, and assigns it to property `response` of object `honoka` (line 3). Then, it parses the body of response according to the specified type and returns it to the next callback (lines 5–12). The second callback (lines 15–21) retrieves the headers of the response which has been previously assigned to `honoka.response` and if the content type is “application/json”, it converts the data of response into a JSON object (lines 17–20). It is easy to see that an analysis, which does not respect the execution order between the first and the second callback, will

[7] `https://github.com/kokororin/honoka`

```
1  asyncRequest(url, options)
2    .then(function (response) {
3      honoka.response = response.clone();
4
5      switch (options.dataType.toLowerCase()) {
6        case "arraybuffer":
7          return honoka.response.arrayBuffer();
8        case "json":
9          return honoka.response.json();
10       ...
11       default:
12         return honoka.response.text();
13     }
14   })
15   .then(function (responseData) {
16     if (options.dataType === "" || options.dataType === "auto") {
17       const contentType = honoka.response.headers.get("Content-Type");
18       if (contentType && contentType.match("/application\/json/i")) {
19         responseData = JSON.parse(responseData);
20       }
21     }
22     ...
23   });
```

Figure 4.5: Real-world example taken from honoka.

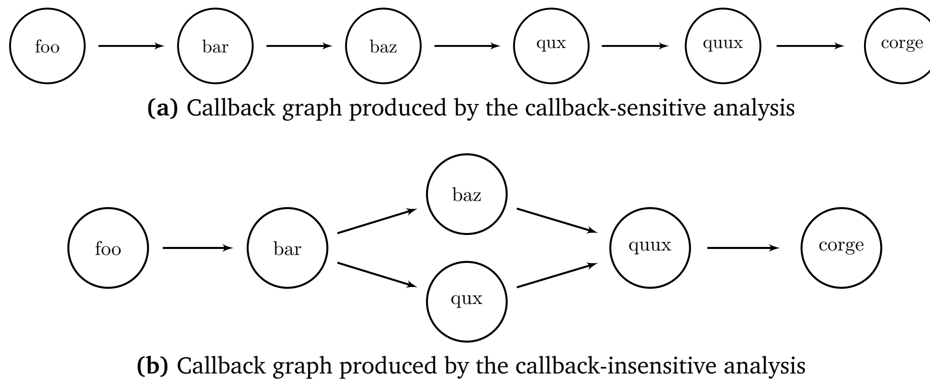
report a type error at line 17. Specifically, the analysis assumes that the callback defined at lines 15–21 might be executed first; therefore, `honoka.response`, which is assigned at line 3, might be uninitialised.

Leveraging Callback Graph

To address those issues, we exploit callback graph to devise a more precise analysis, which we call *callback-sensitive* analysis. The callback-sensitive analysis propagates state with regards to the $\rightarrow_{cg}$ and $\rightarrow^*_{cg}$ relations found in a graph $cg \in CallbackGraph$. Specifically, when the analysis needs to propagate a resulting state from the exit point of a callback x, instead of propagating that state to the caller (note that the caller of a callback is the event loop), it propagates it to the entry points of the next callbacks (i.e. all callbacks y that $x \rightarrow_{cg} y$ holds). In other words, the edges of callback graph reflect how state is propagated from the exit point of a callback x to the entry point of a callback y. Obviously, if there is not any path between two nodes in the graph, that is, $x \not\rightarrow^*_{cg} y$, and $y \not\rightarrow^*_{cg} x$, we propagate the state coming from the exit point of x to the entry point of y and vice versa. In other words, we assume that functions can be executed in any order.

Note that the callback-sensitive analysis also plays an important role on the precision of the computed callback graph. We define the *precision* of callback graph as the quotient between the number of callback pairs whose execution order is determined, and the total number of callback pairs. As a motivating example, which illustrates the positive influence of the callback-sensitive analysis on the precise callback graph construction, consider the following code snippet:

Listing 4.1: An example program which creates two promise chains.



(a) Callback graph produced by the callback-sensitive analysis

(b) Callback graph produced by the callback-insensitive analysis

Figure 4.6: Callback graph of program of Listing 4.1 produced by the callback-sensitive and the callback-insensitive analysis respectively.

```
1  var x = Promise.resolve().then(function foo() {
2    return "value";
3  }).then(function baz() {
4    return "value";
5  }).then(function quux() {
6    return "value";
7  }).then(function corge(value) {
8    ...
9  });
10
11
12 var y = Promise.resolve().then(function bar() {
13   return "value";
14 }).then(function qux(value){
15   ...
16 });
```

In this example, we create two chains of promises. According to the concrete execution of that program, the callbacks are invoked in the following order: $(\texttt{foo}() \rightarrow \texttt{bar}()) \rightarrow (\texttt{baz}() \rightarrow \texttt{qux}()) \rightarrow (\texttt{quux}()) \rightarrow (\texttt{corge}())$. Note that we use brackets to denote callbacks that are called in the same iteration of the event loop. Figure 4.6 presents the callback graphs of Listing 4.1 produced by the callback-sensitive and callback-insensitive analysis. Note that the callback-insensitive analysis is the naive analysis which considers that all callbacks can be executed in any order. By examining Figure 4.6, we immediately observe that the callback-sensitive analysis (Figure 4.6a) produces a callback graph that accurately captures the execution order of callbacks. On the other hand, the callback-insensitive analysis (Figure 4.6b) is not able to determine the precise execution order between callbacks `baz()` and `qux()`. The cause of this inaccuracy is that the analysis assumes that all callbacks are executed in any order. Therefore, when we analyse callbacks `foo()`, and `bar()`, all data flow is merged, since their relative execution order is unspecified. As a result, we end up with an inaccurate list $\kappa \in \widehat{ScheduledCallbacks}$, which cannot decide the exact execution order between `baz()` and `qux()`, which are scheduled by the side-effects of `foo()` and `bar()`.

Remark: Callback-sensitivity does not work with contexts to improve the precision of the analysis, therefore the event loop is still represented as a single program point. As a result, the state produced by the last executed callbacks is propagated to the event loop, leading to the join of this state with the initial one. The join of those states is then again propagated

across the nodes of the callback graph until convergence. Therefore, there still some kind of imprecision. However, callback-sensitivity minimises the number of those joins, as they stem only from the callbacks invoked last.

4.4.2 Sources of Imprecision

Since callback graph is constructed from the lists of scheduled callbacks $\kappa \in \widehat{ScheduledCallbacks}$ and $\tau \in \widehat{ScheduledTimerIO}$, the way we abstract the order in which callbacks are scheduled highly affects the precision of callback graph. We identify two major sources of imprecision during callback graph construction:

Timers and Asynchronous I/O: Our analysis loses the happens-before relations between callbacks related to timers and asynchronous I/O. However, this source of imprecision is sensible and inevitable to some extent. Since a static analysis like ours cannot reason about time and external environment, we have to follow a conservative approach. Recall from Section 4.2.1, however, that our analysis correctly captures the execution order between nested callbacks.

Programming Patterns: Different programming patterns may harm the effectiveness of the analysis, leading to imprecise callback graphs. We provide three code fragments as motivating examples in Figure 4.7. In Listing 4.2, we have a function `foo()` that expects a promise object as argument, and then registers a new callback on it at line 2. If we call function `foo()` multiple times, we lose the execution order between `foo()` and any other callback scheduled in between the invocations of `foo()`, since the analysis is not able to separate the first registration from the second one. Listing 4.3 presents a similar scenario to that of Listing 4.2. This time function `foo()` creates a new promise during its execution. For the same reason as before, we might lose the execution order between the callback scheduled by the two calls of `foo()` and any other callback scheduled in the meantime. Listing 4.4 is an example of scheduling the same callback multiple times in a promise chain. In that case, we cannot determine the temporal relation between `foo()`, and `bar()`, because `foo()` is scheduled both before and after `bar()` (Listing 4.4, lines 6, 10).

4.4.3 Context-Sensitivity Strategies

Recall from Section 4.3.1 that a callback graph is defined as $\mathcal{P}(Node \times Node)$, where $n \in Node = C \times F$. So, it is possible to increase the precision of callback graph (and therefore the precision of the analysis) by distinguishing callbacks based on the context in which they are scheduled. A reader might notice that existing context-sensitivity strategies are not so effective in differentiating callbacks from each other. For instance, object-sensitivity is not fruitful at all, because in most cases, `this` object corresponds to the global object. On the other hand, creating context according to the arguments of a function (i.e. parameter sensitivity) might not be applicable in some cases. For example, in Listing 4.4, callback `foo()` is called without any arguments. Thus, a parameter-sensitivity strategy cannot distinguish the first invocation of `foo()` from the second one, as the context remains the same.

To address those issues, and enhance the accuracy of the analysis for some of the programming patterns shown in Figure 4.7, we introduce two novel context-sensitivity strategies that are specifically targeted for our analysis: *R-sensitivity*, and *Q-sensitivity*. Also, we define *QR-sensitivity*, which is the combination of those two.

Listing 4.2: Example 1

```
1  function foo(promise) {
2    return promise.then(
         function () {
3        ...
4    })
5  }
6
7  var x = Promise.resolve();
8  foo(x);
9  ...
10 var y = Promise.resolve();
11 foo(y);
```

Listing 4.3: Example 2

```
1  function foo() {
2    return Promise.resolve()
3    .then(function () {
4      ...
5    });
6  }
7
8  foo();
9  ...
10 foo();
```

Listing 4.4: Example 3

```
1  function foo() {
2    ...
3  }
4
5  var x = Promise.resolve()
6    .then(foo)
7    .then(function bar() {
8      ...
9    })
10   .then(foo)
```

Figure 4.7: Examples of different programming patterns that might lead to an imprecise callback graph.

R-Sensitivity

The R-sensitivity (R from return) separates callbacks according to the queue object that their return value fulfills or rejects. The domain of context of this strategy is defined as:

$$c \in C = L_{async}$$

Notice that this domain is finite, thus, the analysis is guaranteed to terminate. This strategy is effective in cases where we schedule the same callback at different program locations. For instance, the R-sensitivity analysis produces precise results for the program of Listing 4.4. Specifically, the analysis is able to distinguish the first call of `foo()` from the second one, as the former fulfills the queue object allocated at line 6 (i.e. allocated by `then()` function), whereas the latter settles the object coming from line 10.

Q-Sensitivity

The Q-sensitivity (Q from queue) separates callbacks based on the queue object to which they are added. The domain of contexts is given by the following power set:

$$c \in C = L_{async}$$

Notice that the Q-sensitivity strategy also ensures the termination of the analysis, because the domain of contexts remains bound. The Q-sensitivity strategy produces good results

when it is applied on callbacks which are registered on different queue objects. For example, the Q-sensitivity leads to a more precise analysis, when it is applied on the programs of Listing 4.2 and Listing 4.4. In both programs, we schedule the same callback multiple times; we add it to different queue objects each time though.

QR-sensitivity

The QR-sensitivity strategy combines both the R- and Q-sensitivity strategies. It distinguishes every function using pairs of that form: (l_q, l_r), where l_q stands for the allocation site that the queue object on which they are registered, may point to, and l_r is the abstract address of the queue object they fulfill. Therefore, the finite domain of contexts is defined as follows:

$$c \in C = L_{async} \times L_{async}$$

This strategy has all the advantages of the R- and Q-sensitivity, so, it is effective in cases where at least one of the R- or Q-sensitivity strategy is. For example, consider the following program:

```
1  function bar() {
2    ...
3  }
4
5  function foo(promise) {
6    return promise.then(function baz() {
7      ...
8    })
9  }
10
11 var z = Promise.resolve();
12 var x = Promise.resolve();
13 var y = Promise.resolve();
14
15 foo(x);
16 z.then(bar);
17 foo(y);
18 z.then(bar);
```

The QR-sensitivity is capable of producing precise results, because it is able to separate the different calls of callback `baz()` (line 6), because it is registered on different queue objects. At the same time, it can distinguish the different calls of callback `baz()` (lines 16, 18) from each other, as each invocation fulfills a different queue object. Applying R- and Q-sensitivity strategy individually results in less accurate analysis, as they can distinguish the invocations of one callback only.

However, observe that there are still cases when all these context-sensitivity strategies fail to separate data flow. For example, they cannot positively affect the accuracy of the analysis of Listing 4.3; every time we schedule the callback at line 3, it corresponds to and fulfills the same queue object.

4.5 Implementation Details

Prior to our extensions, TAJS consisted of approximately 83,500 lines of Java code. The size of our additions is roughly 6,000 lines of Java code. Our implementation is straightforward

```
1  function open(filename, flags, mode, callback) {
2      TAJS_makeContextSensitive(open, 3);
3      var err = TAJS_join(TAJS_make("Undef"), TAJS_makeGenericError());
4      var fd = TAJS_join(TAJS_make("Undef"), TAJS_make("AnyNum"));
5      TAJS_addAsyncIOCallback(callback, err, fd);
6  }
7
8  var fs = {
9    open: open
10   ...
11 }
```

Figure 4.8: A model for `fs.open` function.

and is guided by the design of our analysis. Specifically, we first extend the definition of state by implementing and adding the definitions of our domains that deal with asynchrony. Then, we provide models for promises written in Java by faithfully following the ECMAScript specification (ECMA-262, 2015). Note that our models exploit the abstract domains presented in Table 4.2 and they produce side-effects that over-approximate the behaviour of promises specified in ECMAScript. Beyond that, we implement models for the special constructs of λ_q, i.e. `addTimerCallback`, `addIOCallback` (Recall Section 3.2), which are used for adding callbacks to queue objects which are responsible for timers and asynchronous I/O respectively. The models for timers are implemented in Java. However, we write JavaScript models for asynchronous I/O operations, when it is necessary.

For example, Figure 4.8 shows the JavaScript code that models function `open()` from Node.js module `fs`. In particular, `fs` asynchronously opens a given file. When I/O operation completes, the callback provided by the developer is called with two arguments: 1) `err` which is not `undefined` if there is an error during I/O, 2) `fd` which is an integer indicating the file descriptor of the opened file. Note that `fd` is `undefined`, if any error occurs. Our model first makes `open()` parameter-sensitive on the third argument which corresponds to the callback provided by the programmer. Then, at lines 3,4, it initialises the arguments of the callback, i.e. `err`, `fd`. Observe that we initialise those arguments so that they capture all the possible execution scenarios, i.e. `err` might be `undefined` or point to an error object, and `fd` might be `undefined` or any integer reflecting all possible file descriptors. Finally, at line 5, we call special function `TAJS_addAsyncIOCallback()`, which registers the given callback on the queue object responsible for I/O operations, implementing the semantics of `addIOCallback` of our λ_q calculus.

Remark: All functions starting with `TAJS_` are special functions whose body does not correspond to any node in the CFG. They are just hooks for producing side-effects to the state or evaluating to some value, and their models are implemented in Java. For instance, `TAJS_make("AnyStr")` evaluates to a value that can be any string.

Also, we implement the transfer function for the event loop. This transfer function inspects the lists of scheduled callbacks from the current state and it updates callback graph on the basis of them. Then, it propagates the current state to the entries of the functions that should be executed first according to the current callback graph. If callback-sensitivity is disabled, the event loop simply propagates the current state to the entries of all registered callbacks until convergence.

Chapter 5

Empirical Evaluation

In this chapter we evaluate our static analysis on a set of hand-written micro benchmarks and a set of real-world JavaScript modules. Then, we experiment with different parameterisations of the analysis and report precision and performance metrics.

5.1 Experimental Setup

To test that our technique behaves as expected we first wrote a number of micro benchmarks. Each of those programs consists of approximately 20–50 lines of code and examines certain parts of the analysis. For instance, Figure 5.1 shows the code of a micro benchmark used in our evaluation process. That program demonstrates a complex scenario; specifically, at line 24, we create a promise chain. The callback registered on the first promise calls function `foo()` (line 25). If we examine the body of `foo()`, we immediately notice that the return value of `foo()` is a promise object, meaning that the promise allocated by the call of `then()` at line 24 is settled whenever the returned promise of `foo()` is settled. Observe that we have created imprecision on purpose. In particular, at line 4, function `TAJS_make("AnyBool")` evaluates to $\top$ (i.e. evaluates to any boolean), thus, a sound analysis should take both branches into account. The `if` branch (lines 5–12) returns a promise chain, whereas the `else` branch (lines 14–19) creates a promise which is fulfilled with a thenable object; therefore, according to the semantics of the ECMAScript (ECMA-262, 2018), it is fulfilled asynchronously.

Beyond micro benchmarks, we evaluate our analysis on 6 real-world JavaScript modules, using both hand-written and real test cases. The most common macro benchmarks used in literature are those provided by JetStream[1], and V8 engine[2] (Jensen et al., 2009; Kashyap et al., 2014; Ko et al., 2016). However, those benchmarks are not asynchronous; thus, they are not suitable for evaluating our analysis. In order to find interesting benchmarks, we developed an automatic mechanism for collecting and analyzing Github repositories. Specifically, first, we collected a large number of Github repositories using two different options. The first option extracted the Github repositories of the most depended upon `npm` packages[3] by using a web scraper developed by us. The second option employed Github API[4] to find

[1] `https://browserbench.org/JetStream/`
[2] `http://www.netchain.com/Tools/v8/`
[3] `https://www.npmjs.com/browse/depended`
[4] `https://developer.github.com/v3/`

```
1  var t;
2
3  function foo() {
4    if (TAJS_make("AnyBool")) {
5      return Promise.resolve("foo").then(
6        function fun2() {
7          t = {bar: 1};
8          return "foo";
9        }
10     ).then(function fun3(value) {
11       throw "baz";
12     });
13   } else {
14     return Promise.resolve({
15       then: function fun4(res) {
16         t = {bar : 1};
17         res("bar");
18       }
19     });
20   }
21 }
22
23 var x = Promise.resolve("bar");
24 x.then(function fun1(value) {
25   return foo();
26 }).then(function fun5(value) {
27   t.bar; // no error;
28 }).catch(function fun6(value) {
29   // This error is propagated from fun3.
30 });
```

Figure 5.1: An example of a micro benchmark used in our experiments.

JavaScript repositories which are related to promises.

We further investigated the Github repositories which we collected at the first phase by computing various metrics such as lines of code, number of promise-, timer- and asynchronous IO-related statements. In turn, by inspecting those metrics, we manually selected the 6 JavaScript modules presented in Table 5.1. Most of them are libraries for performing HTTP requests or file system operations in an asynchronous manner. Each benchmark is described by its lines of code (LOC), its lines of code including its dependencies (ELOC), number of files, number of dependencies, number of promise-related statements (e.g. `Promise.resolve()`, `Promise.reject()`, `then()`, etc.), and number of statements associated with timers (e.g. `setTimeout()`, `setImmediate()`, etc.) or asynchronous I/O (e.g. asynchronous file system or network operations etc.).

We evaluate the accuracy of the analysis in terms of number of analysed callbacks, precision of computed callback graph, and number of reported type errors. Recall from 4.4 that the precision of callback graph is the quotient between the number of callback pairs whose execution order is determined and the total number of callback pairs. Also, we embrace a client-based precision metric, namely, the number of reported type errors as in the work of Kashyap et al. (2014). The fewer type errors an analysis reports, the more precise it is. The same applies to the number of callbacks inspected by the analysis; less callbacks indicates

Benchmark	LOC	ELOC	Files	Dependencies	Promises	Timers/Async I/O
controlled-promise	225	225	1	0	4	1
fetch	517	1,118	1	1	26	2
honoka	324	1,643	6	6	4	1
axios	1,733	1,733	26	0	7	2
pixiv-client	1,031	3,469	1	2	64	2
node-glob	1,519	6,131	3	6	0	5

Table 5.1: List of selected macro benchmarks and their description.

a more accurate analysis. To compute the performance characteristics of every analysis, we re-run every experiment ten times in order to get reliable measurements. All the experiments were run on a machine with an Intel i7 2.4`GHz` quad core processor and 8`GB` of RAM.

5.2 Micro Benchmarks

For micro benchmarks, we experiment with 8 different analyses that compute callback graph. Specifically, we first turn off callback-sensitivity and then we consider every possible combination of our new context-sensitivity strategies, that is, a context-insensitive analysis (No), a R-sensitive analysis (R), a Q-sensitive analysis (Q), a both R- and Q-sensitive analysis (QR). We repeat the same procedure, namely, we test every possible context-sensitivity strategy, but this time, with callback-sensitivity enabled.

Table 5.2 and Table 5.3 show how every context-sensitivity policy performs on a callback-insensitive and callback-sensitive analysis respectively. Starting with Table 5.2, we observe that context-sensitivity improves the precision of callback graph. In particular, R- and Q-sensitivity increase the average accuracy by 2,3% and 3,6% respectively, indicating that Q-sensitivity is more effective than R-sensitivity for micro benchmarks. That small boost of context-sensitivity is explained by the fact that *only* 3 out of 29 micro benchmarks invoke the same callback multiple times. Recall from Section 4.4 that context-sensitivity is used to distinguish different calls of the same callback. Therefore, if one program do not use a certain callback multiple times, context-sensitivity does not make any difference. However, if we focus on the results of the micro benchmarks where we encounter such behaviours, i.e. micro18, micro19, and micro29, we get a significant divergence of the precision of callback graph. Specifically, context-sensitivity improves precision by 20,5%, 27,4% and 100% in micro18, micro19 and micro29 respectively. Besides that, in micro19, there is a striking increase of the number of analysed callbacks: a context-insensitive analysis inspects 14 callbacks compared to the context-sensitive analyses which examine only 7. The results regarding the number of type errors are almost identical for every analysis: a context-insensitive analysis reports 42 type errors in total, whereas all the other context-sensitive analyses produce warnings for 41 cases.

Moving to Table 5.3 the results of callback-sensitive analyses indicate clear differences. First, the naive callback-sensitive analysis (i.e. when all context-sensitivity strategies are disabled) reports only 16 type errors in total, and amplifies the average accuracy of callback graph from 0.85 to 0.91. As before, context-sensitive analyses boost the precision of callback graph by 20.4%, 35.1%, and 100% in micro18, micro19, and micro29 respectively. Recall again that only in those micro benchmarks, we register the same callback more than one time.

	Analysed Callbacks				**Callback Graph Precision**				**Type Errors**			
Benchmark	**No**	**R**	**Q**	**QR**	**No**	**R**	**Q**	**QR**	**No**	**R**	**Q**	**QR**
micro01	5	5	5	5	0.8	0.8	0.8	0.8	2	2	2	2
micro02	3	3	3	3	1.0	1.0	1.0	1.0	1	1	1	1
micro03	2	2	2	2	1.0	1.0	1.0	1.0	1	1	1	1
micro04	4	4	4	4	0.5	0.5	0.5	0.5	1	1	1	1
micro05	8	8	8	8	0.96	0.96	0.96	0.96	3	3	3	3
micro06	11	11	11	11	1.0	1.0	1.0	1.0	3	3	3	3
micro07	14	14	14	14	0.86	0.86	0.86	0.87	1	1	1	1
micro08	5	5	5	5	0.8	0.8	0.8	0.8	1	1	1	1
micro09	5	5	5	5	0.9	0.9	0.9	0.9	1	1	1	1
micro10	3	3	3	3	1.0	1.0	1.0	1.0	1	1	1	1
micro11	4	4	4	4	0.83	0.83	0.83	0.83	5	5	5	5
micro12	5	5	5	5	0.9	0.9	0.9	0.9	2	2	2	2
micro13	4	4	4	4	0.83	0.83	0.83	0.83	1	1	1	1
micro14	6	6	6	6	0.8	0.8	0.8	0.8	2	2	2	2
micro15	6	6	6	6	0.8	0.8	0.8	0.8	0	0	0	0
micro16	6	6	6	6	1.0	1.0	1.0	1.0	1	1	1	1
micro17	3	3	3	3	0.67	0.67	0.67	0.67	2	2	2	2
micro18	4	3	3	3	0.83	1.0	1.0	1.0	1	0	0	0
micro19	14	7	7	7	0.73	0.93	0.93	0.93	0	0	0	0
micro20	6	6	6	6	0.93	0.93	0.93	0.93	0	0	0	0
micro21	5	5	5	5	0.9	0.9	0.9	0.9	1	1	1	1
micro22	6	6	6	6	0.87	0.87	0.87	0.87	1	1	1	1
micro23	6	6	6	6	0.87	0.87	0.87	0.87	3	3	3	3
micro24	3	3	3	3	1.0	1.0	1.0	1.0	2	2	2	2
micro25	8	8	8	8	0.79	0.79	0.79	0.79	1	1	1	1
micro26	9	9	9	9	0.89	0.89	0.89	0.89	3	3	3	3
micro27	3	3	3	3	1.0	1.0	1.0	1.0	1	1	1	1
micro28	7	7	7	7	0.81	0.81	0.81	0.81	1	1	1	1
micro29	4	4	4	4	0.5	0.5	1.0	1.0	0	0	0	0
Average	**5.83**	**5.55**	**5.55**	**5.55**	**0.85**	**0.87**	**0.88**	**0.88**	**1.45**	**1.41**	**1.41**	**1.41**
Total	**169**	**161**	**161**	**161**					**42**	**41**	**41**	**41**

Table 5.2: Precision on micro benchmarks when callback-sensitivity is disabled.

	Analysed Callbacks				**Callback Graph Precision**				**Type Errors**			
Benchmark	**No**	**R**	**Q**	**QR**	**No**	**R**	**Q**	**QR**	**No**	**R**	**Q**	**QR**
micro01	4	4	4	4	1.0	1.0	1.0	1.0	0	0	0	0
micro02	3	3	3	3	1.0	1.0	1.0	1.0	0	0	0	0
micro03	2	2	2	2	1.0	1.0	1.0	1.0	0	0	0	0
micro04	4	4	4	4	0.5	0.5	0.5	0.5	1	1	1	1
micro05	7	7	7	7	1.0	1.0	1.0	1.0	0	0	0	0
micro06	11	11	11	11	1.0	1.0	1.0	1.0	1	1	1	1
micro07	13	13	13	13	1.0	1.0	1.0	1.0	0	0	0	0
micro08	5	5	5	5	0.8	0.8	0.8	0.8	0	0	0	0
micro09	4	4	4	4	1.0	1.0	1.0	1.0	0	0	0	0
micro10	3	3	3	3	1.0	1.0	1.0	1.0	1	1	1	1
micro11	4	4	4	4	0.83	0.83	0.83	0.83	5	5	5	5
micro12	5	5	5	5	1.0	1.0	1.0	1.0	0	0	0	0
micro13	3	3	3	3	1.0	1.0	1.0	1.0	0	0	0	0
micro14	5	5	5	5	1.0	1.0	1.0	1.0	0	0	0	0
micro15	6	6	6	6	1.0	1.0	1.0	1.0	0	0	0	0
micro16	6	6	6	6	1.0	1.0	1.0	1.0	0	0	0	0
micro17	3	3	3	3	0.67	0.67	0.67	0.67	2	2	2	2
micro18	4	3	3	3	0.83	1.0	1.0	1.0	1	0	0	0
micro19	14	7	7	7	0.74	1.0	1.0	1.0	0	0	0	0
micro20	6	6	6	6	1.0	1.0	1.0	1.0	0	0	0	0
micro21	4	4	4	4	1.0	1.0	1.0	1.0	0	0	0	0
micro22	5	5	5	5	0.9	0.9	0.9	0.9	0	0	0	0
micro23	5	5	5	5	1.0	1.0	1.0	1.0	1	1	1	1
micro24	3	3	3	3	1.0	1.0	1.0	1.0	1	1	1	1
micro25	8	8	8	8	0.79	0.79	0.79	0.79	0	0	0	0
micro26	7	7	7	7	1.0	1.0	1.0	1.0	1	1	1	1
micro27	3	3	3	3	1.0	1.0	1.0	1.0	1	1	1	1
micro28	7	7	7	7	0.81	0.81	0.81	0.81	1	1	1	1
micro29	4	4	4	4	0.5	0.5	1.0	1.0	0	0	0	0
Average	**5.45**	**5.17**	**5.17**	**5.17**	**0.91**	**0.92**	**0.94**	**0.94**	**0.55**	**0.52**	**0.52**	**0.52**
Total	**158**	**150**	**150**	**150**					**16**	**15**	**15**	**15**

Table 5.3: Precision on micro benchmarks when callback-sensitivity is enabled.

	Analysed Callbacks				Callback Graph Precision				Type Errors			
Benchmark	**NC-No**	**NC-QR**	**C-No**	**C-QR**	**NC-No**	**NC-QR**	**C-No**	**C-QR**	**NC-No**	**NC-QR**	**C-No**	**C-QR**
controlled-promise	6	6	6	6	0.866	0.905	0.866	0.905	3	3	2	2
fetch	22	22	19	19	0.829	0.956	0.822	0.972	10	10	8	8
honoka	8	8	6	6	0.929	0.929	1.0	1.0	2	2	1	1
axios	15	15	14	14	0.678	0.83	0.686	0.871	2	2	1	1
pixiv-client	18	18	17	15	0.771	0.803	0.794	0.863	15	15	15	14
node-glob	3	3	3	3	0.667	0.667	0.667	0.667	21	21	21	21
Average	**12**	**12**	**10.8**	**10.5**	**0.79**	**0.848**	**0.805**	**0.88**	**8**	**8**	**7.3**	**7.3**
Total	**72**	**72**	**65**	**63**					**53**	**53**	**48**	**47**

Table 5.4: Precision on macro benchmarks.

Finally, a naive callback-sensitive analysis decreases the total number of analysed callbacks from 169 to 158. Notice that if callback-sensitivity and context-sensitivity are combined together, the total number of callbacks is reduced by 11,2%.

Since micro benchmarks are pretty small programs, we did not observe significant differences between their running times, so, we omit a discussion on the performance of micro-benchmarks. However, an reader interested in those metrics is referred to Appendix B.

5.3 Macro Benchmarks

For macro benchmarks, we consider 4 different analyses: 1) an analysis which is neither callback- nor context-sensitive (NC-No), 2) a callback-insensitive but QR-sensitive analysis (NC-QR), 3) a callback-sensitive but context-insensitive analysis (C-No), and 4) a both callback- and QR-sensitive analysis (C-QR). Table 5.4 reports the accuracy metrics of every analysis and every benchmark. First, we make similar observations as micro benchmarks: in general, context-sensitivity leads to a more precise callback graph for 4 out of 6 benchmarks. The improvement ranges from 4,6% to 26,9%. On the other hand, callback-sensitive analyses contribute to fewer type errors for 5 out of 6 benchmarks reporting 47–48 type errors in total instead of 53. This small variation is justified by the fact that in practise only a small number of type errors are caused due to ordering violation of callbacks.

By examining the results for `node-glob` benchmark, we see that every analysis produces identical results. Recall from Table 5.1 that `node-glob` uses only timers and asynchronous I/O operations. Neither callback- nor context-sensitivity are effective for that kind of benchmarks, because we follow a conservative approach for modelling timers and asynchronous I/O, regardless of how callbacks are scheduled; we assume that a callback x and y are executed in any order, even if x might be registered before y. Therefore, keeping a more precise state does not lead to a more precise callback graph. Similarly, since every callback is registered on the same queue object (i.e. queue object responsible for timers or asynchronous I/O), context-sensitivity is not able to separate two calls of the same callback (Recall Section 4.4).

Table 5.5 gives the running times of every analysis on every macro benchmark. We notice that in some benchmarks (such as fetch) a more precise analysis may decrease the running times by 3%-18%. This is explained by the fact that a more precise analysis might compress state faster than an inaccurate analysis. For instance, in fetch, an imprecise analysis led to the analysis of 3 spurious callbacks, yielding to a greater analysis time. The results appears

	Average Time				Median			
Benchmark	**NC-No**	**NC-QR**	**C-No**	**C-QR**	**NC-No**	**NC-QR**	**C-No**	**C-QR**
controlled-promise	2.3	2.22	2.27	2.28	2.29	2.26	2.25	2.31
fetch	8.53	7.97	7.07	6.98	8.52	8.26	7.46	7.22
honoka	4.14	4.05	3.86	3.94	4.12	4.0	3.61	3.81
axios	6.99	7.86	6.74	8.32	7.02	8.0	6.94	8.37
pixiv-client	22.11	24.92	24.77	28.89	22.19	25.16	24.65	29.2
node-glob	15.55	16.71	15.46	14.47	16.62	16.71	16.17	15.74

Table 5.5: Times of different analyses in seconds.

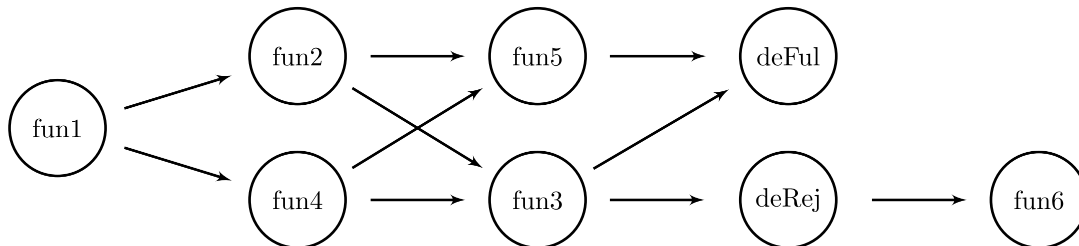


Figure 5.2: Callback graph corresponding to the micro benchmark of Figure 5.1

to be consistent with those of the recent literature which suggests that precision might lead to a faster analysis in some cases (Park et al., 2016). On the other hand, we also notice that analysis sensitivity increased the running times of `pixiv-client` by 12%-30,6%. However, such an increase seems to be acceptable.

5.4 Case Studies

In this section, we describe some case studies coming from both micro- and macro benchmarks.

micro25. Starting with micro benchmarks, Figure 5.2 presents the callback graph computed by the analysis of program of Figure 5.1. Function `fun1()` is the first callback executed by the event loop. Subsequently, either function `fun2()` or `fun4()` is invoked, corresponding to the callbacks scheduled at the `if` and `else` branches of function `foo()` respectively (Figure 5.1, lines 5, 14). Recall that we have injected inaccuracy on purpose (through function `TAJS_make("AnyBool")`); that is why both paths are considered. After those callbacks, we either schedule callback `fun5()` (triggered by the resolution of promise object allocated at line 14) or callback `fun3()` (triggered by `fun2()`). Callback `fun3()` (Figure 5.1, line 10) raises an exception, so, it schedules function `defaultReject()` (`deRej`) upon exit. Recall from Section 3.2 that if the second argument of `then()` is undefined, the `defaultReject()` function is provided instead. Also, note that since `fun3()` raises an exception, function `fun5()` cannot be executed after `fun3()`; `fun5()` is triggered only when the promise allocated at line 24 is fulfilled. That behaviour is correctly captured by the computed callback graph, since there is not any edge from `fun5()` to `deRej()`. Beyond that, after `fun5()`, we call `defaultFulfill()` (`deFul`) which is implicitly scheduled by the invocation of `catch()` at line 28 (Recall that `catch(f)` is syntactic sugar for `then(undefined, f)`). At the same time, `defaultReject()` triggers the execution of `fun6()`. It is easy to see that the resulting

```
1  function foo(promise) {
2      TAJS_makeContextSensitive(foo, 0);
3      promise.then(function bar() {
4        ...
5      });
6  }
7
8  var x = Promise.resolve();
9  foo(Promise.resolve());
10 x.then(function baz() {
11   ...
12 });
13
14 Promise.resolve().then(function qux() {
15   ...
16 });
17 foo(Promise.resolve());
18
19 Promise.resolve().then(function quux() {
20   ...
21 });
22 foo(Promise.resolve());
```

Figure 5.3: Program corresponding to micro29.

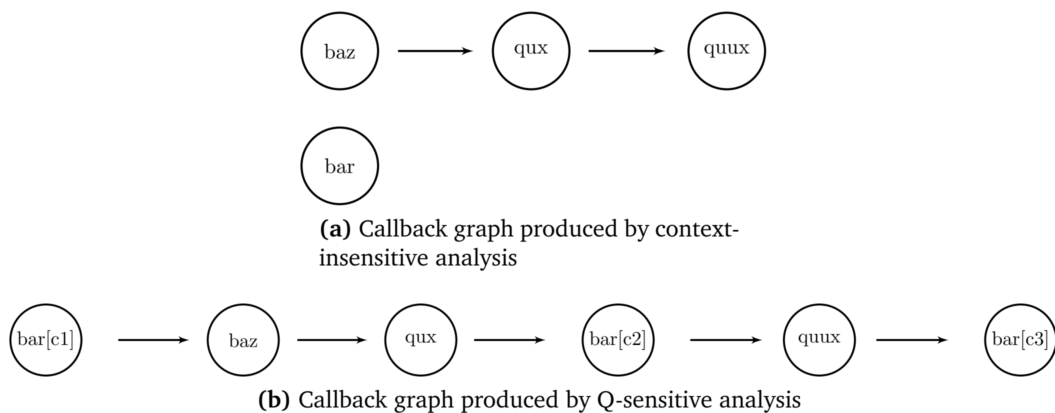
callback graph of Figure 5.2 is the join of two execution scenarios stemming from the imprecision of function `foo()`.

micro29. Figure 5.3 shows the code of micro benchmark 29. Function `foo()` expects a promise as parameter, and registers callback `bar()` on it (line 3). By examining the top level code, we observe that we invoke `foo()` three times by passing three different promises as a parameter each time (Figure 5.3, lines 9, 17, 22). Note that at line 2, function `TAJS_makeContextSensitive()` makes function `foo()` parameter-sensitive on the first parameter (i.e. parameter `promise`). Between every call of `foo()`, we schedule other callbacks, i.e. callbacks `baz()`, `qux()`, and `quux()`.

Figure 5.4a and Figure 5.4b present the callback graphs computed by a context-insensitive and a Q-sensitive analysis respectively. A context-insensitive analysis misses the execution order between `bar()` and the rest of callbacks, as it is not able to separate the first call of `bar()` from the second one, and so on. On the other hand, a Q-sensitive analysis distinguishes every invocation of `bar()` by creating a context based on the queue object on which that callback is registered. Since we add callback `bar()` to three different promises (queue objects), we create three different contexts (i.e. c1, c2, and c3); thus, we end up with a fully precicse callback graph.

fetch. Figure 5.5 shows a code fragment taken from fetch[5]. Note that we omit irrelevant code for brevity. Function `Body()` binds a couple of methods (e.g. `text()`, `formData()`) for manipulating the body of of response. Observe that those methods are registered on the prototype of `Response` using function `Function.prototype.call()` at line 42. Note that `Body` also contains a method, namely, `_initBody()` for initialising the body of a response

[5] `https://github.com/github/fetch`



(a) Callback graph produced by context-insensitive analysis

(b) Callback graph produced by Q-sensitive analysis

Figure 5.4: Callback graph of program of Figure 5.3 produced by context-insensitive and Q-sensitive analysis respectively.

according to the type of the input. To this end, the `Response` constructors takes a body as parameter and initialises it through `_initBody()` (line 50). Function `text()` reads the body of a response in a text format through promises (lines 29-37). If the body of the response has been already read, `text()` returns a rejected promise (lines 4, 25-26). Otherwise, it marks `bodyUsed` of the response object as true (line 6), and then it returns a fulfilled promise depending on the type of the given body (lines 29-37). Function `formData()` (lines 42-44) asynchronously reads the body of a response in a text format, and then it parses it into a `FormData` object[6] through function `decode()`. Function `fetch()` (lines 55-63) makes a new asynchronous request. When the request completes successfully, a callback `onLoad()` is executed asynchronously (line 59). That callbacks finally fulfills the promise returned by `fetch()` with a new response object initialised with the response of the server (line 61).

Case 1

Listing 5.1: Case 1: Using fetch API

```
1 fetch("/helloWorld").then(function foo(value) {
2   var formData = value.formData();
3   // Do something with form data.
4 })
```

In Listing 5.1, we make an asynchronous request to endpoint "/helloWorld" using fetch API. Upon success, we schedule callback `foo()`. Recall that the parameter `value` of `foo()` corresponds to the response object coming from line 61 (Figure 5.5). In `foo()`, we convert the response of the server into a `FormData` object (line 2).

A callback-insensitive analysis, which considers that all callbacks are scheduled in any order, merges the data flow of all callbacks. At this point, recall that the event loop is represented as single program point. For that reason, the side effects of `onLoad()` and `foo()` are propagated to the event loop. In turn, the event loop propagates again the resulting state to those callbacks. This is repeated until convergence. Specifically, callback `foo()` calls `value.formData()`, which updates the property `bodyUsed` of response object to true (Figure 5.5, line 6). The resulting state is propagated to the event loop where is joined with the state which stems from callback `onLoad()`. Notice that the state of `onLoad()` indicates that

[6] `https://developer.mozilla.org/en-US/docs/Web/API/FormData`

```
1  ...
2  function consumed(body) {
3      if (body.bodyUsed) {
4          return Promise.reject(new TypeError("Already read"));
5      }
6      body.bodyUsed = true;
7  }
8  ...
9
10 function Body() {
11   ...
12
13   this.bodyUsed = false;
14   this._bodyInit = function() {
15     ...
16     if (typeof body === "string") {
17           this._bodyText = body;
18     } else if (Blob.prototype.isPrototypeOf(body)) {
19         this._bodyBlob = body;
20     }
21     ...
22   }
23   this.text = function text() {
24         var rejected = consumed(this);
25         if (rejected) {
26             return rejected;
27         }
28
29         if (this._bodyBlob) {
30             return readBlobAsText(this._bodyBlob);
31         } else if (this._bodyArrayBuffer) {
32             return Promise.resolve(readArrayBufferAsText(this._bodyArrayBuffer));
33         } else if (this._bodyFormData) {
34             throw new Error("could not read FormData body as text");
35         } else {
36             return Promise.resolve(this._bodyText);
37         }
38     };
39
40   ...
41
42   this.formData = function formData() {
43        return this.text().then(decode);
44   }
45 }
46 ...
47
48 function Response(body) {
49   ...
50   this._bodyInit(body);
51 }
52 Body.call(Response.prototype);
53 ...
54
55 function fetch(input, init) {
56   return new Promise(function (resolve, reject) {
57     ...
58     var xhr = new XMLHttpRequest();
59     xhr.onload = function onLoad() {
60       ...
61       resolve(new Response(xhr.response));
62     }
63   });
64 }
```

Figure 5.5: Code fragment taken from fetch.

`bodyUsed` is false, because a fresh response object is initialised with property `bodyUsed` as false (Figure 5.5, line 13). The join of those states changes the abstract value of `bodyUsed` to $\top$. That change is propagated again to `foo()`.

This imprecision makes analysis to consider both `if` and `else` branches at lines 3-6. Thus, the analysis allocates a rejected promise at line 4, as it mistakenly considers that the body has been already consumed. This makes `consumed()` return a value that is either `undefined` or a rejected promise (line 26). The value returned by `consumed()` is finally propagated to `formData()` at line 43, where the analysis reports a false positive; a property access of an `undefined` variable (access of property "then"), because `text()` might return an undefined variable due to the return statement at line 26. A callback-sensitive analysis neither reports a type error at line 43 nor creates a rejected promise at line 4. It respects the execution order of callbacks, that is, callback `foo()` is executed *after* callback `onLoad()`. Therefore, the analysis propagates a more precise state to the entry of `foo()`, that is, a state resulted by the execution of `onLoad()`, where a new response object is initialised with field `bodyUsed` as false. In other words, the side effects of `foo()` are not taken into account.

Case 2.

Listing 5.2: Case 2: Using fetch API

```
1  ...
2  var response = new Response("foo=bar");
3  var formData = response.formData();
4  ...
5
6  var response2 = new Response(new Blob("foo=bar"));
7  var formData2 = response2.formData();
```

In Listing 5.2, we initialise a response object with a body of string type (line 2). In turn, by calling `formData()` method, we first read the body of the response in text format, and then we decode it into a `FormData` object by asynchronously calling `decode()` function (Figure 5.5, line 43). Since the body of the response is already in a text format, `text()` returns a fulfilled promise (Figure 5.5, line 36). At the same time, at line 5 of Listing 5.2, we allocate a fresh response object, whose body is an instance of `Blob`[7]. Therefore, calling `formData()` schedules function `decode()` again. However this time, callback `decode()` is registered on a different promise, because `text()` returns a promise created by function `readBlobAsText()` (Figure 5.5, line 30). A context-sensitive analysis (which creates context according to the queue object on which a callback is registered, such as Q-sensitivity or QR-sensitivity) is capable of separating the two invocations of `decode()`, because the first call of `decode()` is registered on the promise object which comes from line 36, whereas the second call of `decode()` is added to the promise created by `readBlobAsText()` at line 30.

honoka. We return back to Figure 4.5. Recall that a callback-insensitive analysis reports a spurious type error at line 17 when we try to access property `headers` of `honoka.response`, because it considers the case when the callback defined at lines 15–21 might be executed before the callback defined at lines 2–14; thus, `honoka.response` might be uninitialised (Recall that `honoka.response` is initialised during the execution of the first callback at line 3). On the other hand, a callback-sensitive analysis consults callback graph when it is time to propagate the state from the exit point of a callback to the entry point of another. In

[7] `https://developer.mozilla.org/en-US/docs/Web/API/Blob`

particular, when the exit node of the first callback is analysed, we propagate the current state to the second callback. Therefore, the entry point of the second function has a state which contains a precise value for `honoka.response`, that is, the object coming from the assignment at line 3.

5.5 Conclusions

We test our prototype against a set of micro- and macro benchmarks. We show that our analysis is able to handle small and medium sized JavaScript programs, and reason about the execution of their asynchronous functions. Below we summarize some of the main observations. We mark the observations as stemming from micro benchmarks, and macro benchmarks,

Overall, callback sensitivity and context sensitivity improve the precision of the analysis, as observed in both micro- *and* macro benchmarks.

Callback graph:

- **Micro:** Callback-sensitivity boosts the average accuracy of micro benchmarks by 7%. In certain micro benchmarks, callback-sensitivity enhances callback graph precision up to 25%. On the other hand, the improvement of our new context-sensitivity strategies ranges from 20,4% to 100%. Combining callback-sensitivity and context-sensitivity leads to the highest average score, i.e. 0.94; the divergence rises by 10,6%

- **Macro:** Callback sensitivity, as observed in macro benchmarks, improves the average precision of callback graph up to 3,8%. However, context-sensitivity appears to be more effective than callback-sensitivity: it increases the average accuracy by 9,3%, while in specific benchmarks it boosts precision by 26,9%. The percentage increase of the average improvement resulted by the combination of callback- and context-sensitivity reaches 11,3%.

Type errors:

- **Micro:** Callback-sensitive analyses report 26 less type errors than callback-insensitive analyses. The decrease of the total number of type errors is 61,9%. Context-sensitivity is less effective though; it reduces the number of type errors only by one.

- **Macro:** Callback-sensitivity produces fewer warnings for 5 out 6 benchmarks, leading to 5–6 less type errors in total; the percentage drop of type errors is up to 11,3%.

Analysed Callbacks:

- **Micro:** Analysis sensitivity reduces the total number of analysed callbacks by 11,2%. In certain benchmarks, though, the number of inspected callbacks is reduced by *half*, implying that an imprecise analysis might contribute to a notable increase of spurious asynchronous callbacks.

- **Macro:** A both callback- and context-sensitive analysis decreases the total number of examined callbacks by 12,5%.

5.6 Threats to Validity

Below we pinpoint the main threats to validity:

- Our analysis is an extension of an existing analysis, namely, that of TAJS, thus, the precision and performance of TAJS play an important role on the results of our work.
- Even though our analysis is designed to be sound, it models some functions of the native API unsoundly. For instance, native function `Object.freeze()`, which is used to prevent an object from being updated, is modelled unsoundly, i.e. the evaluation of `Object.freeze()` simply returns the passed object.
- We provide manual models for some built-in Node.js modules like `fs`, `http`, etc. or other APIs used in client-side applications such as `XMLHttpRequest`, `Blob`, etc. However, manual modelling might neglect some of the side-effects which stem from the interaction with those APIs, leading to unsoundness (Guarnieri and Livshits, 2009; Park et al., 2016). It is worth noting that we assumed that an HTTP request, which is issued to a server, always succeeds. Therefore, our models did not trigger any callbacks which are called in case of failure (e.g. on timeout, on network failure, etc.).
- Our macro benchmarks consist of JavaScript libraries, therefore we needed to write some test cases that invoke the API functions of those benchmarks. We provided both hand-written test cases and test cases or examples taken from their documentation, trying to test the main APIs that exercise asynchrony in JavaScript.

Chapter 6

Conclusions & Future Work

In this chapter, we first make a summary of the main achievements of our work (Section 6.1), and then we discuss the future work that arises from this report (Section 6.2).

6.1 Summary

Building upon previous works, we presented a calculus, namely, λ_q, for modelling asynchrony in JavaScript. Our calculus λ_q is flexible enough so that we can express almost every asynchronous primitive in the JavaScript language, up to the 7th edition of the ECMAScript (including promises, timers, and non-blocking I/O) in terms of λ_q.

We then presented an abstract version of λ_q which over-approximates the semantics of our calculus. By exploiting that abstract version, we designed and implemented what is, to the best of our knowledge, the first static analysis for dealing with promises; an important element of asynchronous programming. Our analysis significantly extends previous works which mostly focused on modelling the event system of client-side JavaScript programs. In the belief that there is not any implementation of a static analysis for supporting promises, a feature which have been widely embraced by the community for developing asynchronous programs (Gallaba et al., 2017), we argue that our work is a base for constructing new tools and techniques for asynchronous JavaScript.

At the same time, we introduced the concept of callback graph; a directed acyclic graph which represents the temporal relations between the execution of asynchronous callbacks. Each node in the graph denotes a callback, whereas a directed edge between two nodes indicates data flow being propagated from the source node to the target. We then designed a more precise analysis, that is, callback-sensitivity. Unlike existing static analyses which assume that asynchronous callbacks are not called in a certain order, our approach propagates state from one callback to another with regards to callback graph, respecting its execution order.

We parameterised our analysis with new context-sensitivity approaches that are specifically used for distinguishing the invocation of asynchronous callbacks. Traditional context-sensitivity policies like object-sensitivity and parameter-sensitivity are not always effective in the context of asynchronous callbacks, therefore, we leveraged the abstract model of our analysis to create context domains depending on they way that callbacks are scheduled.

```
1  if (TAJS_make("AnyBool")) {
2    Promise.resolve().then(function foo() {
3
4    }).then(function baz() {
5
6    });
7  } else {
8    Promise.resolve().then(function bar() {
9
10   }).then(function qux() {
11
12   });
13 }
```

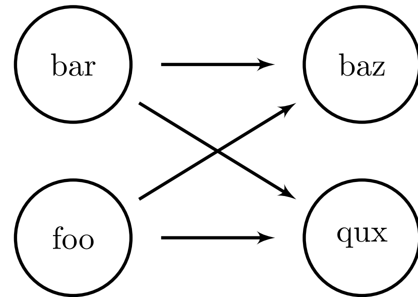


Figure 6.1: An example program and its resulting callback graph.

We evaluated the precision and performance of our analysis and the different parameterisations of it on a set of hand-written and real-world programs. The results showed that our approach can be applied on small and medium sized JavaScript programs, while analysis sensitivity (i.e. both callback- and context-sensitivity) modestly ameliorates the accuracy of the analysis results without sacrificing performance.

6.2 Future Work

We discuss a number of limitations and optimisations that emerge in the context of our analysis and describe how they can be addressed in the future (Section 6.2.1). We wrap up with a discussion on how our work can be leveraged for building new tools in the areas of bug detection and debugging (Section 6.2.2).

6.2.1 Limitations & Optimisations

We identify the main areas of our work that could benefit from further improvements.

Modelling: The λ_q calculus proposed in this report is not capable of expressing function `Promise.all()`. That function is often used by the developers to defer a computation until a number of requirements is fist met. Similarly, our model does not support `async/await` keywords which were introduced in the 8th version of ECMAScript released in 2017 as a mean for facilitating the use of promises. In particular, the `async` keyword declares a function as asynchronous which always returns a promise, while `await x` defers the execution of the *asynchronous* function in which is placed, until `x` is settled. Those language constructs are tightly coupled with promises, thus, we believe that our analysis model would be extended with a small number of modifications.

Precision: There are a few sources of imprecision that could be potentially addressed in the future via a number of optimisations. As discussed in Section 4.4.1, callback-sensitivity might lead to inaccuracy, because the event loop remains context-insensitive, and therefore, the state of the last executed callbacks is still propagated across the functions of the callback graph. One workaround for this issue is to make the event loop context-sensitive as proposed by the work (Madsen et al., 2015). In this context, the event loop is represented with

multiple program points that try to imitate the different iterations of it. The drawback of this approach is that it might introduce a significant overhead to the analysis (Madsen et al., 2015). However, it is worth seeing if such an alternative could be fruitful in the context of our work, by estimating the performance Vs. precision trade-off.

Another kind of imprecision is associated with the construction of callback graph. As a motivating example, consider an example program and its callback graph shown in Figure 6.1. Callback graph correctly states that `baz()` operates before `foo()`, and `qux()` before `bar()`, but it also contains edges from `foo()` to `qux()` and from `bar()` to `baz()`. Although it does not violate the ordering precision between those callbacks, it can potentially lead to an imprecise state at entry points of functions `baz()` and `qux()`, because their resulting state will be the join of the states coming from `foo()` and `baz()`. This means that `foo()` (`bar()`) propagates its state to `qux()` (`baz()`), even if their execution is mutual exclusive. To handle this shortcoming, our analysis could connect a callback x only with the function y that triggered its execution, excluding all the other functions which are mutual exclusive with y.

Applicability: We evaluated our analysis on a set of small and medium sized programs. Can our analysis be applied on larger benchmarks? This goal requires a lot more work beyond the context of our work, at it applies on flow-sensitive analyses in general. Specifically, we faced two major obstacles that prevented the evaluation of our prototype on larger benchmarks. First, many real-world applications might operate in a complex setting, where modelling is extremely challenging, e.g. many applications are built on top of frameworks like `react`[1] or `angular`[2], or they heavily use intricate libraries like `mongodb`[3]. Our static analysis should reason about the interactions with those libraries in scalable manner or sacrifice soundness for scalability (Madsen et al., 2013). Second, many programming patterns adopted in JavaScript degrade the analysis precision which in turn significantly hinders the performance (Andreasen and Møller, 2014). This is indicated by the fact that known flow-sensitive analyses face a number of scalability challenges on analysing large JavaScript programs (Jensen et al., 2009; Kashyap et al., 2014; Madsen et al., 2015; Park et al., 2016). Even though much progress has been made for other programming languages like C or Java (Gharat et al., 2016; Li et al., 2018), improving scalability in JavaScript seems like a tall order. In this context we need to identify the main bottlenecks of the analysis and provide workarounds and mechanisms for boosting scalability.

6.2.2 Further Research

The analysis proposed in this report constitutes one of the first steps for building new tools and techniques for asynchronous programming in JavaScript. Our work is a generic technique that can be exploited by other works in different aspects of software reliability. Below we highlight two of them:

Detecting Concurrency Bugs: Recall that in JavaScript concurrency bugs are caused due to asynchrony (Wang et al., 2017; Davis et al., 2017). Since we now have a technique for dealing with asynchronous programs, we could design a client analysis on top of it so that it statically detects data races in JavaScript programs. Such a client analysis will report a data race, if there is a read-write or write-write conflict between the program variables or

[1] `https://reactjs.org/`
[2] `https://angular.io/`
[3] `https://github.com/mongodb/node-mongodb-native`

```
1  var z;
2  var x = new Promise(function (resolve) {
3    asyncIO1(function foo(data) {
4      resolve(data);
5    })
6  });
7  x.then(function baz() {
8    z = someOperation();
9  });
10 ...
11 var y = new Promise(function (resolve) {
12   asyncIO2(function bar(data) {
13     resolve(data);
14   })
15 });
16 y.then(function baz() {
17   z = someOperation();
18 });
```

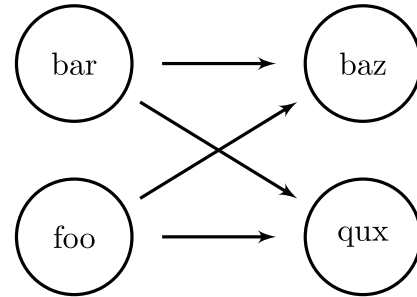


Figure 6.2: A simple program with a data race.

even resources (e.g. a file, a database, etc.) that two functions access and the execution of those functions is not deterministic. Our callback graph might be an *essential* element for such an analysis, because we can inspect it for identifying callbacks with a potential non-deterministic execution, i.e. we need to find callbacks where there is not any connection to each other in callback graph. Since the execution of a callback is non-preemptive, we will need to examine call graph to propagate the information about the fact that execution is not deterministic to the callees of an asynchronous callback. In this context, given a callback graph and a call graph, all we need to do is to formulate the exact rules for detecting data races.

As a case study, consider the simple scenario of Figure 6.2. The program creates two promises which are resolved asynchronously once the asynchronous operations `asyncIO1()` and `asyncIO2()` complete (lines 3, 12). Since the resolution of promises depends on the order in which the IO operations terminate, it is not known which promise is resolved first, meaning that the execution of their callbacks (i.e. `baz()`, and `qux()`) is not deterministic. The callback graph of the program captures this fact as there is not any path between `baz()` and `qux()`. Therefore, a race detector would take advantage of it and report a bug, because there is a write-write conflict in the body of `baz()` and `qux()`, i.e. variable `z` (lines 8, 17).

Debugging: Asynchrony in JavaScript is one of the main causes why developers introduce bugs to their programs. Many programmers do not seem to understand how flow is propagated in asynchronous applications; something that is indicated by the large number of asynchrony-related questions issued in forums like stackoverflow.com[4] (Madsen et al., 2015, 2017) or concurrency bugs reported in popular repositories (Wang et al., 2017; Davis et al., 2017). Therefore, on the basis of our work, we could develop a debugging tool that allows developers to visualise, review, and understand the data flow of their asynchronous programs. To this end, in such a scenario, we might need to extend the definition of our callback graph by enriching it with more information such as causal relations between call-

[4]`https://stackoverflow.com/`

backs. Also we would need to investigate whether such a debugging tool is indeed beneficial to developers for finding and fixing bugs. Note that there are some existing proposals for designing debugging tools for asynchrony, but as far as we know, there is not any implementation available (Madsen et al., 2017; Loring et al., 2017). Existing efforts and implementations focused on the debugging of reactive programs; a subset of asynchronous programming (Banken et al., 2018).

Bibliography

Andreasen, E. and Møller, A. (2014). Determinacy in Static Analysis for jQuery. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. pages 70

Andreasen, E. and Møller, A. (2014). Determinacy in static analysis for jQuery. pages 17–31. cited By 34. pages 13

Bae, S., Cho, H., Lim, I., and Ryu, S. (2014). SAFE¡inf¿WAPI¡/inf¿: Web API misuse detector for web applications. volume 16-21-November-2014, pages 507–517. cited By 16. pages 9, 12, 13

Balakrishnan, G. and Reps, T. (2006). Recency-abstraction for heap-allocated storage. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4134 LNCS:221–239. cited By 42. pages 10

Banken, H., Meijer, E., and Gousios, G. (2018). Debugging Data Flows in Reactive Programs. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 752–763, New York, NY, USA. ACM. pages 72

Cooper, K. D. and Torczon, L. (2012). Chapter 9 - data-flow analysis. In Cooper, K. D. and Torczon, L., editors, *Engineering a Compiler (Second Edition)*, pages 475 – 538. Morgan Kaufmann, Boston, second edition edition. pages 6

Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM. pages 13

Davis, J., Thekumparampil, A., and Lee, D. (Apr 23, 2017). Node.fz: Fuzzing the server-side event-driven architecture. EuroSys '17, pages 145–160. ACM. pages 1, 2, 5, 9, 18, 19, 20, 21, 70, 71

ECMA-262 (2015). ECMAScript 2015 Language Specification. `http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf`. [Online; accessed 03-June-2018]. pages 1, 16, 35, 54

ECMA-262 (2018). ECMAScript 2018 Language Specification. `https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`. [Online; accessed 05-June-2018]. pages 1, 4, 22, 30, 37, 55

Emanuelsson, P. and Nilsson, U. (2008). A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21. pages 6, 8

Feldthaus, A., Millstein, T., Møller, A., Schäfer, M., and Tip, F. (2011). Tool-supported refactoring for JavaScript. pages 119–137. cited By 17. pages 1, 9

Feldthaus, A. and Møller, A. (2013). Semi-automatic rename refactoring for JavaScript. pages 323–337. cited By 11. pages 1, 9

Feldthaus, A., Schafer, M., Sridharan, M., Dolby, J., and Tip, F. (2013). Efficient construction of approximate call graphs for JavaScript IDE services. pages 752–761. cited By 38. pages 1

Felleisen, M., Findler, R. B., and Flatt, M. (2009). *Semantics engineering with PLT Redex*. Mit Press. pages 24

Felleisen, M. and Friedman, D. (1987). A calculus for assignments in higher-order languages. volume Part F130236, pages 314–325. cited By 36. pages 13

Fragoso Santos, J., Gardner, P., Maksimović, P., and Naudžiūnienė, D. (2017). Towards Logic-based Verification of JavaScript Programs. In *Proceedings of 26th Conference on Automated Deduction (CADE 26)*. pages 10

Fragoso Santos, J., Maksimović, P., Naudžiūnienė, D., Wood, T., and Gardner, P. (2018). Javert: Javascript verification toolchain. *PACMPL*, 2(POPL):50:1–50:33. pages 10

Gallaba, K., Hanam, Q., Mesbah, A., and Beschastnikh, I. (2017). Refactoring asynchrony in JavaScript. pages 353–363. pages 1, 16, 17, 68

Gallaba, K., Mesbah, A., and Beschastnikh, I. (2015). Don't Call Us, We'll Call You: Characterizing Callbacks in Javascript. volume 2015-November, pages 247–256. cited By 14. pages 14, 16

Gao, Z., Bird, C., and Barr, E. (2017). To Type or Not to Type: Quantifying Detectable Bugs in JavaScript. pages 758–769. cited By 1. pages 5

Gardner, P., Maffeis, S., and Smith, G. (2012). Towards a Program Logic for JavaScript. In Field, J. and Hicks, M., editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 31–44. ACM. pages 9, 10

Gharat, P., Khedker, U., and Mycroft, A. (2016). Flow- and context-sensitive points-to analysis using generalized points-to graphs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9837 LNCS:212–236. cited By 0. pages 70

Github (2017). GitHub Octoverse 2017 — Highlights from the last twelve months. `https://octoverse.github.com/`. [Online; accessed 04-Semptember-2018]. pages 1

Guarnieri, S. and Livshits, V. B. (2009). GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *USENIX Security Symposium*, volume 10, pages 78–85. pages 1, 9, 10, 11, 12, 67

Guha, A., Saftoiu, C., and Krishnamurthi, S. (2010). The essence of javascript. In D'Hondt, T., editor, *ECOOP 2010 – Object-Oriented Programming*, pages 126–150, Berlin, Heidelberg. Springer Berlin Heidelberg. pages 9, 22, 24

Hardekopf, B., Wiedermann, B., Churchill, B., and Kashyap, V. (2014). Widening for control-flow. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8318 LNCS:472–491. cited By 2. pages 14

Hong, S., Park, Y., and Kim, M. (2014). Detecting concurrency errors in client-side java script web applications. pages 61–70. cited By 16. pages 19

Jensen, S., Jonsson, P., and Møller, A. (2012). Remedying the eval that men do. pages 34–44. cited By 44. pages 11

Jensen, S., Madsen, M., and Møller, A. (2011). Modeling the HTML DOM and browser API in static analysis of JavaScript Web Applications. pages 59–69. cited By 47. pages 2, 12, 13, 18, 35, 48

Jensen, S., Møller, A., and Thiemann, P. (2009). Type analysis for JavaScript. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5673 LNCS:238–255. cited By 85. pages 1, 2, 6, 9, 10, 13, 35, 36, 37, 38, 55, 70

Jensen, S. H., Møller, A., and Thiemann, P. (2010). Interprocedural analysis with lazy propagation. In *Proc. 17th International Static Analysis Symposium (SAS)*, volume 6337 of *LNCS*. Springer-Verlag. pages 2, 36, 37, 38

Kam, J. and Ullman, J. (1977). Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317. cited By 188. pages 6, 13, 35, 36

Kashyap, V., Dewey, K., Kuefner, E., Wagner, J., Gibbons, K., Sarracino, J., Wiedermann, B., and Hardekopf, B. (2014). JSAI: A static analysis platform for JavaScript. volume 16-21-November-2014, pages 121–132. cited By 37. pages 1, 9, 10, 11, 12, 13, 36, 48, 55, 56, 70

Ko, Y., Lee, H., Dolby, J., and Ryu, S. (2016). Practically tunable static analysis framework for large-scale JavaScript applications. pages 541–551. cited By 10. pages 55

Lee, H., Won, S., Jin, J., Cho, J., and Ryu, S. (2012). SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages*, page 96. Citeseer. pages 1, 5, 9, 13

Lhoták, O. and Hendren, L. (2008). Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Transactions on Software Engineering and Methodology*, 18(1). cited By 57. pages 8

Li, Y., Tan, T., Møler, A., and Smaragdakis, Y. (2018). Scalability-First Pointer Analysis with Self-Tuning Context-Sensitivity. In *Proc. 12th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. pages 70

Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J., Chang, B.-Y., Guyer, S., Khedker, U., Møler, A., and Vardoulakis, D. (2015). In defense of soundiness: A manifesto. *Communications of the ACM*, 58(2):44–46. cited By 45. pages 11

Loring, M. C., Marron, M., and Leijen, D. (Oct 24, 2017). Semantics of asynchronous JavaScript. DLS 2017, pages 51–62. ACM. pages 2, 10, 18, 19, 20, 21, 22, 24, 25, 28, 29, 30, 32, 33, 72

Madsen, M., Lhoták, O., and Tip, F. (2017). A model for reasoning about JavaScript promises. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–24. pages 2, 10, 16, 17, 21, 22, 24, 25, 33, 71, 72

Madsen, M., Livshits, B., and Fanning, M. (2013). Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. pages 12, 70

Madsen, M., Tip, F., and Lhoták, O. (2015). Static analysis of event-driven Node.js JavaScript applications. *ACM SIGPLAN Notices*, 50(10):505–519. pages 2, 5, 8, 9, 10, 12, 19, 24, 40, 69, 70, 71

Maffeis, S., Mitchell, J., and Taly, A. (2008). An operational semantics for JavaScript. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5356 LNCS:307–325. cited By 48. pages 9

Maffeis, S. and Taly, A. (2009). Language-based isolation of untrusted JavaScript. pages 77–91. cited By 47. pages 1, 10

MDN (2018). EventTarget - Web APIs. `https://developer.mozilla.org/en-US/docs/Web/API/EventTarget`. [Online; accessed 03-June-2018]. pages 15

Milanova, A., Rountev, A., and Ryder, B. (2002). Parameterized object sensitivity for points-to and side-effect analyses for Java. pages 1–11. cited By 104. pages 8

Mutlu, E., Tasiran, S., and Livshits, B. (2015). Detecting JavaScript races that matter. pages 381–392. cited By 6. pages 1, 20, 21

Node.js (2018a). Don't Block the Event Loop (or the Worker Pool). `https://nodejs.org/en/docs/guides/dont-block-the-event-loop/`. [Online; accessed 04-June-2018]. pages 18

Node.js (2018b). The Node.js Event Loop, Timers, and process.nextTick(). `https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/`. [Online; accessed 04-June-2018]. pages 18, 19, 28, 29

Ocariza Jr., F., Bajaj, K., Pattabiraman, K., and Mesbah, A. (2013). An empirical study of client-side JavaScript bugs. pages 55–64. cited By 38. pages 5

Ocariza Jr., F., Pattabiraman, K., and Zorn, B. (2011). JavaScript errors in the wild: An empirical study. pages 100–109. cited By 34. pages 5

Park, C., Lee, H., and Ryu, S. (2013). All about the with statement in javascript: Removing with statements in JavaScript applications. pages 73–84. cited By 4. pages 11

Park, C., Won, S., Jin, J., and Ryu, S. (2016). Static analysis of JavaScript web applications in the wild via practical DOM modeling. pages 552–562. cited By 12. pages 2, 8, 12, 13, 35, 48, 61, 67, 70

Petrov, B., Vechev, M., Sridharan, M., and Dolby, J. (2012). Race detection for web applications. pages 251–261. cited By 24. pages 1, 21

Reynolds, J. (2002). Separation logic: A logic for shared mutable data structures. pages 55–74. cited By 1083. pages 9

Richards, G., Hammer, C., Burg, B., and Vitek, J. (2011). The Eval That Men Do A Large-Scale Study of the Use of Eval in JavaScript Applications. In *ECOOP 2011 - OBJECT-ORIENTED PROGRAMMING*, volume 6813 of *Lecture Notes in Computer Science*, pages 52–78. 25th European Conference on Object-Oriented Programming (ECOOP), Lancaster, ENGLAND, JUL 25-29, 2011. pages 4, 11

Richards, G., Lebresne, S., Burg, B., and Vitek, J. (2010). An Analysis of the Dynamic Behavior of JavaScript Programs. *ACM SIGPLAN NOTICES*, 45(6):1–12. ACM SIGPLAN Conference on Programming Language Design and Implementation, Toronto, CANADA, JUN 05-10, 2010. pages 1

Smaragdakis, Y., Bravenboer, M., and Lhoták, O. (2011). Pick your contexts well: Understanding object-sensitivity: The making of a precise and scalable pointer analysis. *ACM SIGPLAN Notices*, 46(1):17–29. cited By 53. pages 8

Staicu, C.-A., Pradel, M., and Livshits, B. (2018). Synode: Understanding and Automatically Preventing Injection Attacks on Node.js. In *Network and Distributed System Security (NDSS)*. pages 1

Sun, K. and Ryu, S. (2017). Analysis of JavaScript programs: Challenges and research trends. *ACM Computing Surveys*, 50(4). cited By 0. pages 1, 9

v10.3.0 Documentation, N. (2018). Events. `https://nodejs.org/api/events.html`. [Online; accessed 03-June-2018]. pages 15

Wang, J. (2017). Characterizing and taming non-deterministic bugs in Javascript applications. In *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*, pages 1006–1009. IEEE. pages 21

Wang, J., Dou, W., Gao, Y., Gao, C., Qin, F., Yin, K., and Wei, J. (2017). A comprehensive study on real world concurrency bugs in Node.js. pages 520–531. cited By 2. pages 2, 5, 14, 18, 19, 20, 21, 70, 71

Wei, S. and Ryder, B. (2014). State-sensitive points-to analysis for the dynamic behavior of JavaScript objects. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8586 LNCS:1–26. cited By 10. pages 9, 10

Zheng, Y., Bao, T., and Zhang, X. (Mar 28, 2011). Statically locating web application bugs caused by asynchronous calls. WWW ’11, pages 805–814. ACM. pages 5, 19, 20

Appendices

Appendix A

Semantics of $\lambda_{\mathtt{q}}$

$$\frac{\begin{array}{c} v \neq \bot \quad (\texttt{pending},\ t,\ k, l) = \pi(p) \quad v \notin dom(\pi) \quad k' = \langle(\alpha,\ f,\ [v],\ r) \mid (\alpha,\ f,\ a,\ r) \in t\rangle \\ \kappa' = \kappa :: k' \quad \chi = (\texttt{rejected}, v) \quad \pi' = \pi[p \mapsto (\chi, [], [], l)] \quad p \neq l_{time} \wedge p \neq l_{io} \end{array}}{\pi,\ \phi,\ \kappa,\ \tau,\ E[p.\texttt{reject}(v)] \rightarrow \pi',\ \phi,\ \kappa',\ \tau,\ E[l.\texttt{reject}(v)]} \text{[reject-pending]}$$

$$\frac{\begin{array}{c} (\texttt{pending},\ t,\ k, l) = \pi(p) \quad v \in dom(\pi) \\ p(v) = (\texttt{pending},\ t',\ k', \bot) \quad \pi' = \pi[v \mapsto (\texttt{pending},\ t',\ k',\ p)] \end{array}}{\pi,\ \phi,\ \kappa,\ \tau,\ E[p.\texttt{reject}(v)] \rightarrow \pi',\ \phi,\ \kappa',\ \tau,\ E[\texttt{undef}]} \text{[reject-pend-pend]}$$

$$\frac{(\texttt{pending},\ t,\ k, l) = \pi(p) \quad v \in dom(\pi) \quad p(v) = ((\texttt{rejected},\ v'),\ t',\ k', m)}{\pi,\ \phi,\ \kappa,\ \tau,\ E[p.\texttt{reject}(v)] \rightarrow \pi,\ \phi,\ \kappa,\ \tau,\ E[p.\texttt{reject}(\texttt{v}')]} \text{[reject-pend-ful]}$$

$$\frac{\begin{array}{c} v = \bot \quad (\texttt{pending},\ t,\ k, l) = \pi(p) \quad \kappa' = \kappa :: k \\ \chi = (\texttt{rejected}, v) \quad \pi' = \pi[p \mapsto (\chi, [], [], l)] \end{array}}{\pi,\ \phi,\ \kappa,\ \tau,\ E[p.\texttt{reject}(v)] \rightarrow \pi',\ \phi,\ \kappa',\ \tau,\ E[l.\texttt{reject}(v)]} \text{[reject-pending-}\bot\text{]}$$

$$\frac{\pi(p) \downarrow_1 \neq \texttt{pending}}{\pi,\ \phi, \kappa,\ \tau,\ E[p.\texttt{reject}(v)] \rightarrow \pi,\ \phi,\ \kappa,\ \tau,\ E[\texttt{undef}]} \text{[reject-settled]}$$

$$\frac{\begin{array}{c} (\texttt{pending}\ , t\ , k, l) = \pi(p) \quad k' = k \cdot (p',\ f,\ [n_1, n_2, \ldots, n_n],\ r) \\ \pi' = \pi[p \mapsto (\texttt{pending},\ t,\ k')] \end{array}}{\pi,\ \phi,\ \kappa,\ \tau,\ E[p.\texttt{registerRej}(f, p', r, n_1, n_2, \ldots, n_n)] \rightarrow \pi,'\ \phi,\ \kappa,\ \tau,\ E[\texttt{undef}]} \text{[registerRej-pending]}$$

$$\frac{\begin{array}{c} p \neq l_{time} \wedge p \neq l_{io} \quad (s,\ t\ , k\ , \chi) = \pi(p) \quad s \downarrow_1 = \texttt{rejected} \\ s \downarrow_2 \neq \bot \quad \kappa' = \kappa \cdot (p',\ f,\ [s \downarrow_2],\ r) \end{array}}{\pi,\ \phi,\ \kappa,\ \tau,\ E[p.\texttt{registerRej}(f, p', r, n_1, n_2, \ldots, n_n)] \rightarrow \pi,'\ \phi,\ \kappa,\ \tau,\ E[\texttt{undef}]} \text{[registerRej-rejected]}$$

$$\frac{\begin{array}{c} p \neq l_{time} \wedge p \neq l_{io} \quad (s,\ t\ , k\ , \chi) = \pi(p) \quad s \downarrow_1 = \texttt{rejected} \\ s \downarrow_2 = \bot \quad \kappa' = \kappa \cdot (p',\ f,\ [n_1, n_2, \ldots, n_n],\ r) \end{array}}{\pi,\ \phi,\ \kappa,\ \tau,\ E[p.\texttt{registerRej}(f, p', r, n_1, n_2, \ldots, n_n)] \rightarrow \pi,'\ \phi,\ \kappa,\ \tau,\ E[\texttt{undef}]} \text{[registerRej-rejected-}\bot\text{]}$$

$$\frac{\begin{array}{c} p = l_{time} \vee p = l_{io} \quad (s,\ t\ , k\ , \chi) = \pi(p) \quad s \downarrow_1 = \texttt{rejected} \\ s \downarrow_2 = \bot \quad \tau' = \tau \cdot (p',\ f,\ [n_1, n_2, \ldots, n_n],\ r) \end{array}}{\pi,\ \phi,\ \kappa,\ \tau,\ E[p.\texttt{registerRej}(f, p', r, n_1, n_2, \ldots, n_n)] \rightarrow \pi,'\ \phi,\ \kappa,\ \tau,\ E[\texttt{undef}]} \text{[registerRej-timer-io-}\bot\text{]}$$

$$\frac{(s,\ t\ , k\ , \chi) = \pi(p) \quad s \downarrow_1 = \texttt{rejected}}{\pi,\ \phi,\ \kappa,\ \tau,\ E[p.\texttt{registerRes}(f, p', r, n_1, n_2, \ldots, n_n)] \rightarrow \pi,\ \phi,\ \kappa,\ \tau,\ E[\texttt{undef}]} \text{[registerRes-rejected]}$$

$$\frac{(s,\ t\ , k\ , \chi) = \pi(p) \quad s \downarrow_1 = \texttt{fulfilled}}{\pi,\ \phi,\ \kappa,\ \tau,\ E[p.\texttt{registerRej}(f, p', r, n_1, n_2, \ldots, n_n)] \rightarrow \pi,\ \phi,\ \kappa,\ \tau,\ E[\texttt{undef}]} \text{[registerRej-fulfilled]}$$

Figure A.1: The semantics of λ_q

Appendix B

Performance of Micro Benchmarks

	Average Time								Median							
Benchmark	**NC-No**	**NC-R**	**NC-Q**	**NC-QR**	**C-No**	**C-R**	**C-Q**	**C-QR**	**NC-No**	**NC-R**	**NC-Q**	**NC-QR**	**C-No**	**C-R**	**C-Q**	**C-QR**
micro01	0.49	0.5	0.52	0.48	0.38	0.35	0.35	0.34	0.44	0.43	0.45	0.45	0.33	0.33	0.33	0.33
micro02	0.4	0.41	0.4	0.4	0.37	0.37	0.38	0.37	0.41	0.4	0.4	0.4	0.37	0.37	0.38	0.38
micro03	0.37	0.36	0.34	0.35	0.3	0.3	0.32	0.32	0.34	0.34	0.34	0.35	0.29	0.29	0.3	0.3
micro04	0.34	0.36	0.34	0.36	0.36	0.35	0.32	0.31	0.32	0.32	0.31	0.31	0.33	0.33	0.31	0.31
micro05	0.79	0.83	0.86	0.92	0.7	0.67	0.69	0.64	0.81	0.75	0.89	0.96	0.69	0.67	0.65	0.62
micro06	1.26	1.23	1.24	1.23	1.04	1.04	1.05	1.04	1.26	1.23	1.23	1.22	1.03	1.03	1.04	1.02
micro07	1.25	1.21	1.25	1.23	0.86	0.87	0.88	0.86	1.23	1.23	1.26	1.2	0.86	0.86	0.86	0.85
micro08	0.71	0.75	0.71	0.71	0.73	0.73	0.78	0.75	0.72	0.76	0.7	0.69	0.73	0.73	0.76	0.75
micro09	0.67	0.68	0.66	0.68	0.59	0.59	0.61	0.59	0.68	0.68	0.65	0.66	0.58	0.59	0.59	0.58
micro10	0.53	0.53	0.52	0.51	0.43	0.42	0.42	0.45	0.56	0.57	0.56	0.57	0.47	0.46	0.46	0.5
micro11	0.91	0.92	0.9	0.92	0.83	0.79	0.77	0.79	0.91	0.93	0.9	0.92	0.81	0.79	0.75	0.81
micro12	0.91	0.9	0.9	0.92	0.71	0.7	0.7	0.71	0.91	0.89	0.9	0.93	0.71	0.7	0.7	0.72
micro13	0.64	0.63	0.61	0.62	0.49	0.5	0.51	0.51	0.63	0.62	0.62	0.62	0.49	0.5	0.51	0.51
micro14	0.9	0.92	0.91	0.91	0.73	0.72	0.77	0.71	0.9	0.92	0.9	0.9	0.72	0.72	0.75	0.7
micro15	0.4	0.39	0.39	0.39	0.38	0.36	0.36	0.37	0.4	0.39	0.39	0.39	0.38	0.36	0.36	0.37
micro16	1.19	1.16	1.18	1.14	0.9	0.91	0.94	0.93	1.19	1.15	1.17	1.15	0.9	0.93	0.94	0.94
micro17	2.04	2.18	1.95	2.11	2.11	2.27	2.08	2.11	2.14	2.24	1.96	2.13	2.05	2.44	2.16	2.1
micro18	0.39	0.34	0.31	0.31	0.37	0.31	0.32	0.33	0.36	0.31	0.3	0.3	0.36	0.31	0.3	0.3
micro19	1.54	1.14	1.1	1.15	1.81	1.04	1.03	1.04	1.52	1.14	1.1	1.14	1.79	1.05	1.05	1.06
micro20	0.77	0.72	0.75	0.7	0.72	0.71	0.68	0.68	0.77	0.72	0.75	0.73	0.74	0.71	0.7	0.68
micro21	0.68	0.69	0.61	0.66	0.59	0.61	0.61	0.58	0.66	0.73	0.56	0.71	0.6	0.63	0.65	0.63
micro22	0.89	0.97	1.02	0.92	1.0	0.95	0.97	0.98	0.93	0.99	1.01	0.91	0.99	0.91	0.99	0.98
micro23	0.68	0.65	0.65	0.68	0.54	0.54	0.53	0.54	0.78	0.52	0.48	0.78	0.61	0.62	0.6	0.61
micro24	0.59	0.58	0.57	0.62	0.45	0.42	0.45	0.44	0.63	0.59	0.63	0.61	0.46	0.37	0.47	0.46
micro25	0.76	0.74	0.71	0.8	0.85	0.88	0.87	0.88	0.65	0.66	0.63	0.72	0.74	0.78	0.72	0.82
micro26	1.41	1.38	1.4	1.42	0.96	0.95	0.89	0.94	1.45	1.4	1.39	1.46	0.96	0.96	0.94	0.97
micro27	0.51	0.47	0.46	0.5	0.48	0.43	0.43	0.46	0.55	0.47	0.41	0.5	0.5	0.48	0.43	0.49
micro28	0.73	0.72	0.75	0.76	0.9	0.87	0.89	0.91	0.8	0.77	0.8	0.79	0.99	0.9	0.93	0.95
micro29	0.65	0.61	0.66	0.65	0.63	0.61	0.61	0.65	0.65	0.64	0.67	0.65	0.64	0.62	0.62	0.66

Table B.1: Times of different analyses in seconds

Appendix C

Ethics Checklist

Table C.1 shows the complete ethics checklist.

Table C.1: Complete ethics checklist

	Yes	No
Section 1: HUMAN EMBRYOS/FOETUSES		
Does your project involve Human Embryonic Stem Cells?		X
Does your project involve the use of human embryos?		X
Does your project involve the use of human foetal tissues / cells?		X
Section 2: HUMANS		
Does your project involve human participants?		X
Section 3: HUMAN CELLS / TISSUES		
Does your project involve human cells or tissues? (Other than from “Human Embryos/Foetuses” i.e. Section 1)?		X
Section 4: PROTECTION OF PERSONAL DATA		
Does your project involve personal data collection and/or processing?		X
Does it involve the collection and/or processing of sensitive personal data (e.g. health, sexual lifestyle, ethnicity, political opinion, religious or philosophical conviction)?		X
Does it involve processing of genetic information?		X
Does it involve tracking or observation of participants? It should be noted that this issue is not limited to surveillance or localization data. It also applies to Wan data such as IP address, MACs, cookies etc.		X
Does your project involve further processing of previously collected personal data (secondary use)? For example Does your project involve merging existing data sets?		X
Section 5: ANIMALS		
Does your project involve animals?		X
Section 6: DEVELOPING COUNTRIES		
Does your project involve developing countries?		X
If your project involves low and/or lower-middle income countries, are any benefit-sharing actions planned?		X
Could the situation in the country put the individuals taking part in the project at risk?		X
Section 7: ENVIRONMENTAL PROTECTION AND SAFETY		
Does your project involve the use of elements that may cause harm to the environment, animals or plants?		X
Does your project deal with endangered fauna and/or flora /protected areas?		X
Does your project involve the use of elements that may cause harm to humans, including project staff?		X
Does your project involve other harmful materials or equipment, e.g. high-powered laser systems?		X
Section 8: DUAL USE		
Does your project have the potential for military applications?	X	
Does your project have an exclusive civilian application focus?		X
Will your project use or produce goods or information that will require export licenses in accordance with legislation on dual use items?		X
Does your project affect current standards in military ethics – e.g., global ban on weapons of mass destruction, issues of proportionality, discrimination of combatants and accountability in drone and autonomous robotics developments, incendiary or laser weapons?		X
Section 9: MISUSE		
Does your project have the potential for malevolent/criminal/terrorist abuse?		X
Does your project involve information on/or the use of biological-, chemical-, nuclear/radiological-security sensitive materials and explosives, and means of their delivery?		X
Does your project involve the development of technologies or the creation of information that could have severe negative impacts on human rights standards (e.g. privacy, stigmatization, discrimination), if misapplied?		X
Does your project have the potential for terrorist or criminal abuse e.g. infrastructural vulnerability studies, cybersecurity related project?		X
SECTION 10: LEGAL ISSUES		
Will your project use or produce software for which there are copyright licensing implications?		X
Will your project use or produce goods or information for which there are data protection, or other legal implications?		X
SECTION 11: OTHER ETHICS ISSUES		
Are there any other ethics issues that should be taken into consideration?		X

Appendix D

Ethical and Professional Considerations

Our project is related to a program analysis technique which is used during the development and maintance of JavaScript applications. Therefore, it does not involve any ethical considerations for the most of the categories. The justification is self-explanatory.

Our work is not related to human embryos, foetuses, human cells, tissues, or animals. Our work neither involves human participants nor collects or processes any personal data. Similarly, our project is not relevant with developing countries. Our work does not have any negative impacts on the environment or other individuals. Beyond that, the software we use for our work comes with an Apache Licence 2.0, a permissive licence that allows us to make our modifications for private use; therefore, they are not any license implications. However, the analysis we implemented can be potentially used during the development and maintanance of military applications, if they are written in JavaScript.