Static Analysis for Asynchronous JavaScript Programs

Thodoris Sotiropoulos[1]
Athens University of Economics and Business, Greece

Benjamin Livshits
Imperial College London, UK
Brave Software, London, UK

Abstract

Asynchrony has become an inherent element of JavaScript, as an effort to improve 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 not always well-understood by the developers who introduce many asynchrony-related bugs to their programs. Worse, there are few tools and techniques available for analyzing 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 the *callback graph*, a representation for capturing data flow between asynchronous code. We exploit the callback graph for designing a more precise analysis that respects the execution order between different asynchronous functions. We parameterize our analysis with one novel context-sensitivity flavor, and we end up with multiple analysis variations for building callback graph.

We performed a number of experiments on a set of hand-written and real-world JavaScript programs. Our results show that our analysis can be applied to medium-sized programs achieving 79% precision, on average. The findings further suggest that analysis sensitivity is beneficial for the vast majority of the benchmarks. Specifically, it is able to improve precision by up to 28.5%, while it achieves an 88% precision on average without highly sacrificing performance.

2012 ACM Subject Classification Theory of computation → Program analysis; Software and its engineering → Semantics

Keywords and phrases static analysis, asynchrony, JavaScript

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.8

Related Version A full version of the paper is available at `https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/1718-pg-projects/SotiropoulosT-Static-Analysis-for-Asynchronous-JavaScript-Programs.pdf`.

Funding The work described here was partially supported by the CROSSMINER Project, which has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 732223.

Acknowledgements We would like to thank the anonymous ECOOP reviewers, as well as Max Schlueter and Dimitris Mitropoulos for their insightful comments and feedback.

[1] The work of this author was mostly done while at Imperial College London.

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 8; pp. 8:1–8:29
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

JavaScript is an integral part of web development. Since its initial release in 1995, it has evolved from a simple scripting language – 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 a dramatic increase in 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 from 2014 to 2018 [14].

Although the dominance of JavaScript is impressive, the community has widely criticized it because it poses many concerns as to the security or correctness of the programs [36]. 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, and more. Many developers often do not understand or do not properly use these features. As a result, they introduce 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 analyzers [19, 23, 21], IDE and refactoring tools [6, 7, 8, 11], or bug and vulnerability detectors [28, 15, 34, 31, 5, 37] – to assist developers with the development and maintenance of their applications. Program analysis, and especially static analysis, plays a crucial role in the design of such tools [38].

Additionally, preserving the scalability of modern web applications has become more critical than ever. As an effort to improve the throughput of web programs, JavaScript has started to adopt an event-driven programming paradigm [4]. In this context, code is executed *asynchronously* in response to certain events, e.g., user input, a response from a server, data read from disk, etc. In the first years of JavaScript, someone could come across that asynchrony mainly in a browser environment e.g., DOM events, AJAX calls, timers, etc. However, in recent years, asynchrony has become a salient and intrinsic element of the language, as newer versions of the language's core specification (i.e., ECMAScript) have introduced more and more asynchronous features. For example, ECMAScript 6 introduces promises; an essential element of asynchronous programming that allows developers to track the state of an asynchronous computation easily. Specifically, the state of a promise object can be: (1) *fulfilled* when the associated operation is complete, and the promise object tracks its resulting value, (2) *rejected* when the associated operation has failed, and (3) *pending* when the associated operation has been neither completed nor failed.

Promises are particularly useful for asynchronous programming because they provide an intuitive way for creating chains of asynchronous computation that facilitate the enforcement of program's execution order as well as error propagation [12, 11]. Depending on their state, promises trigger the execution of certain functions (i.e., *callbacks*) asynchronously. To do so, the API of promises provides the method `x.then(f1, f2)` for registering new callbacks (i.e., `f1` and `f2`) on a promise object `x`. For example, we call the callback `f1` when the promise is fulfilled, while we trigger the callback `f2` once the promise is rejected. The method `x.then()` returns a new promise which the return value of the provided callbacks (i.e., `f1, f2`) fulfills. Since their emergence, JavaScript developers have widely embraced promises. For example, a study in 2015 showed that 75% of JavaScript frameworks use promises [12].

```
 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 1 Real-world example that mixes promises with asynchronous I/O.

Building upon promises, newer versions of ECMAScript have added more language features related to asynchrony. Specifically, in ECMAScript 8, we have the `async/await` keywords. The `async` keyword declares a function as asynchronous that returns a promise fulfilled with its return value, while `await x` defers the execution of an asynchronous function until the promise object `x` is settled (i.e., it is either fulfilled or rejected). The latest edition of ECMAScript (ECMAScript 9) adds asynchronous iterators and generators that allow developers to iterate over asynchronous data sources.

Beyond promises, many JavaScript applications are written to perform non-blocking I/O operations. Unlike traditional statements, when we perform a non-blocking I/O operation, the execution is not interrupted until I/O terminates. By contrast, the I/O operation is done asynchronously, which means that the execution proceeds to the next tasks while I/O takes place. Programmers often mix asynchronous I/O with promises. For instance, consider the real-world example of Figure 1. At line 1, the code performs an asynchronous request and returns a promise object that is fulfilled asynchronously once the request succeeds. Then, this promise object can be used for processing the response of the server asynchronously. For instance, at lines 2–23, we create a promise chain. The first callback of this chain (lines 2–14) clones the response of the request, and assigns it to the property `response` of the object `honoka` (line 3). Then, it parses the body of the response, and finally, the return value of the callback fulfills the promise object allocated by the first invocation of `then()` (lines 5–13). The second callback (lines 15–23) retrieves the headers of the response – which the statement at line 3 assigns to `honoka.response` – and if the content type is "application/json", it converts the data of the response into a JSON object (lines 17–19).

Like the other characteristics of JavaScript, programmers do not always clearly understand asynchrony, as a large number of asynchrony-related questions issued in popular sites like `stackoverflow.com`[2] [27, 26], or the number of bugs reported in open-source repositories [39, 5] indicate. However, existing tools and techniques have limited (and in many cases no)

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

support for asynchronous programs. Designing static analysis for asynchrony involves several challenges not addressed by previous work. In particular, existing tools mainly focus on the event system of client-side JavaScript applications [18, 33], and they lack the support of the more recent features added to the language, such as promises. Also, many previous tools conservatively consider that all the asynchronous callbacks processed by the *event loop* – the program point that continuously waits for new events to come and is responsible for the scheduling and execution of callbacks – can be called in any order [18, 33, 21]. However, such an approach may lead to imprecision and false positives. Back to the example of Figure 1, it is easy to see that an analysis that does not respect the execution order between the first and the second callback will report a type error at line 17 (access of `honoka.response.headers.get (" Content - Type "))`. Specifically, an imprecise analysis assumes that the callback defined at lines 15–23 might be executed first; therefore, `honoka.response`, assigned at line 3, might be uninitialized.

In this work, we tackle the issues above, by designing and implementing a static analysis that deals with asynchronous JavaScript programs. To do so, we first propose a model for understanding and expressing a wide range of asynchronous constructs found in JavaScript. Based on this model, we design our static analysis. We propose a new representation, which we call callback graph, that provides information about the execution order of the asynchronous code. The callback graph proposed in this work tries to shed light on how data flow between asynchronous code is propagated. Contrary to previous works, we leverage the callback graph and devise a more precise analysis that respects the execution order of asynchronous functions. Furthermore, we parameterize our analysis with one novel context-sensitivity option designed for asynchronous code. Specifically, we distinguish data flow between asynchronous callbacks based on the promise object that they belong to, or the next computation that the execution proceeds to.

Contributions. Our work makes the following four contributions:

- We propose a calculus, i.e., $\lambda_{\mathtt{q}}$, for modeling asynchrony. Our calculus unifies different asynchronous idioms into a single model. Thus, we can express promises, timers, asynchronous I/O, and other asynchronous features found in the language (§2).
- We design and implement a static analysis that is capable of handling asynchronous JavaScript programs by exploiting the abstract version of $\lambda_{\mathtt{q}}$. To the best of our knowledge, our analysis is the first to deal with JavaScript promises (§3.1).
- We propose the callback graph, a representation that illustrates the execution order between asynchronous callbacks. Building on that, we propose a more precise analysis, (i.e., callback-sensitive analysis) that internally consults the callback graph to retrieve information about the temporal relations of asynchronous functions so that it propagates data flow accordingly. Besides that, we parameterize our analysis with a novel context-sensitivity option (i.e., QR-sensitivity) used for distinguishing asynchronous callbacks. (§3.2, §3.3).
- We evaluate the performance and the precision of our analysis on a set of micro benchmarks and a set of real-world JavaScript modules. For the impatient reader, we find that our prototype is able to analyze medium-sized asynchronous programs, and the analysis sensitivity is beneficial for improving the analysis precision. The results showed that our analysis is able to achieve a 79% precision for the callback graph, on average. The analysis sensitivity (i.e. callback- and QR-sensitivity) can further improve callback graph precision by up to 28.5% and reduce the total number of type errors by 13.9% as observed in the real-world benchmarks (§4).

$$
\begin{array}{l}
v \in \mathit{Val} ::= \ \ldots \mid \bot \\[4pt]
e \in \mathit{Exp} ::= \ \ldots \\
\quad \mid \mathsf{newQ}() \mid e.\mathsf{fulfill}(e) \mid e.\mathsf{reject}(e) \mid e.\mathsf{registerFul}(e, e, \ldots) \mid e.\mathsf{registerRej}(e, e, \ldots) \\
\quad \mid \mathsf{append}(e) \mid \mathsf{pop}() \mid \bullet \\[4pt]
E ::= \ \ldots \\
\quad \mid E.\mathsf{fullfill}(e) \mid v.\mathsf{fulfill}(E) \mid E.\mathsf{reject}(e) \mid v.\mathsf{reject}(E) \\
\quad \mid E.\mathsf{registerFul}(e, e, \ldots) \mid v.\mathsf{registerFul}(v, \ldots, E, e, \ldots) \\
\quad \mid E.\mathsf{registerRej}(e, e, \ldots) \mid v.\mathsf{registerRej}(v, \ldots, E, e, \ldots) \\
\quad \mid \mathsf{append}(E)
\end{array}
$$

Figure 2 The syntax of $\lambda_{\mathtt{q}}$.

2 Modeling Asynchrony

As a starting point, we need to define a model to express asynchrony. The goal of this model is to provide us with the foundations for gaining a better understanding of the asynchronous primitives and ease the design of a static analysis for asynchronous JavaScript programs. This model is expressed through a calculus called $\lambda_{\mathtt{q}}$; an extension of $\lambda_{\mathtt{js}}$ which is the core calculus for JavaScript developed by Guha et. al. [16]. Our calculus is inspired by previous work [26, 25], however, it is designed to be flexible so that we can express promises, timers, asynchronous I/O, and other sources of asynchrony found in the language up to ECMAScript 7, such as thenables. Also unlike previous work – as we will see later on – our calculus enables us to model the effects of the exceptions trapped by the event loop. Additionally, we are able to handle callbacks that are invoked with arguments passed during callback registration; something that is not supported by the previous models. However, note that λ_q does not handle the `async/await` keywords and the asynchronous iterators/generators introduced in the recent editions of the language.

2.1 The $\lambda_{\mathtt{q}}$ calculus

The key component of our model is *queue objects*. Queue objects are 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 trigger the execution of callbacks depending on its state. Initially, a queue object is pending. A pending queue object can transition to a fulfilled or a rejected queue object. A queue object is fulfilled or rejected with a value. This value is later passed as an argument of the corresponding 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*.

2.1.1 Syntax and Domains

Figure 2 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:

- $\mathsf{newQ}()$: This expression creates a fresh queue object in a pending state. It has no callbacks associated with it.
- $e_1.\mathsf{fulfill}(e_2)$: This expression fulfills the receiver (i.e., the expression e_1) with the value of e_2.

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

Figure 3 The concrete domains of λ_{q}.

- $e_1.\mathsf{reject}(e_2)$: This expression rejects the receiver (i.e., the expression e_1) with the value of e_2.
- $e_1.\mathsf{registerFul}(e_2, e_3, \dots)$: This expression registers the callback e_2 to the receiver. This callback is executed *only* when the receiver is fulfilled. This expression also expects another queue object passed as the second argument, i.e., e_3. This queue object will be fulfilled with the return value of the callback e_2. This allows us to model chains of promises where a promise resolves with the return value of another promise's callback. This expression might receive optional parameters (i.e., expressed through "…") with which e_2 is called when the queue object is fulfilled with $\perp$ value. We will justify the intuition behind that later on.
- $e_1.\mathsf{registerRej}(e_2, e_3, \dots)$: The same as $e.\mathsf{registerFul}(\dots)$, but this time the given callback is executed once the receiver is rejected.
- $\mathsf{append}(e)$: This expression appends the queue object e to the top of the current queue chain. As we will see later, the top element of a queue chain corresponds to the queue object that is needed to be rejected when the execution encounters an uncaught exception.
- $\mathsf{pop}()$: 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 [9, 16, 27, 25, 26] to express how the evaluation of an expression proceeds. The symbol E denotes which sub-expression is currently being evaluated. For instance, $E.\mathsf{fulfill}(e)$ describes that we evaluate the receiver of fulfill, whereas $v.\mathsf{fulfill}(E)$ 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 λ_{q} calculus introduces a new value, that is, $\perp$. This value differs from `null` and `undefined` because it expresses the absence of value. Thus, it does not have correspondence with any JavaScript value.

Figure 3 presents the semantic domains of λ_{q}. In particular, a queue is a partial map of addresses to queue objects. The symbol l_i – where i is a positive integer – indicates an address. Notice that the set of the addresses also includes two special reserved addresses, i.e., l_{time}, l_{io}. We use these two addresses to store the queue objects responsible for keeping the state of callbacks related to timers and asynchronous I/O respectively (Section 2.1.4 explains how we model those JavaScript features). A queue object is described by its state – recall that a queue object is either pending or fulfilled and rejected with a value – a sequence of callbacks executed on fulfillment, and a sequence of callbacks called on rejection. The last element of a queue object is a sequence of addresses. These addresses correspond to the queue objects that are dependent on the current. A queue object q_1 depends on q_2, when q_1 is settled whenever q_2 is settled. This means that when the queue object q_2 is fulfilled (rejected), q_1 is also fulfilled (rejected) with the same value as q_2. 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 an argument.

Moving to the domains of callbacks, we see that a callback consists of an address, a function, and a list of values (i.e., arguments of the function). Note that the first component denotes the address of the queue object that is fulfilled with the return value of the function. In the list of callbacks $\kappa \in \mathit{ScheduledCallbacks}$, we keep the order in which callbacks are scheduled. Note that we maintain one more list of callbacks (i.e., $\tau \in \mathit{ScheduledTimerIO}$) where we store the callbacks registered on the queue objects located at the addresses l_{time}, l_{io}. We defer the discussion about why we keep two separate lists until Section 2.1.3.

A queue chain $\phi \in \mathit{QueueChain}$ is a sequence of addresses. In a queue chain, we store the queue object that we reject, when 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 usual.

2.1.2 Semantics

Equipped with the appropriate definitions of the syntax and domains, in Figure 4, we present the small-step semantics of λ_{q} which is an adaptation of previous calculi [25, 26]. Note that we demonstrate the most representative rules of our semantics; we omit some rules for brevity. For what follows, the binary operation denoted by the symbol $\cdot$ means the addition of an element to a list, the operation indicated by :: stands for list concatenation, while $\downarrow_i$ means the projection of the i^{th} element.

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 κ and τ, and an expression e in the evaluation context E lead to a new queue π', a new queue chain ϕ', two new sequences of callbacks κ' and τ', and a new expression e' in the same evaluation context E, assuming that the expression e is reduced to e' (i.e., $e \hookrightarrow e'$). The [E-CONTEXT] rule describes this behavior.

The [NEWQ] rule creates a new queue object and adds it to the queue using a fresh address. This new queue object is pending, 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 the value v, where $v \neq \bot$, and $v \notin \mathit{dom}(\pi)$ (i.e., it does not correspond to any queue object). First, we change the state of the receiver object from "pending" to "fulfilled". Second, we update the already registered callbacks (if any) by setting the value v as the only argument of them (forming the list t'). Third, we asynchronously fulfill any queue object that depend on the current one (see the list d). To do so, we form the list of callbacks f. Every element $(\alpha, \lambda x.x, [v]) \in f$ contains the identity function $\lambda x.x$ that is invoked with the value v. Upon exit, the identity function fulfills the queue object α, where $\alpha \in d$. Then, we add the updated callbacks t' and the list of functions f to the list of scheduled callbacks κ. Notice that the receiver must be neither l_{time} nor l_{io}. Also, note that the callbacks of f are scheduled before those included in t' (i.e., $\kappa' = \kappa :: (f :: t')$). This means that we fulfill any dependent queue objects before the execution of callbacks.

E-CONTEXT

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

NEWQ

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

FULFILL-PENDING

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

FULFILL-PEND-PEND

$$\frac{\begin{array}{c} \pi(p) \downarrow_1 = \text{pending} \qquad v \in dom(\pi) \\ (\text{pending}, t, k, d) = \pi(v) \qquad \pi' = \pi[v \mapsto (\text{pending}, t, k, d \cdot p)] \end{array}}{\pi, \phi, \kappa, \tau, E[p.\text{fulfill}(v)] \rightarrow \pi', \phi, \kappa, \tau, E[\text{undef}]}$$

FULFILL-PEND-FUL

$$\frac{\pi(p) \downarrow_1 = \text{pending} \qquad v \in dom(\pi) \qquad \pi(v) \downarrow_1 = (\text{fulfilled}, v')}{\pi, \phi, \kappa, \tau, E[p.\text{fulfill}(v)] \rightarrow \pi, \phi, \kappa, \tau, E[p.\text{fulfill}(v')]}$$

FULFILL-SETTLED

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

REGISTERFUL-PENDING

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

REGISTERFUL-FULFILLED

$$\frac{\begin{array}{c} p \neq l_{time} \wedge p \neq l_{io} \qquad \pi(p) \downarrow_1 = (\text{fulfilled}, v) \\ v \neq \bot \qquad \kappa' = \kappa \cdot (p', f, [v]) \end{array}}{\pi, \phi, \kappa, \tau, E[p.\text{registerFul}(f, p', n_1, n_2, \dots, n_n)] \rightarrow \pi, \phi, \kappa', \tau, E[\text{undef}]}$$

REGISTERFUL-FULFILLED-⊥

$$\frac{\begin{array}{c} p \neq l_{time} \wedge p \neq l_{io} \qquad \pi(p) \downarrow_1 = (\text{fulfilled}, \bot) \\ \kappa' = \kappa \cdot (p', f, [n_1, n_2, \dots, n_n]) \end{array}}{\pi, \phi, \kappa, \tau, E[p.\text{registerFul}(f, p', n_1, n_2, \dots, n_n)] \rightarrow \pi, \phi, \kappa', \tau, E[\text{undef}]}$$

REGISTERFUL-TIMER-IO-⊥

$$\frac{\begin{array}{c} p = l_{time} \vee p = l_{io} \qquad \pi(p) \downarrow_1 = (\text{fulfilled}, \bot) \\ \tau' = \tau \cdot (p', f, [n_1, n_2, \dots, n_n]) \end{array}}{\pi, \phi, \kappa, \tau, E[p.\text{registerFul}(f, p', n_1, n_2, \dots, n_n)] \rightarrow \pi, \phi, \kappa, \tau', E[\text{undef}]}$$

APPEND

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

POP

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

ERROR

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

Figure 4 The semantics of λ_{q}.

Remark. When we fulfill a queue object with a $\perp$ value (i.e., $v = \perp$), we do not update the arguments of the callbacks registered on the queue object p. In other words, we follow all the steps described in the [FULFILL-PENDING] rule except for creating the list t'. We omit the corresponding rule for brevity.

The [FULFILL-PEND-PEND] describes the scenario of fulfilling a pending queue object p with another pending queue object v. In this case, we do not fulfill the queue object p synchronously. Instead, we make it dependent on the queue object v given as an argument. To do so, we update the queue object v by adding p to its list of dependent queue objects (i.e., $d \cdot p$). Notice that 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 the fulfilled queue object v. Then, p resolves with the same value as the queue object v. This is expressed by the resulting expression $p.\mathsf{fulfill}(v')$.

The [FULFILL-SETTLED] rule illustrates the case when we try to fulfill a settled queue object. This rule does not update the state.

The [REGISTERFUL-PENDING] rule adds the provided callback f to the list of callbacks that we should execute 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. This means that p' is fulfilled upon the termination of f. Also, this rule adds any extra arguments passed in registerFul as the 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}). We use the fulfilled value of the receiver as the only argument of the given function. Like the previous rule, it relates the provided queue object p' with the execution of the callback. This time we do ignore any extra arguments passed in registerFul, as we fulfill the queue object p with a value that is not $\perp$.

The [REGISTERFUL-FULFILLED-$\perp$] rule describes the case where we register a callback f on a queue object fulfilled with a $\perp$ value. Unlike the [REGISTERFUL-FULFILLED] rule, this rule does not neglect any extra arguments passed in registerFul. In particular, it makes them parameters of the given callback. This distinction allows us to pass arguments explicitly to a callback. Most notably, these arguments are not dependent on the value with which a queue object is fulfilled or rejected. For example, this rules enables us to model extra arguments passed in a timer- or asynchronous I/O-related callback (e.g., `setTimeout(func, 10, arg1, arg2)`, etc.).

The [REGISTERFUL-TIMER-IO-$\perp$] rule is the same as the previous one, but this time we deal with queue objects located either at l_{time} or l_{io}. Thus, we add the given callback f to the list τ instead of κ.

The [APPEND] rule appends the 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 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 specified in that top element. In this way, we capture the actual behavior of the uncaught exceptions triggered during the execution of an asynchronous callback.

2.1.3 Modeling the Event Loop

A reader might wonder why do we keep two separate lists, i.e., the list τ for holding callbacks coming from the l_{time} or l_{io} queue objects, and the list κ for callbacks stemming from any other queue object. The intuition behind this design choice is that it is convenient for us to

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

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

Figure 5 The semantics of the event loop.

$$\begin{aligned}
&e \in \mathit{Exp} ::= \ \ldots \\
&\qquad \mid \mathsf{addTimerCallback}(e_1, e_2, e_3, \ldots) \mid \mathsf{addIOCallback}(e_1, e_2, e_3, \ldots) \\
&E ::= \ \ldots \\
&\qquad \mid \mathsf{addTimerCallbackCallback}(E, e, \ldots) \mid \mathsf{addTimerCallback}(v, \ldots, E, e, \ldots) \\
&\qquad \mid \mathsf{addIOCallback}(E, e, \ldots) \mid \mathsf{addIOCallback}(v, \ldots, E, e, \ldots)
\end{aligned}$$

Figure 6 Extending the syntax of λ_q to deal with timers and asynchronous I/O.

model the concrete semantics of the event loop correctly. In particular, the implementation of the event loop assigns different priorities to the callbacks depending on their kind [25, 32]. For example, the event loop processes a callback of a promise object before any timer- or asynchronous I/O-related callback regardless of their registration order.

In this context, Figure 5 demonstrates the semantics of the event loop. The [EVENT-LOOP] rule pops the first scheduled callback from the list κ. We get the queue object q included in that callback, and we attach it to the front of the queue chain. Adding q to the top of the queue chain allows us to reject that queue object, when 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 (recall the [FULLFILL-SETTLED] rule). Furthermore, observe how the event loop is reduced, i.e., $q.\mathsf{fulfill}(f(a)); \texttt{pop}(); \bullet$. Specifically, once we execute the callback f and fulfill the dependent queue object q with the return value of f, we evaluate the `pop()` expression. This means that we pop the top element of the queue chain before re-evaluating the event loop. This is an invariant of the semantics of the event loop: every time we evaluate it, the queue chain is always empty.

The [EVENT-LOOP-TIMERS-IO] rule handles the case when the list κ is empty. In other words, the rule states that when there are not any callbacks that neither come from the l_{time} nor the l_{io} queue object, inspect the list τ, and pick *non-deterministically* one of those. Selecting a callback non-deterministically allows us to over-approximate the actual behavior of the event loop regarding its different execution phases [25]. Overall, the rule describes the scheduling policy presented in the work of Loring et. al. [25], where initially we look for any promise-related callback (if any). Otherwise, we choose any callback associated with timers or asynchronous I/O at random.

2.1.4 Modeling Timers & Asynchronous I/O

To model timers and asynchronous I/O, we follow a similar approach to the work of Loring et. al. [25]. Specifically, we start with an initial queue π that contains two queue objects: the q_{time}, and q_{io} that are located at l_{time} and l_{io} respectively. Both q_{time} and q_{io} are initialized as $((\mathsf{fulfilled}, \bot), [], [], [])$. We extend the syntax of λ_q by adding two more expressions. Figure 6

ADD-TIMER-CALLBACK

$$\frac{q = \pi(l_{time})}{\pi, \phi, \kappa, \tau, \mathsf{addTimerCallback}(f,\ n_1, \dots) \rightarrow \pi, \phi, \kappa, q.\mathsf{registerFul}(f,\ q,\ n_1, \dots)}$$

ADD-IO-CALLBACK

$$\frac{q = \pi(l_{io})}{\pi, \phi, \kappa, \tau, \mathsf{addIOCallback}(f,\ n_1, \dots) \rightarrow \pi, \phi, \kappa, q.\mathsf{registerFul}(f,\ q,\ n_1, \dots)}$$

Figure 7 Extending the semantics of λ_q to deal with timers and asynchronous I/O.

shows the extended syntax of λ_q to deal with timers and asynchronous I/O, while Figure 7 presents the rules related to those expressions.

The new expressions have high correspondence to each other. Specifically, the $\mathsf{addTimerCallback}(\dots)$ construct adds the callback e_1 to the queue object located at the address l_{time}. The arguments of that callback are any optional parameters passed in $\mathsf{addTimerCallback}$, i.e., e_2, e_3, and so on. From Figure 7, we observe that the [ADD-TIMER-CALLBACK] rule retrieves the queue object q corresponding to the address l_{time}. Recall again that the l_{time} can be found in the initial queue. Then, the given expression is reduced to $q.\mathsf{registerFul}(f,\ q,\ n_1, \dots)$. In particular, we add the 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 $\mathsf{registerFul}$. Recall from Figure 4, according to the [FULFILL-SETTLED] rule, trying to fulfill (and similarly to reject) a settled queue object does not have any effect on the state. Beyond that, since q is fulfilled with $\bot$, the extra arguments (i.e., $n_1, \dots$) are also passed as arguments in the invocation of f.

The semantics of the $\mathsf{addIOCallback}(\dots)$ primitive is the same with that of $\mathsf{addTimerCallback}(\dots)$; however, this time, we use the queue object located at l_{io}.

2.2 Expressing Promises in Terms of λ_q

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

```
Promise.resolve = function(value) {
  var promise = newQ();
  if (typeof value.then === "function") {
    var t = newQ();
    t.fulfill(⊥);
    t.registerFul(value.then, t, promise.fulfill, promise.reject);
  } else
    promise.fulfill(value);
  return promise;
}
```

Figure 8 Expressing `Promise.resolve` in terms of λ_q.

Example – Modeling Promise.resolve(). In Figure 8, we see how we model the `Promise.resolve()` function in terms of λ_q[3]. The JavaScript `Promise.resolve()` function creates a new promise, and resolves it with the given value. According to ECMAScript, if the given `value` is a *thenable*, (i.e., an object that has a property named "then" and that property is a callable), the created promise resolves asynchronously. Specifically, we execute the function `value.then()` asynchronously, and we pass the resolving functions (i.e., fulfill, reject) as its arguments. Observe how the expressiveness of λ_q can model this source of asynchrony (lines 4–6), which we cannot model through the previous work [26, 25]. First, we create a fresh queue object `t`, and we fulfill it with $\perp$ (lines 4, 5). Then, at line 6, we schedule the execution of `value.then()` by registering it on the newly created queue object `t`. Notice that we also pass `promise.fulfill` and `promise.reject` as extra arguments. This means that those functions will be the actual arguments of `value.then()` because `t` is fulfilled with $\perp$. On the other hand, if `value` is not a thenable, we synchronously resolve the created promise using the `promise.fulfill` construct at line 8.

3 The Core Analysis

The λ_q calculus presented in Section 2 is the touchstone of the static analysis proposed for asynchronous JavaScript programs. The analysis is designed to be sound; thus, we devise abstract domains and semantics that over-approximate the behavior of λ_q. Currently, there are few implementations available for asynchronous JavaScript, and previous efforts mainly focus on modeling the event system of client-side applications [18, 33]. To the best of our knowledge, it is the first static analysis for ES6 promises. The rest of this section describes the details of the analysis.

3.1 The Analysis Domains

$$\begin{aligned}
l \in \widehat{Addr} &= \{l_i \mid i \text{ is an allocation site}\} \cup \{l_{time}, l_{io}\}\\
\pi \in \widehat{Queue} &= \widehat{Addr} \hookrightarrow \mathcal{P}(\widehat{QueueObject})\\
q \in \widehat{QueueObject} &= \widehat{QueueState} \times \mathcal{P}(\widehat{Callback}) \times \mathcal{P}(\widehat{Callback}) \times \mathcal{P}(\widehat{Addr})\\
qs \in \widehat{QueueState} &= \{\mathsf{pending}\} \cup (\{\mathsf{fulfilled}, \mathsf{rejected}\} \times Value)\\
clb \in \widehat{Callback} &= \widehat{Addr} \times F \times Value^*\\
\kappa \in \widehat{ScheduledCallbacks} &= (\mathcal{P}(\widehat{Callback}))^*\\
\tau \in \widehat{ScheduledTimerIO} &= (\mathcal{P}(\widehat{Callback}))^*\\
\phi \in \widehat{QueueChain} &= (\mathcal{P}(\widehat{Addr}))^*
\end{aligned}$$

Figure 9 The abstract domains of λ_q.

[3] For brevity, Figure 8 omits some steps described in the specification of the `Promise.resolve()` function. For example, according to ECMAScript, if `this` value of the `Promise.resolve()` function is not an object, then a `TypeError` is thrown. In the implementation, though, we follow all the steps that are described in the specification.

Figure 9 presents the abstract domains of the $\lambda_{\mathtt{q}}$ calculus that underpin our static analysis. Below we make a summary of our primary design choices.

Abstract Addresses: As a starting point, we employ allocation site abstraction for modeling the space of addresses. It is the standard way used in literature for abstracting addresses that keeps the domain finite [19, 27]. Notice that we still define two internal addresses, i.e., l_{time}, l_{io}, corresponding to the addresses of the queue objects responsible for timers and asynchronous I/O respectively.

Abstract Queue: We define an abstract queue as the partial map of abstract addresses to an element of the power set of abstract queue objects. Therefore, an address might point to multiple queue objects. This abstraction over-approximates the behavior of $\lambda_{\mathtt{q}}$ and allows us to capture all possible program's behaviors that might stem from the analysis imprecision.

Abstract Queue Objects: A tuple consisting of an abstract queue state – observe that the domain of abstract queue states is the same as $\lambda_{\mathtt{q}}$ – two sets of abstract callbacks (executed on fulfillment and rejection respectively), and a set of abstract addresses (used to store the queue objects that are dependent on the current one) represents an abstract queue object. Notice how this definition differs from that of $\lambda_{\mathtt{q}}$. First, we do not keep the registration order of callbacks; therefore, we convert the two lists into two sets. The programming pattern related to promises supports our design decision. Specifically, developers often use promises as a chain; registering two callbacks on the same promise object is quite uncommon. Madsen et. al. [27] made similar observations for the event-driven programs.

This abstraction can negatively affect precision only when we register multiple callbacks on a *pending* queue object. Recall from Figure 4, when we register a callback on a settled queue object, we can precisely track its execution order since we directly add it to the list of scheduled callbacks.

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

Abstract Callback: An abstract callback comprises one abstract address, one function, and a list of values that stands for the arguments of the function. Recall that the abstract address corresponds to the queue object that the return value of the function fulfills.

Abstract List of Scheduled Callbacks: We use a list of sets to abstract the domain that is responsible for maintaining the callbacks that are ready for execution (i.e., $\widehat{ScheduledCallbacks}$ and $\widehat{ScheduledTimerIO}$). In this context, the i^{th} 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, we might execute y before z and vice versa.

Note that a critical requirement of our domains' definition 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 event loop might process the same callback multiple times. Therefore, we have to add it to the lists κ or τ more than one time. For that reason, those lists monitor the execution order of callbacks up to a certain limit n. The execution order of the callbacks scheduled after that limit is not preserved; thus, the analysis places them into the same set.

Abstract Queue Chain: The analysis uses the last component of our abstract domains to capture the effects of uncaught exceptions during the execution of callbacks. We define it as a sequence of sets of addresses. Based on the abstract translation of the semantics of $\lambda_{\mathtt{q}}$, 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 propagates the exception to the caller function as usual. 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 one because the event loop executes only one callback at a time. The only case when the abstract queue chain contains multiple elements is when we have nested promise executors. A promise executor is a function passed as an argument in a promise constructor. However, since we cannot have an unbound number of nested promise executors, the size of the abstract queue chain remains finite.

3.1.1 Tracking the Execution Order

Promises. Estimating the order in which the event loop executes promise-related callbacks is straightforward because it is a direct translation of the corresponding semantics of $\lambda_{\mathtt{q}}$. In particular, there are two possible cases:

- *Settle a promise that has registered callbacks:* When we settle (i.e., either fulfill or reject) a promise object that has registered callbacks, we schedule those callbacks associated with the next state of the promise by putting them on the tail of the list κ. For instance, if we fulfill a promise, we append all the callbacks triggered on fulfillment on the list κ. A reader might observe that when there are multiple callbacks registered on the same promise object, we put them on the same set which is the element that we finally add to κ. This is justified by the fact that an abstract queue object does not keep the registration order of its 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 the list κ) depending on the state of that promise, i.e., we schedule the callback `f1` if `x` is fulfilled and `f2` if `x` 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; thus, the event loop might process them in any order. However, we *do* keep track the execution order between nested callbacks.

3.2 Callback Graph

In this section, we introduce the concept of *callback graph*; a fundamental component of our analysis that captures how data flow is propagated between different asynchronous callbacks. A callback graph is defined as an element of the following power set:

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

We define every node of a callback graph as $n \in \mathit{Node} = C \times F$, where C is the domain of contexts while F is the set of all the functions of the program. Every element of a callback graph $(c_1, f_1, c_2, f_2) \in cg$, where $cg \in \mathit{CallbackGraph}$ has the following meaning: *the function f_2 in context c_2 is executed after the function f_1 in context c_1*. We can treat the above statement as the following expression: $f_1(\dots); f_2(\dots);$

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

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

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

$$\begin{aligned} n_1 \rightarrow_{cg} n_2 &\Rightarrow n_1 \rightarrow^{+}_{cg} n_2 \\ n_1 \rightarrow^{+}_{cg} n_2 \wedge n_2 \rightarrow^{+}_{cg} n_3 &\Rightarrow n_1 \rightarrow^{+}_{cg} n_3, \quad \textit{where } n_3 \in \mathit{Node} \end{aligned}$$

Definition 1 and Definition 2 introduce the concept of path between two nodes in a callback graph $cg \in \mathit{CallbackGraph}$. In particular, the relation $\rightarrow_{cg}$ denotes that there is path of length one between two nodes n_1, n_2, i.e., $(n_1, n_2) \in cg$. 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 property of a callback graph is that it does not have any cycles, i.e.,

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

Notice that if $n_1 \not\rightarrow^{+}_{cg} n_2$, and $n_2 \not\rightarrow^{+}_{cg} n_1$ hold, the analysis cannot estimate the execution order between n_1 and n_2. Therefore, we presume that n_1 and n_2 can be called in any order.

Callback graph is computed on the fly as the analysis progresses. Callback graph exploits both the lists κ and τ, and constructs the $\rightarrow_{cg}$ relations between callbacks by respecting their execution order as specified in those lists.

Previous work has proposed similar program representations for asynchrony. Madsen et. al. [27] introduce the event-based call graph that abstracts the data-flow of the event-based JavaScript programs. However, it does not support promises. More recently, promise graph [26, 1] has been used for debugging promise-related programs. Our callback graph is distinguished from promise graph, as it also captures callbacks that stem from timers or asynchronous I/O. Therefore, we can handle common programming patterns where we mix promises with asynchronous I/O (Figure 1). Also, since the promise graph aims to detect anti-patterns related to promise code (e.g., unsettled promises), it does not track the order in which promises are settled. Therefore, it misses the happens-before relations between the corresponding callbacks.

3.3 Analysis Sensitivity

Here, we introduce two methods for boosting the analysis precision of asynchronous code.

3.3.1 Callback Sensitivity

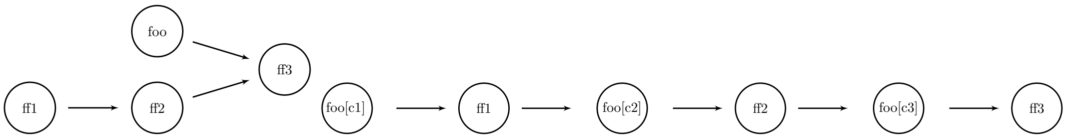
Knowing the temporal relations between asynchronous callbacks enables us to capture how data flow is propagated precisely. Typically, a naive flow-sensitive analysis, which exploits the control flow graph (CFG), represents the event loop as a single program point with only one context corresponding to it. Therefore – unlike traditional function calls – the analysis misses the happens-before relations between callbacks because they are triggered by the same program location (i.e., the event loop).

To address those issues, we exploit the callback graph to devise a more precise analysis, which we call *callback-sensitive* analysis. The callback-sensitive analysis propagates the state with regards to the $\rightarrow_{cg}$ and $\rightarrow^{+}_{cg}$ relations found in a callback graph $cg \in \mathit{CallbackGraph}$. Specifically, when the analysis needs to propagate the resulting state from the exit point of a

```
function foo() { ... }

var x = Promise.resolve()
  .then(foo)
  .then(function ff1() { ... })
  .then(foo)
  .then(function ff2() { ... })
  .then(foo)
  .then(function ff3() { ... });
```

Figure 10 An example program where we create a promise chain. Notice that we register the function `foo` multiple times across the chain.



(a) QR-insensitive analysis **(b)** QR-sensitive analysis

Figure 11 Callback graph of program of Figure 10 produced by the QR-insensitive and QR-sensitive analysis respectively.

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 callback nodes $y \in Node$ where $x \rightarrow_{cg} y$ holds. In other words, the edges of a callback graph reflect how the state is propagated from the exit point of a callback node x to the entry point of a callback node 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.

Remark. Callback-sensitivity does not work with contexts to improve the precision of the analysis. We still represent the event loop 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 is still some imprecision. However, callback-sensitivity minimizes the number of those joins, as they are only caused by the callbacks invoked last.

3.3.2 Context-Sensitivity

Recall from Section 3.2 that a callback graph is defined as $\mathcal{P}(Node \times Node)$, where $n \in Node = C \times F$. It is possible to increase the precision of a callback graph by distinguishing callbacks based on the context in which they are invoked. Existing flavors of context-sensitivity are not so useful in differentiating asynchronous functions from each other. For instance, object-sensitivity [30, 24], which separates invocations based on the value of the receiver – and has been proven to be particularly effective for the analysis of object-oriented languages – is not fruitful in the context of asynchronous callbacks because in most cases the receiver of callbacks corresponds to the global object. Similarly, previous work in the static analysis of JavaScript [19, 21] creates a context with regards to the arguments of a function. Such a strategy might not be effective in cases where a callback expects no arguments or the arguments from two different calls are indistinguishable.

We introduce one novel context-sensitivity flavor – which we call *QR-sensitivity* – as an effort to boost the analysis precision. QR-sensitivity separates callbacks according to: (1) the queue object that they belong to (Q), and (2) the queue object their return value fulfills (R). In this case, the domain of contexts is given by:

$$c \in C = \widehat{Addr} \times \widehat{Addr}$$

In other words, every context is a pair $(l_q, l_r) \in \widehat{Addr} \times \widehat{Addr}$, where l_q stands for the allocation site of callback's queue object, and l_r is the abstract address of the queue object that the return value of the callback fulfills. Notice that this domain is finite, so the analysis always terminates.

As a motivating example, consider the program of Figure 10. This program creates a promise chain where we register different callbacks at every step of the asynchronous computation. At line 1, we define the function `foo()`. We asynchronously call `foo()` multiple times, i.e., at lines 4, 6, and 8. Recall that chains of promises enable us to enforce a deterministic execution of the corresponding callbacks. Specifically, based on the actual execution, the event loop invokes the callbacks in the following order: `foo()` → `ff1()` → `foo()` → `ff2()` → `foo()` → `ff3()`. Figure 11a presents the callback graph of the program of our example produced by a QR-insensitive analysis. In this case, the analysis considers the different invocations of `foo()` as identical. As a result, the analysis loses the temporal relation between `foo()` and `ff1()`, `ff2()` – indicated by the fact that the respective nodes are not connected to each other – because `foo()` is called both before and after `ff1()` and `ff2()`. On the contrary, a QR-sensitive analysis ends up with an entirely precise callback graph as shown in Figure 11b. The QR-sensitive analysis distinguishes the different invocations of `foo()` from each other because it creates three different contexts; one for every call of `foo()`. Specifically, we have $c_1 = (l_3, l_4), c_2 = (l_5, l_6), c_3 = (l_7, l_8)$, where l_i stands for the promise object allocated at line i. For example, the second invocation of `foo()` is related to the promise object created by the call of `then()` at line 5, and its return value fulfills the promise object allocated by the invocation of `then()` at line 6.

3.4 Implementation

Our prototype implementation[4] extends *TAJS* [19, 20, 18]; a state-of-the-art static analyzer for JavaScript. TAJS analysis is implemented as an instance of the abstract interpretation framework [3], and it is designed to be sound. It uses a lattice specifically designed for JavaScript that is capable of handling the vast majority of JavaScript's complicated features and semantics. TAJS analysis is both flow- and context-sensitive. The output of the analysis is the set of all reachable states from an initial state along with a call graph. TAJS can detect various type-related errors such as the use of a non-function variable in a call expression, property access of `null` or `undefined` variables, inconsistencies caused by implicit type conversions, and many others [19].

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 and is guided by the design of our analysis. Specifically, we first incorporate the domains presented in Figure 9 into the definition of the abstract state of TAJS. Then, we provide models for promises written in Java by faithfully following the ECMAScript specification. Recall again that our models exploit the λ_{q} calculus presented in Section 2 and they produce

[4] `https://github.com/theosotr/async-tajs`

```
 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 12 A model for `fs.open` function. 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.

side-effects that over-approximate the behavior of JavaScript promises. Beyond that, we implement models for the special constructs of λ_q (i.e., addTimerCallback, addIOCallback) that are used for adding callbacks to the timer- and asynchronous I/O-related queue objects respectively. We implement the models for timers in Java; however, we write JavaScript models for asynchronous I/O operations, when it is necessary.

For example, Figure 12 shows the JavaScript code that models the function `open()` of the `fs` Node.js module. In particular, `open()` asynchronously opens a given file. When I/O operation completes, the callback provided by the developer is called with two arguments: (1) `err` that is not `undefined` when 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`, when any error occurs. Our model first makes `open()` parameter-sensitive on the third argument that corresponds to the callback provided by the programmer. Then, at lines 3 and 4, it initializes the arguments of the callback, (i.e., `err` and `fd`). Observe that we initialize 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 the special function `TAJS_addAsyncIOCallback()` that registers the given callback on the queue object responsible for I/O operations, implementing the semantics of the addIOCallback primitive from our λ_q calculus.

3.5 Limitations

Although our analysis aims to support all the asynchronous features of JavaScript up to the 7th version of ECMAScript, it does not handle the `Promise.all()` function of the Promise API. This function expects an iterable of promises, and it creates a new object that is fulfilled whenever all promises included in that iterable are fulfilled. Statically capturing all program's behaviors that stem from `Promise.all()` is challenging because the analysis imprecision might cause the number of behaviors to grow exponentially. However, `Promise.all()` is less common than other functions of the Promise API such as `Promise.resolve()` or `Promise.reject()`.

Some of our design choices about analysis abstractions might lead to imprecision. For example, we do not track the registration order of a pending promise's callbacks. Therefore, when we settle such a promise, the analysis assumes that all its registered callbacks can be invoked in any order. However, as we mentioned in Section 3.1.1, that programming pattern (i.e., adding multiple callbacks to the same pending object) is quite rare.

Table 1 List of the selected macro-benchmarks and their description. 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.).

Benchmark	LOC	ELOC	Files	Dependencies	Promises	Timers/Async I/O
controlled-promise	225	225	1	0	4	1
fetch	517	1,118	1	1	12	4
honoka	324	1,643	6	6	4	1
axios	1,733	1,733	26	0	13	2
pixiv-client	1,031	3,469	1	2	64	2
node-glob	1,519	6,131	3	6	0	5

The analysis sensitivity options introduced in Section 3.3.2 might not be so effective when dealing with timers or asynchronous I/O. Since we follow a conservative approach for modeling the execution order of timers and asynchronous I/O – regardless of the registration order of their callbacks – keeping a more precise state does not necessarily lead to a more precise callback graph.

4 Evaluation

In this section, 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 parameterizations of the analysis, and report the precision and performance metrics.

4.1 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. Beyond micro-benchmarks, we evaluate our analysis on 6 real-world JavaScript modules. The most common macro-benchmarks for static analyses used in the literature are those provided by JetStream[5], and V8 engine[6][19, 21, 22]. However, those benchmarks are not suitable for our case because they are not asynchronous. To find interesting benchmarks, we developed an automatic mechanism for collecting and analyzing Github repositories. 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[7]. The second option employed the Github API[8] to find JavaScript repositories that are related to promises. We then investigated the Github repositories that we collected at the first phase by computing various metrics such as lines of code, number of promise-, timer- and asynchronous IO-related statements. We manually selected the 6 JavaScript modules presented in Table 1. Most of them are libraries for performing HTTP requests or file system operations.

[5] `https://browserbench.org/JetStream/`
[6] `http://www.netchain.com/Tools/v8/`
[7] `https://www.npmjs.com/browse/depended`
[8] `https://developer.github.com/v3/`

Table 2 Precision on micro-benchmarks.

	Analyzed 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**
micro01	5	5	4	4	0.8	0.8	1.0	1.0	2	2	0	0
micro02	3	3	3	3	1.0	1.0	1.0	1.0	1	1	0	0
micro03	2	2	2	2	1.0	1.0	1.0	1.0	1	1	0	0
micro04	4	4	4	4	0.5	0.5	0.5	0.5	1	1	1	1
micro05	8	8	7	7	0.96	0.96	1.0	1.0	3	3	0	0
micro06	11	11	11	11	1.0	1.0	1.0	1.0	3	3	1	1
micro07	14	14	13	13	0.86	0.87	1.0	1.0	1	1	0	0
micro08	5	5	5	5	0.8	0.8	0.8	0.8	1	1	0	0
micro09	5	5	4	4	0.9	0.9	1.0	1.0	1	1	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	0.9	0.9	1.0	1.0	2	2	0	0
micro13	4	4	3	3	0.83	0.83	1.0	1.0	1	1	0	0
micro14	6	6	5	5	0.8	0.8	1.0	1.0	2	2	0	0
micro15	6	6	6	6	0.8	0.8	1.0	1.0	0	0	0	0
micro16	6	6	6	6	1.0	1.0	1.0	1.0	1	1	0	0
micro17	3	3	3	3	0.67	0.67	0.67	0.67	2	2	2	2
micro18	4	3	4	3	0.83	1.0	0.83	1.0	1	0	1	0
micro19	14	7	14	7	0.73	0.93	0.74	1.0	0	0	0	0
micro20	6	6	6	6	0.93	0.93	1.0	1.0	0	0	0	0
micro21	5	5	4	4	0.9	0.9	1.0	1.0	1	1	0	0
micro22	6	6	5	5	0.87	0.87	0.9	0.9	1	1	0	0
micro23	6	6	5	5	0.87	0.87	1.0	1.0	3	3	1	1
micro24	3	3	3	3	1.0	1.0	1.0	1.0	2	2	1	1
micro25	8	8	8	8	0.79	0.79	0.79	0.79	1	1	0	0
micro26	9	9	7	7	0.89	0.89	1.0	1.0	3	3	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	1.0	0.5	1.0	0	0	0	0
Average	**5.83**	**5.55**	**5.45**	**5.17**	**0.85**	**0.88**	**0.91**	**0.94**	**1.45**	**1.41**	**0.55**	**0.52**
Total	**169**	**161**	**158**	**150**					**42**	**41**	**16**	**15**

We experiment with 4 different analyses: (1) an analysis that is neither callback- nor QR-sensitive (NC-No), (2) a callback-insensitive but QR-sensitive analysis (NC-QR), (3) a callback-sensitive but QR-insensitive analysis (C-No), and (4) a both callback- and QR-sensitive analysis (C-QR). Note that recency abstraction [2] – which is a technique for minimizing weak updates and is natively supported by TAJS – is enabled for every analysis. For every analysis, we use object-sensitivity, while we enable parameter-sensitivity in certain functions for further boosting the precision of top-level code. Finally, the lists κ and τ are bounded by $n = 30$.

We evaluate the precision of each analysis in terms of the number of the analyzed callbacks, the precision of the computed callback graph, and the number of reported type errors triggered by the execution of asynchronous callbacks. We define the precision of a callback graph as 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, i.e., the number of reported type errors as in the work of Kashyap et. al. [21]. The fewer type errors an analysis reports, the more precise it is. The same applies to the number of callbacks inspected by the analysis. To compute the performance characteristics of every analysis, we run every experiment ten times to get reliable measurements. All the experiments were run on a machine with an Intel i7 2.4GHz quad-core processor and 8GB of RAM.

Table 3 Precision on macro-benchmarks.

	Analyzed 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	8	8	7	7
honoka	8	8	6	6	0.929	0.929	1.0	1.0	1	1	0	0
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	3	3	3	2
node-glob	3	3	3	3	0.667	0.667	0.667	0.667	19	19	19	19
Average	**12**	**12**	**10.8**	**10.5**	**0.79**	**0.848**	**0.805**	**0.88**	**6**	**6**	**5.1**	**5**
Total	**72**	**72**	**65**	**63**					**36**	**36**	**32**	**31**

4.2 Results

Micro-benchmarks. Table 2 shows how precise every analysis is on every micro-benchmark. Starting with callback-insensitive analyses (see the NC-No and NC-QR columns), we observe that in general QR-sensitivity improves the precision of the callback graph by 3.6%, on average. That small boost of QR-sensitivity is explained by the fact that *only* 3 out of 29 micro-benchmarks invoke the same callback multiple times.

Recall from Section 3.3.2 that QR-sensitivity is used to distinguish different calls of the same callback. Therefore, if one program does not use a specific callback multiple times, QR-sensitivity does not make any difference. However, if we focus on the results of the micro-benchmarks where we come across such behaviors, i.e. micro18, micro19, and micro29, we get a significant divergence of the precision of callback graph. Specifically, QR-sensitivity improves the precision by 20.5%, 27.4% and 100% in micro18, micro19 and micro29 respectively. Besides that, in micro19, there is a striking decrease in the number of the analyzed callbacks: the QR-insensitive analyses inspect 14 callbacks, while the QR-sensitive analyses examine only 7.

The results regarding the number of type errors are almost identical for every analysis: a QR-insensitive analysis reports 42 type errors in total, whereas all the other QR-sensitive analyses produce warnings for 41 cases.

Moving to callback-sensitive analyses, the results indicate clear differences. First, a callback-sensitive but QR-insensitive analysis reports only 16 type errors in total (i.e., 61.9% fewer type errors than callback-insensitive analyses), and amplifies the average precision of the callback graph from 0.85 to 0.91. As before, the QR-sensitive analyses boost the precision of the callback graph by 20.4%, 35.1%, and 100% in micro18, micro19, and micro29 respectively. Finally, a callback-sensitive and QR-insensitive analysis decreases the total number of the analyzed callbacks from 169 to 158. Notice that when callback- and QR-sensitivity are combined, the total number of callbacks is reduced by 11.2%.

Macro-benchmarks. Table 3 reports the precision metrics of every analysis on the macro-benchmarks. First, we make similar observations as those of micro-benchmarks. In general, QR-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 13.9% fewer type errors in total. Additionally, when we combine QR- and callback-sensitivity, we can boost the analysis precision for 5 out of 6 benchmarks. Specifically, the QR- and callback-sensitive analysis improves the callback graph precision by up to 28.5% (see the `axios` benchmark), and achieves a 88% callback graph precision, on average. On the other hand, the naive analysis (neither QR- nor callback-sensitive) reports only a 79% precision for callback graph, on average.

Table 4 Times of different analyses in seconds.

	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

By examining the results for the `node-glob` benchmark, we see that every analysis produces identical results. `node-glob` uses only timers and asynchronous I/O operations. As we mentioned in Section 3.5, in such cases, neither callback- nor QR-sensitivity is effective as we follow a conservative approach for modelling timers and asynchronous I/O. For example, we assume that two callbacks x and y are executed in any order, even if x is scheduled before y (and vice versa).

Table 4 gives the running times of every analysis on macro-benchmarks. We notice that in some benchmarks (see fetch) a more precise analysis decreases the running times by 3%–18%. This is justified by the fact that a more precise analysis might compress the state faster than an imprecise analysis. For instance, in fetch, an imprecise analysis led to the analysis of 3 spurious callbacks, yielding to a higher analysis time. The results appear to be consistent with those of the recent literature that suggest that precision might lead to a faster analysis in some cases [33]. On the other hand, we observe a non-trivial fall in the analysis performance in only one benchmark. Specifically, the analysis sensitivity increased the running times of `pixiv-client` by 12%–30.6%. However, such an increase seems to be acceptable.

4.3 Case Studies

In this section, we describe some case studies coming from the macro-benchmarks.

fetch. Figure 13 shows a code fragment taken from fetch[9]. Note that we omit irrelevant code for brevity. The function `Body()` defines a couple of methods (e.g., `text()`, `formData()`) for manipulating the body of a response. Observe that those methods are registered on the prototype of `Response` using the function `Function.prototype.call()` at line 45. Note that `Body` also contains a method (i.e., `_initBody()`) for initializing the body of a response according to the type of the input. To this end, the `Response` constructor takes a body as a parameter and initializes it through the invocation of `_initBody()` (lines 41, 43). The function `text()` reads the body of a response in a text format (lines 20–34). When the body of the response has been already read, `text()` returns a rejected promise (lines 3, 22–23). Otherwise, it marks the property `bodyUsed` of the response object as true (line 5), and then it returns a promise object depending on the type of the body of the given response (lines 25–33). The function `formData()` (lines 36–38) asynchronously reads the body of a response in a text format, and then it parses it into a `FormData` object[10] through the call of the function `decode()`. The function `fetch()` (lines 47–56) makes a new asynchronous request.

[9] https://github.com/github/fetch

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

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

Figure 13 Code fragment taken from fetch.

Listing 1 Case 1.

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

Listing 2 Case 2.

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

Figure 14 Code fragments which use the `fetch` API.

When the request completes successfully, the callback `onLoad()` is executed asynchronously (line 51). This callback finally fulfills the promise returned by `fetch()` with a response object that contains the response of the server (line 53).

In Listing 1, we make an asynchronous request to the endpoint "/helloWorld" using the `fetch` API. Upon success, we schedule the callback `foo()`. Recall that the parameter `value` of `foo()` corresponds to the response object coming from line 53 (Figure 13). In `foo()`, we convert the response of the server into a `FormData` object (line 2). A callback-insensitive analysis, which considers that the event loop executes all callbacks in any order, merges all the data flow stemming from those callbacks into a single point. As a result, the side effects of `onLoad()` and `foo()` are directly propagated to the event loop. In turn, the event loop propagates the resulting state again to those callbacks. This is repeated until convergence. Specifically, the callback `foo()` calls `value.formData()` that updates the property `bodyUsed` of the response object to true (Figure 13, line 5). The resulting state is propagated to the event loop where is joined with the state that stems from the callback `onLoad()`. Notice that the state of `onLoad()` indicates that `bodyUsed` is false because the callback `onLoad()` creates a fresh response object (Figure 13, lines 10, 53). The join of those states changes the abstract value of `bodyUsed` to $\top$. That change is propagated again to `foo()`.

This imprecision makes the analysis to consider both the `if` and `else` branches at lines 2–5. Thus, the analysis allocates a rejected promise at line 3, 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 at line 23. The value returned by `consumed()` is finally propagated to `formData()` at line 37, where the analysis reports a false positive; a property access of an `undefined` variable (access of the 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, the callback `foo()` is executed *after* the callback `onLoad()`. Therefore, the analysis propagates a more precise state to the entry of `foo()`: the state resulted by the execution of `onLoad()`, where a new response object is initialized with the field `bodyUsed` set to false.

In Listing 2, we initialize a response object with a body that has a string type (line 1). In turn, by calling the `formData()` method, we first read the body of the response in a text format, and then we decode it into a `FormData` object by asynchronously calling the `decode()` function (Figure 13, line 37). Since the body of the response is already in a text format, `text()` returns a fulfilled promise (Figure 13, line 32). At the same time, at line 4 of Listing 2, we allocate a fresh response object whose body is an instance of `Blob`[11]. Therefore, calling `formData()` schedules function `decode()` again. However this time, the callback `decode()` is registered on a different promise because the second call of `text()` returns a promise created by the function `readBlobAsText()` (Figure 13, line 26). A QR-sensitive analysis – which creates a context according to the queue object a callback belongs to – is capable of separating the two invocations of `decode()` because the first call of `decode()` is registered on the promise object that comes from line 32, whereas the second call of `decode()` is added to the promise created by `readBlobAsText()` at line 26.

honoka. We return back to Figure 1. Recall that a callback-insensitive analysis reports a spurious type error at line 17 when we try to access the property `headers` of `honoka.response` because it considers the case where the callback defined at lines 15–23 is executed before that defined at lines 2–14. Thus, `honoka.response` might be uninitialized (recall that `honoka.response` is initialized during the execution of the first callback at line 3). On the other hand, a callback-sensitive analysis consults the callback graph when it is time to propagate the state from the exit point of a callback to the entry point of the next one. In particular, when we analyze the exit node of the first callback, we propagate the current state to the second callback. Therefore, the entry point of the second function has a state that contains a precise value for `honoka.response`, that is, the object coming from the assignment at line 3.

4.4 Threats to Validity

Below we pinpoint the main threats to the validity of our results:

- Our analysis is an extension of TAJS. Therefore, 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 native functions of the JavaScript language unsoundly. For instance, we unsoundly model the native function

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

`Object.freeze()`, which is used to prevent an object from being updated. Specifically, the model of `Object.freeze()` simply returns the object given as argument.

- 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 modeling might neglect some of the side-effects that stem from the interaction with those APIs, leading to unsoundness [15, 33].
- Our macro-benchmarks consist of JavaScript libraries. Therefore, we need 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.

5 Related Work

In this section, we briefly present previous work related to formalization and program analysis for (asynchronous) JavaScript.

Semantics. Maffeis et al. [29] presented one of the first formalizations of JavaScript by designing small-step operational semantics for a subset of the 3rd version of ECMAScript. In subsequent work, Guha et al.[16] expressed the semantics of the 3rd edition of ECMAScript through a different approach; they developed a lambda calculus called λ_{JS}, and provided a desugaring mechanism for converting JavaScript code into λ_{JS}. We used λ_{JS} as a base for modeling asynchronous JavaScript. Later, Gardner et al. [13] introduced a program logic for reasoning about client-side JavaScript programs that support ECMAScript 3. They presented big-step operational semantics on the basis of that proposed by Maffeis et. al. [29], and they introduced inference rules for program reasoning which are highly inspired from separation logic [35]. More recently, Madsen et al. [26] and Loring et al. [25] extended λ_{JS} for modeling promises and asynchronous JavaScript respectively. Our model is a variation of their work; our modifications enable us to model different asynchronous features. Some of them are not handled by their models.

Static Analysis for JavaScript. Guarnieri et al. [15] proposed a pointer analysis for a subset of JavaScript. They precluded the use of `eval`-family functions from their analysis as their work focused on widgets where the use of `eval` is not common. It was one of the first attempts that managed to model some of the most peculiar features of JavaScript, such as prototype-based inheritance. TAJS [19, 20, 18] is a typer analyzer for JavaScript which is implemented as a classical dataflow analysis. Our work is implemented as an extension of TAJS. SAFE [23] is a static analysis framework that provides three different formal representations of JavaScript programs: an abstract syntax tree (AST), an intermediate language (IR) and a control-flow graph (CFG). SAFE implements a default analysis phase that 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 that operates on top of CFG. JSAI [21] implements an analysis through the abstract interpretation framework [3]. 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, the abstract semantics is specified on a CESK abstract machine [10], 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 analysis is configurable with different flavors of context-sensitivity which are plugged into the analysis through the widening operator used in the fix-point calculation [17].

Existing static analyses provide sufficient support for precisely modeling browser environment. Jensen et al. [18] modeled HTML DOM by creating a hierarchy of abstract states that reflect the actual HTML object hierarchy. Before the analysis begins, an initial heap is constructed that contains the set of the abstract objects corresponding to the HTML code of the page. Park et al. [33] followed a similar approach for modeling HTML DOM. They also provided a more precise model that respects the actual tree hierarchy of the DOM. For example, their model distinguishes whether one DOM node is nested to another or not.

Program Analysis for Asynchronous JavaScript Programs. The majority of static analyses for JavaScript treat asynchronous programs conservatively [19, 23, 21] – they assume that the event loop processes all the asynchronous callbacks in any order – leading to the analysis imprecision. Also, they focus on the client-side applications, where asynchrony mainly appears in DOM events and AJAX calls.

Madsen et al. [27] proposed one of the first static analysis for server-side event-driven programs. Although their approach is able to handle asynchronous I/O operations – unlike our work – they do not provide support for ES6 promises. Additionally, their work introduced a context-sensitivity strategy that tries to imitate the different iterations of the event loop. However, it imposes a large overhead on the analysis; it is able to handle only small programs (less than 400 lines of code). In our work, we propose callback-sensitivity that improves precision without highly sacrificing performance. More recently, Alimadadi et al. [1] presented a dynamic analysis technique for detecting promise-related errors and anti-patterns in JavaScript programs. Specifically, their approach exploits the promise graph; a representation designed for debugging promise-based programs. Beyond promises, our work also handles a broad spectrum of asynchronous features.

Race detection. Zheng et al. [40] presented one of the first race detectors by employing a static analysis for identifying concurrency issues in asynchronous AJAX calls. The aim of their analysis was to detect data races between the code that pre-processes an AJAX request and the callback invoked when the response of the server is received. A subsequent work [34] adopted a dynamic analysis to detect data races in web applications. They first proposed a happens-before relation model to capture the execution order between different operations that are present in a client-side application, such as the loading of HTML elements, execution of scripts, etc. Using this model, their analyses reports data races, by detecting memory conflicts between unordered functions, However, their approach introduced a lot of false positives because most data races did not lead to severe concurrency bugs. Mutlu et al. [31] combined both dynamic and static analysis and primarily focused on detecting data races that have pernicious consequences on the correctness of applications, such as those that affect the browser storage. Initially, they collected the execution traces of an application, and then, they applied a dataflow analysis on those traces to identify data races. Their approach effectively managed to report a very small number of false positives.

6 Conclusions & Future Work

Building upon previous works, we presented the λ_q calculus for modeling 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. We then presented an abstract version of λ_q that over-approximates the semantics of our calculus.

By exploiting the abstract version of λ_q, we designed and implemented what is, to the best of our knowledge, the first static analysis for dealing with a wide range of asynchrony-related features. At the same time, we introduced the concept of callback graph; a directed acyclic graph that represents the temporal relations between the execution of asynchronous callbacks, and we proposed a more precise analysis, i.e., callback-sensitive analysis that respects the execution order of callbacks. We parameterized our analysis with a new context-sensitivity option that is specifically used for asynchronous callbacks.

We then experimented with different parameterizations of our analysis on a set of hand-written and real-world programs. The results revealed that we can analyze medium-sized JavaScript programs. The analysis sensitivity (i.e., both callback- and context-sensitivity) was able to ameliorate the analysis precision without highly sacrificing performance. Specifically, as observed in the real-world modules, our analysis achieved a 79% precision for the callback graph, on average. When we combined callback- and QR-sensitivity, we could further improve the callback graph precision by up to 28.5%, and reduce the total number of type errors by 13.9%.

Our work constitutes a general technique that can be used as a base for further research. Specifically, recent studies showed that concurrency bugs found in JavaScript programs may sometimes be caused by asynchrony [39, 5]. We could leverage our work to design a client analysis on top of it so that it statically detects data races in JavaScript programs. Our callback graph might be an essential element for such an analysis because we could inspect it to identify callbacks whose execution might be non-deterministic, i.e., unconnected nodes in the callback graph.

References

1 Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. Finding Broken Promises in Asynchronous JavaScript Programs. *Proc. ACM Program. Lang.*, 2(OOPSLA):162:1–162:26, 2018. doi:10.1145/3276532.

2 Gogul Balakrishnan and Thomas Reps. Recency-Abstraction for Heap-allocated Storage. In *Proceedings of the 13th International Conference on Static Analysis*, SAS'06, pages 221–239, 2006. doi:10.1007/11823230_15.

3 Patrick Cousot and Radhia Cousot. 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*, POPL '77, pages 238–252, 1977. doi:10.1145/512950.512973.

4 Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 186–189. ACM, 2002.

5 James Davis, Arun Thekumparampil, and Dongyoon Lee. Node.Fz: Fuzzing the server-side event-driven architecture. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 145–160, 2017. doi:10.1145/3064176.3064188.

6 Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported Refactoring for JavaScript. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 119–138, 2011. doi:10.1145/2048066.2048078.

7 Asger Feldthaus and Anders Møller. Semi-automatic Rename Refactoring for JavaScript. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 323–338, 2013. doi:10.1145/2509136.2509520.

8 Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient Construction of Approximate Call Graphs for JavaScript IDE Services. In *Proceedings of*

the 2013 International Conference on Software Engineering, ICSE '13, pages 752–761, 2013. doi:10.1109/ICSE.2013.6606621.

9 Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics engineering with PLT Redex*. Mit Press, 2009.

10 Mattias Felleisen and D. P. Friedman. A Calculus for Assignments in Higher-order Languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 314–, 1987. doi:10.1145/41625.41654.

11 K. Gallaba, Q. Hanam, A. Mesbah, and I. Beschastnikh. Refactoring Asynchrony in JavaScript. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 353–363, 2017. doi:10.1109/ICSME.2017.83.

12 K. Gallaba, A. Mesbah, and I. Beschastnikh. Don't Call Us, We'll Call You: Characterizing Callbacks in Javascript. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10, 2015. doi:10.1109/ESEM.2015.7321196.

13 Philippa Anne Gardner, Sergio Maffeis, and Gareth David Smith. Towards a Program Logic for JavaScript. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 31–44, 2012. doi:10.1145/2103656.2103663.

14 Github. GitHub Octoverse 2017 | Highlights from the last twelve months. https://octoverse.github.com/, 2017. [Online; accessed 08-January-2019].

15 Salvatore Guarnieri and Benjamin Livshits. GATEKEEPER: Mostly static enforcement of security and reliability policies for Javascript code. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 151–168, 2009.

16 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of Javascript. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 126–150, 2010.

17 Ben Hardekopf, Ben Wiedermann, Berkeley Churchill, and Vineeth Kashyap. Widening for Control-Flow. In Kenneth L. McMillan and Xavier Rival, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 472–491. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-642-54013-4_26.

18 Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 59–69, 2011. doi:10.1145/2025113.2025125.

19 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 238–255, 2009. doi:10.1007/978-3-642-03237-0_17.

20 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural Analysis with Lazy Propagation. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis*, pages 320–339. Springer Berlin Heidelberg, 2010.

21 Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: A static analysis platform for JavaScript. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 121–132, 2014. doi:10.1145/2635868.2635904.

22 Y. Ko, H. Lee, J. Dolby, and S. Ryu. Practically Tunable Static Analysis Framework for Large-Scale JavaScript Applications (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 541–551, 2015. doi:10.1109/ASE.2015.28.

23 Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. 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, 2012.

24 Ondřej Lhoták and Laurie Hendren. Evaluating the Benefits of Context-sensitive Points-to Analysis Using a BDD-based Implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):3:1–3:53, 2008. doi:10.1145/1391984.1391987.

25 Matthew C. Loring, Mark Marron, and Daan Leijen. Semantics of Asynchronous JavaScript. In *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages*, DLS 2017, pages 51–62, 2017. doi:10.1145/3133841.3133846.

26 Magnus Madsen, Ondřej Lhoták, and Frank Tip. A Model for Reasoning About JavaScript Promises. *Proc. ACM Program. Lang.*, 1(OOPSLA):86:1–86:24, 2017. doi:10.1145/3133910.

27 Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static Analysis of Event-driven Node.Js JavaScript Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 505–519, 2015. doi:10.1145/2814270.2814272.

28 S. Maffeis and A. Taly. Language-Based Isolation of Untrusted JavaScript. In *2009 22nd IEEE Computer Security Foundations Symposium*, pages 77–91, 2009. doi:10.1109/CSF.2009.11.

29 Sergio Maffeis, John C. Mitchell, and Ankur Taly. An Operational Semantics for JavaScript. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 307–325, 2008. doi:10.1007/978-3-540-89330-1_22.

30 Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005. doi:10.1145/1044834.1044835.

31 Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. Detecting JavaScript Races That Matter. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 381–392, 2015. doi:10.1145/2786805.2786820.

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

33 C. Park, S. Won, J. Jin, and S. Ryu. Static Analysis of JavaScript Web Applications in the Wild via Practical DOM Modeling (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 552–562, 2015. doi:10.1109/ASE.2015.27.

34 Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. Race Detection for Web Applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 251–262, 2012. doi:10.1145/2254064.2254095.

35 J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002. doi:10.1109/LICS.2002.1029817.

36 Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 1–12, 2010. doi:10.1145/1806596.1806598.

37 Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. SYNODE: understanding and automatically preventing injection attacks on Node.Js. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

38 Kwangwon Sun and Sukyoung Ryu. Analysis of JavaScript Programs: Challenges and Research Trends. *ACM Comput. Surv.*, 50(4):59:1–59:34, 2017. doi:10.1145/3106741.

39 J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei. A comprehensive study on real world concurrency bugs in Node.js. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 520–531, 2017. doi:10.1109/ASE.2017.8115663.

40 Yunhui Zheng, Tao Bao, and Xiangyu Zhang. Statically Locating Web Application Bugs Caused by Asynchronous Calls. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 805–814, 2011. doi:10.1145/1963405.1963517.