

# Static Analysis for Asynchronous JavaScript Programs

**Thodoris Sotiropoulos**

Athens University of Economics and Business, Greece  
theosotr@aueb.gr

**Benjamin Livshits**

Imperial College London, UK  
b.livshits@imperial.ac.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

## 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 [13].

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 [35]. JavaScript is a language with a lot of dynamic and metaprogramming features, including but

not limited to prototype-based inheritance, dynamic property lookups, implicit type coercions, dynamic code loading, etc. Many developers often do not understand or do not properly use these features, introducing errors to their programs—which are difficult to debug—or baleful security vulnerabilities. In this context, JavaScript has attracted many engineers and researchers over the past decade to 1) study and reason about its peculiar characteristics, and 2) develop new tools and techniques—such as type analyzers [18, 22, 20], IDE and refactoring tools [5, 6, 7, 10], or bug and vulnerability detectors [28, 14, 33, 30, 4, 36]—to assist developers with the development and maintenance of their applications. Program analysis, and especially static analysis, which automatically computes facts about program’s behavior without actually running it, plays a crucial role in the design of such tools [37].

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 [3]. In this context, a 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 asynchrony-related 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 one of:

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

Promises are particularly useful for asynchronous programming because they provide an intuitive way for creating chains of asynchronous computation, facilitating the enforcement of execution order as well as error propagation [11, 10]. To do that, promises trigger the execution of certain functions (i.e., *callbacks*) depending on their state, e.g., callbacks that are executed once a promise is fulfilled or rejected. For that reason, 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 initial introduction to the language, JavaScript developers have widely embraced promises; a study in 2015 showed that 75% of JavaScript frameworks use promises [11].

Building upon promises, newer versions of ECMAScript have added new language features related to asynchrony. Specifically, in ECMAScript 8, we have the `async/await` keywords. The `async` keyword declares a function as asynchronous which returns a promise fulfilled with its return value, while `await x` defers the execution of the *asynchronous* function in which is placed, 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,

```

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.

the execution is not interrupted until that operation terminates. For instance, a file system 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 which is fulfilled asynchronously once the request succeeds. Then that 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 according to its type and fulfills the promise object allocated by the first invocation of `then()`. 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](https://stackoverflow.com)<sup>1</sup> [26, 25], or the number of bugs reported in open-source repositories [38, 4] indicate. However, existing tools and techniques have limited (and in many cases no) support for asynchronous programs. In particular, existing tools mainly focus on the event system of client-side JavaScript applications [17, 32], and they lack the support of the more recent features added to the language like promises. Also, many previous works conservatively considered that all asynchronous callbacks processed by the *event loop*—the program point which continuously waits for new events to come and is responsible for the scheduling and execution of callbacks—can be called in any order [17, 32, 20]. 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, which does not respect the execution order between the first and the

<sup>1</sup> <https://stackoverflow.com/>

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 those issues, by designing and implementing a static analysis that deals with asynchronous JavaScript programs. For that purpose, we first define a model for understanding and expressing a wide range of JavaScript’s asynchronous constructs, and then we design a static analysis based on that. We propose a new representation, which we call *callback graph*, which 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 which respects the execution order of asynchronous functions. Furthermore, we parameterize our analysis with one novel context-sensitivity strategy designed for asynchronous code. Specifically, we distinguish data flow between asynchronous callbacks based on the promise object on which they have registered or the next computation to which execution proceeds.

**Contributions:** Our work makes the following four contributions:

- We propose a calculus, i.e.,  $\lambda_q$ , for modeling the asynchronous features in the JavaScript language, including timers, promises, and asynchronous I/O operations. Our calculus is a variation of existing calculi [24, 25], and provides constructs and domains specifically targeted for our analysis (§2).
- We design and implement a static analysis that is capable of handling asynchronous JavaScript programs by exploiting the abstract version of  $\lambda_q$ . 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 which illustrates the execution order between asynchronous functions. Building on that, we propose a more precise analysis, (i.e., *callback-sensitive* analysis) which 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 flavor (i.e., *QR-sensitivity*) used for distinguishing asynchronous callbacks. (§3.2, §3.3).
- We evaluate the performance and 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 16,7% as observed in the real-world benchmarks (§4).

## 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  $\lambda_q$ ; an extension of  $\lambda_{js}$  which is the core calculus for JavaScript developed by [15]. The  $\lambda_q$  calculus is designed to be flexible so that it can model various sources of asynchrony found in the language up to the 7th edition

```

⟨v ∈ Val⟩ ::= ...
| ⊥

⟨e ∈ Exp⟩ ::= ...
| newQ()
| e.fulfill(e)
| e.reject(e)
| e.registerFul(e, e, ...)
| e.registerRej(e, e, ...)
| append(e)
| pop()
| •

⟨E⟩ ::= ...
| E.fulfill(e) | v.fulfill(E)
| E.reject(e) | v.reject(E)
| E.registerFul(e, e, ...) | v.registerFul(v, ..., E, e, ...)
| E.registerRej(e, e, ...) | v.registerRej(v, ..., E, e, ...)
| append(E)

```

■ **Figure 2** Syntax of  $\lambda_q$ .

of ECMAScript. (i.e., promises, timers, asynchronous I/O). However, it does not handle the `async/await` keywords and the asynchronous iterators/generators introduced in recent editions of the specification.

## 2.1 The $\lambda_q$ 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 might be fulfilled or rejected with a provided value which is later passed as an argument in the execution of its 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  $\lambda_q$ . For brevity, we present only the new constructs added to the language. Specifically, we add eight new expressions:

- `newQ()`: This expression creates a fresh pending queue object with no callbacks associated with it.
- `e1.fulfill(e2)`: This expression fulfills the receiver (i.e., the expression  $e_1$ ) with the value of  $e_2$ .
- `e1.reject(e2)`: This expression rejects the receiver (i.e., the expression  $e_1$ ) with the value of  $e_2$ .
- `e1.registerFul(e2, e3, ...)`: 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.

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

■ **Figure 3** Concrete domains of  $\lambda_q$ .

This expression can also receive optional parameters (i.e., expressed through “...”) with which  $e_2$  is called if the queue object is fulfilled with  $\perp$  value. We will justify later the intuition behind that.

- $e_1.\text{registerRej}(e_2, e_3, \dots)$ : The same as  $e.\text{registerFul}(\dots)$  but this time the given callback is executed once the receiver is rejected.
- $\text{append}(e)$ : This expression appends the queue object  $e$  to the top of the current queue chain. As we shall see later, the top element of a queue chain corresponds to the queue object that is needed to be rejected if the execution encounters an uncaught exception.
- $\text{pop}()$ : 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 [8, 15, 26, 24, 25] to express how the evaluation of an expression proceeds. The symbol  $E$  denotes which sub-expression is currently being evaluated. For instance,  $E.\text{fulfill}(e)$  describes that we evaluate the receiver of  $\text{fulfill}$ , whereas  $v.\text{fulfill}(E)$  implies that the receiver has been evaluated to a value  $v$ , and the evaluation now lies on the argument of  $\text{fulfill}$ . Beyond those expressions, the  $\lambda_q$  calculus introduces a new value, that is,  $\perp$ . This value differs from `null` and `undefined` because it expresses the absence of value and it does not correspond to any JavaScript value.

Figure 3 presents the domains introduced in the semantics of  $\lambda_q$ . 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 2.2 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 an address which corresponds to another queue object which is dependent on the current, i.e., it is settled whenever the current queue object is settled, and with the same state. We create such dependencies when we settle a queue object with another queue object. In this case, the receiver is dependent on the queue object used as 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 which the return value of the function is going to fulfill. In the list of callbacks  $\kappa \in \text{ScheduledCallbacks}$ , we keep the order in which callbacks are scheduled. Note that we maintain one more list of callbacks (i.e.,  $\tau \in \text{ScheduledTimerIO}$ )

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  $\phi \in QueueChain$  is a sequence of addresses. In a queue chain, we store the queue object that we reject, if there is an uncaught exception in the current execution. Specifically, when we encounter an uncaught exception, we inspect the top element of the queue chain, and we reject it. If the queue chain is empty, we propagate the exception to the call stack as 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  $\lambda_q$  which is an adaptation of [24, 25]. 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  $\pi$ , a queue chain  $\phi$ , two sequences of callbacks  $\kappa$  and  $\tau$ , and an expression  $e$  in the evaluation context  $E$  lead to a new queue  $\pi'$ , a new queue chain  $\phi'$ , two new sequences of callbacks  $\kappa'$  and  $\tau'$ , and a new expression  $e'$  in the same evaluation context  $E$ , assuming that the expression  $e$  is reduced to  $e'$  (i.e.,  $e \mapsto e'$ ). 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  $v \neq \perp$  and does *not* correspond to any queue object. In particular, we first change the state of the receiver object from “pending” to “fulfilled”. In turn, we update the already registered callbacks (if any) by setting the value  $v$  as the only argument of them. Then, we add the updated callbacks to the list of scheduled callbacks  $\kappa$  (assuming that the receiver is neither  $l_{time}$  nor  $l_{io}$ ). Also, observe that the initial expression is reduced to  $l.fulfill(v)$ , that is, if there is a dependent queue object  $l$ , we also fulfill that queue object with the same value  $v$ .

The [fulfill-pend-pend] describes the scenario of fulfilling a pending queue object  $p$  with another pending queue object  $v$ . In this case, we update the queue  $\pi$  by making the queue object  $p$  to be dependent on  $v$ . This means that we settle  $p$  whenever we settle  $v$  and in the same way. 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 value with which  $v$  is fulfilled. This is expressed by the resulting expression  $p.fulfill(v')$ .

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

The [fulfill-settled] rule illustrates the case when we try to fulfill a settled queue object. Notice that this rule neither affects the queue  $\pi$  nor the lists of scheduled callbacks  $\kappa$  and  $\tau$ .

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

$$\begin{array}{c}
\frac{e \hookrightarrow e'}{\pi, \phi, \kappa, \tau, E[e] \rightarrow \pi', \phi', \kappa', \tau', E[e']} \quad [\text{e-context}] \\
\\
\frac{\text{fresh } \alpha \quad \pi' = \pi[\alpha \mapsto (\text{pending}, [], [], \perp)]}{\pi, \phi, \kappa, \tau, E[\text{newQ}()] \rightarrow \pi', \phi, \kappa, \tau, E[\alpha]} \quad [\text{newQ}] \\
\\
\frac{v \neq \perp \quad (\text{pending}, t, k, l) = \pi(p) \quad v \notin \text{dom}(\pi) \quad t' = \langle (\alpha, f, [v], r) \mid (\alpha, f, a, r) \in t \rangle}{\kappa' = \kappa :: t' \quad \chi = (\text{fulfilled}, v) \quad \pi' = \pi[p \mapsto (\chi, [], [], l)] \quad p \neq l_{\text{time}} \wedge p \neq l_{\text{io}}} \quad [\text{fulfill-pending}] \\
\pi, \phi, \kappa, \tau, E[p.\text{fulfill}(v)] \rightarrow \pi', \phi, \kappa', \tau, E[l.\text{fulfill}(v)] \\
\\
\frac{(\text{pending}, t, k, l) = \pi(p) \quad v \in \text{dom}(\pi)}{p(v) = (\text{pending}, t', k', \perp) \quad \pi' = \pi[v \mapsto (\text{pending}, t', k', p)]} \quad [\text{fulfill-pend-pend}] \\
\pi, \phi, \kappa, \tau, E[p.\text{fulfill}(v)] \rightarrow \pi', \phi, \kappa, \tau, E[\text{undef}] \\
\\
\frac{(\text{pending}, t, k, l) = \pi(p) \quad v \in \text{dom}(\pi) \quad p(v) = ((\text{fulfilled}, v'), t', k', m)}{\pi, \phi, \kappa, \tau, E[p.\text{fulfill}(v)] \rightarrow \pi, \phi, \kappa, \tau, E[p.\text{fulfill}(v')]} \quad [\text{fulfill-pend-ful}] \\
\\
\frac{v = \perp \quad (\text{pending}, t, k, l) = \pi(p) \quad \kappa' = \kappa :: t}{\chi = (\text{fulfilled}, v) \quad \pi' = \pi[p \mapsto (\chi, [], [], l)]} \quad [\text{fulfill-pending-}\perp] \\
\pi, \phi, \kappa, \tau, E[p.\text{fulfill}(v)] \rightarrow \pi', \phi, \kappa', \tau, E[l.\text{fulfill}(v)] \\
\\
\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}]} \quad [\text{fulfill-settled}] \\
\\
\frac{(\text{pending}, t, k, l) = \pi(p) \quad t' = t \cdot (p', f, [n_1, n_2, \dots, n_n], r)}{\pi' = \pi[p \mapsto (\text{pending}, t', k, l)]} \quad [\text{registerFul-pending}] \\
\pi, \phi, \kappa, \tau, E[p.\text{registerFul}(f, p', r, n_1, n_2, \dots, n_n)] \rightarrow \pi', \phi, \kappa, \tau, E[\text{undef}] \\
\\
\frac{p \neq l_{\text{time}} \wedge p \neq l_{\text{io}} \quad (s, t, k, \chi) = \pi(p) \quad s \downarrow_1 = \text{fulfilled}}{s \downarrow_2 \neq \perp \quad \kappa' = \kappa \cdot (p', f, [s \downarrow_2], r)} \quad [\text{registerFul-fulfilled}] \\
\pi, \phi, \kappa, \tau, E[p.\text{registerFul}(f, p', r, n_1, n_2, \dots, n_n)] \rightarrow \pi', \phi, \kappa', \tau, E[\text{undef}] \\
\\
\frac{p \neq l_{\text{time}} \wedge p \neq l_{\text{io}} \quad (s, t, k, \chi) = \pi(p) \quad s \downarrow_1 = \text{fulfilled}}{s \downarrow_2 = \perp \quad \kappa' = \kappa \cdot (p', f, [n_1, n_2, \dots, n_n], r)} \quad [\text{registerFul-fulfilled-}\perp] \\
\pi, \phi, \kappa, \tau, E[p.\text{registerFul}(f, p', r, n_1, n_2, \dots, n_n)] \rightarrow \pi', \phi, \kappa', \tau, E[\text{undef}] \\
\\
\frac{p = l_{\text{time}} \vee p = l_{\text{io}} \quad (s, t, k, \chi) = \pi(p) \quad s \downarrow_1 = \text{fulfilled}}{s \downarrow_2 = \perp \quad \tau' = \tau \cdot (p', f, [n_1, n_2, \dots, n_n], r)} \quad [\text{registerFul-timer-io-}\perp] \\
\pi, \phi, \kappa, \tau, E[p.\text{registerFul}(f, p', r, n_1, n_2, \dots, n_n)] \rightarrow \pi', \phi, \kappa, \tau', E[\text{undef}] \\
\\
\frac{p \in \text{dom}(\pi) \quad \phi' = p \cdot \phi}{\pi, \phi, \kappa, \tau, E[\text{append}(p)] \rightarrow \pi, \phi', \kappa, \tau, E[\text{undef}]} \quad [\text{append}] \\
\\
\frac{}{\pi, p \cdot \phi, \kappa, \tau, E[\text{pop}()] \rightarrow \pi, \phi, \kappa, \tau, E[\text{undef}]} \quad [\text{pop}] \\
\\
\frac{\phi = p \cdot \phi'}{\pi, \phi, \kappa, \tau, E[\text{err } v] \rightarrow \pi, \phi', \kappa, \tau, E[p.\text{reject}(v)]} \quad [\text{error}]
\end{array}$$

■ **Figure 4** The semantics of  $\lambda_q$

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

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

■ **Figure 5** The semantics of the event loop.

this callback with the queue object  $p'$  given as the second argument. That queue object  $p'$  is fulfilled upon the termination of  $f$ . Also, this rule adds any extra arguments passed in `registerFul` as arguments of  $f$ .

The `[registerFul-fulfilled]` rule adds the given callback  $f$  to the list  $\kappa$  (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 sets those arguments as 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.

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 at  $l_{io}$ . Thus, we add the given callback  $f$  to the list  $\tau$  instead of  $\kappa$ .

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.,  $p \in \text{dom}(\pi)$ ). 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  $\tau$  for holding callbacks coming from the  $l_{time}$  or  $l_{io}$  queue objects, and the list  $\kappa$  for callbacks of any other queue objects. The intuition behind this design choice is that it is convenient for us to 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 [24, 31]. For example, the event loop processes a callback of a promise object before any callback of a timer or an asynchronous I/O operation 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  $\kappa$ . Then, we get the queue object specified in that callback, and we attach it to the front of the queue chain. Adding the queue object  $q$  to the top of the queue chain allows us to reject that queue object if there is an uncaught

## XX:10 Static Analysis for Asynchronous JavaScript Programs

$$\begin{aligned}
 \langle e \in \text{Exp} \rangle ::= & \dots \\
 & | \text{addTimerCallback}(e_1, e_2, e_3, \dots) \\
 & | \text{addIOCallback}(e_1, e_2, e_3, \dots) \\
 \langle E \rangle ::= & \dots \\
 & | \text{addTimerCallbackCallback}(E, e, \dots) \mid \text{addTimerCallback}(v, \dots, E, e, \dots) \\
 & | \text{addIOCallback}(E, e, \dots) \mid \text{addIOCallback}(v, \dots, E, e, \dots)
 \end{aligned}$$

■ **Figure 6** Extending the syntax of  $\lambda_q$  to deal with timers and asynchronous I/O.

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 `[fulfill-settled]` rule). Also, observe how the event loop is reduced, i.e.,  $q.\text{fulfill}(f(a)); \text{pop}(); \bullet$ . Specifically, once we execute the callback  $f$  and fulfill the dependent queue object  $q$  with the return value of that callback, we evaluate `pop()`, that is, we pop the top element of the queue chain before we re-evaluate 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  $\kappa$  is empty. In other words, that rule states that if there are not any callbacks, which neither come from the  $l_{time}$  nor the  $l_{io}$  queue object, inspect the list  $\tau$ , 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 [24]. Overall, that rule describes the scheduling policy presented in the work of [24], where initially we look for any callbacks of promises, and if there exist, we select one of those. Otherwise, we choose any callback associated with timers and asynchronous I/O at random.

### 2.1.4 Modeling Timers & Asynchronous I/O

$$\frac{q = \pi(l_{time})}{\pi, \phi, \kappa, \tau, \text{addTimerCallback}(f, n_1, \dots) \rightarrow \pi, \phi, \kappa, q.\text{registerFul}(f, q, n_1, \dots)} \text{ [add-timer-callback]}$$

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

■ **Figure 7** Extending the semantics of  $\lambda_q$  to deal with timers and asynchronous I/O.

To model timers and asynchronous I/O, we follow a similar approach to the work of [24]. Specifically, we start with an initial queue  $\pi$ , which contains two queue objects: the  $q_{time}$ , and  $q_{io}$  which are located at the  $l_{time}$  and  $l_{io}$  respectively. Both  $q_{time}$  and  $q_{io}$  are initialized as  $((\text{fulfilled}, \perp), [], [], \perp)$ . Besides that, we extend the syntax of  $\lambda_q$  by adding two more expressions. Figure 6 shows the extended syntax of  $\lambda_q$  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 `addTimerCallback(...)` construct adds the callback  $e_1$  to the queue object located at the address  $l_{time}$ . We call the provided callback The arguments of that callback are any optional parameters passed in `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  $q.\text{registerFul}(f, q, n_1, \dots)$ . In particular, we add a new callback  $f$  to the

queue object found at  $l_{time}$ . Observe that we pass the same queue object (i.e.,  $q$ ) as the second argument of `registerFul`. That means that the execution of  $f$  does not affect any queue object since  $q$  is already settled. Recall that according to the `[fulfill-settled]` rule (Figure 4), trying to fulfill (and similarly to reject) a settled queue object does not have any effect. Beyond that, since  $q$  is fulfilled with  $\perp$ , the extra arguments (i.e.,  $n_1, \dots$ ) are also passed as arguments in the invocation of  $f$ .

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

## 2.2 Expressing Promises in Terms of $\lambda_q$

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

```

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

■ **Figure 8** Expressing `Promise.resolve` in terms of  $\lambda_q$

**Example—Modeling `Promise.resolve()`:** In Figure 8, we see how we model the `Promise.resolve()` function in terms of  $\lambda_q$ . 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 which 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  $\lambda_q$  can model this source of asynchrony (lines 4–6). 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. That 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  $\lambda_q$  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  $\lambda_q$ . Currently, there are few implementations available for asynchronous JavaScript, and previous efforts mainly

focus on modeling the event system of client-side applications [17, 32]. 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

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

■ **Figure 9** The abstract domains of  $\lambda_q$ .

Figure 9 presents the abstract domains of the  $\lambda_q$  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 which keeps the domain finite [18, 26]. Also note 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  $\lambda_q$  and allows us to capture all possible program 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  $\lambda_q$ —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  $\lambda_q$ . 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. [26] made similar observations for the event-driven programs.

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

*Abstract Callback:* An abstract callback comprises two abstract addresses, one function, and a list of values which stands for the arguments of the function. Note that the first address

corresponds to `this` object, while the second one is the queue object which the return value of the function fulfills.

*Abstract List of Scheduled Callbacks:* We use a list of sets to abstract the domain responsible for maintaining the callbacks which 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, and therefore, we have to add it to the lists  $\kappa$  or  $\tau$  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 in 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  $\lambda_q$ , 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 that 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 size of the abstract queue chain is greater than 1 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 callbacks of promises is straightforward because it is a direct translation of the corresponding semantics of  $\lambda_q$ . In particular, there are two possible cases:

- *Settle a promise which has registered callbacks:* When we settle (i.e., either fulfill or reject) a promise object which has registered callbacks, we schedule those callbacks associated with the next state of the promise by putting them on the tail of the list  $\kappa$ . For instance, if we fulfill a promise, we append all the callbacks triggered on fulfillment on the list  $\kappa$ . A reader might observe that if 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  $\kappa$ . The reason for this is that an abstract queue object does not keep the registration order of callbacks.
- *Register a callback on an already settled promise:* When we encounter a statement of the form `x.then(f1, f2)`, where `x` is a settled promise, we schedule either callback `f1` or `f2` (i.e., we add it to the list  $\kappa$ ) depending on the state of that promise, i.e., we schedule callback `f1` if `x` is fulfilled and `f2` if it is rejected.

**Timers & Asynchronous I/O.** A static analysis is not able to reason about the external environment. For instance, it cannot decide when an operation on a file system or a request to a server is complete. Similarly, it is not able to deal with time. For that purpose, we adopt a conservative approach for tracking the execution order between callbacks related to timers and asynchronous I/O. In particular, we assume that the execution order between those callbacks is unspecified; 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:

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

We define every node of a callback graph as  $n \in \text{Node} = C \times F$ , 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  $cg \in \text{CallbackGraph}$  has the following meaning: *the function  $f_2$  in context  $c_2$  is executed immediately after the function  $f_1$  in context  $c_1$* . We can treat the above statement as the following expression:  $f_1(); f_2();$

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

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

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

$$\begin{aligned} n_1 \rightarrow_{cg} n_2 &\Rightarrow n_1 \rightarrow_{cg}^* n_2 \\ n_1 \rightarrow_{cg}^* n_2 \wedge n_2 \rightarrow_{cg}^* n_3 &\Rightarrow n_1 \rightarrow_{cg}^* n_3, \quad \text{where } n_3 \in \text{Node} \end{aligned}$$

Definition 1 and Definition 2 introduce the concept of a *path* between two nodes in a callback graph  $cg \in \text{CallbackGraph}$ . 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 significant property of a callback graph is that it does *not* have any cycles, i.e.,

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

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.

### 3.3 Analysis Sensitivity

#### 3.3.1 Callback Sensitivity

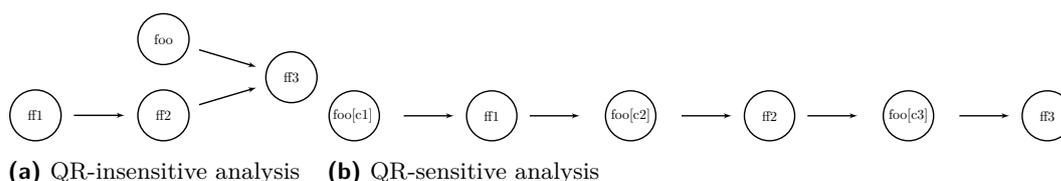
Knowing the temporal relations between asynchronous callbacks enables us to capture how data flow is propagated precisely. Typically, a naive flow-sensitive analysis, which exploits

```

1  function foo() { ... }
2
3  var x = Promise.resolve()
4    .then(foo)
5    .then(function ff1() { ... })
6    .then(foo)
7    .then(function ff2() { ... })
8    .then(foo)
9    .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.



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

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  $cg \in \text{CallbackGraph}$ . Specifically, when the analysis needs to propagate the resulting state from the exit point of a callback  $x$ , instead of propagating that state to the caller (note that the caller of a callback is the event loop), it propagates it to the entry points of the next callbacks, i.e., all callback nodes  $y \in \text{Node}$  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. Therefore, 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  $\mathcal{P}(\text{Node} \times \text{Node})$ , where  $n \in \text{Node} = C \times F$ . It is possible to increase the precision of a callback graph (and therefore the precision of the analysis) 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 [29, 23], 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 [18, 20] creates a context with regards to the arguments of a function. Such a strategy might not be potent 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 to which they are added (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 the queue object to which we add a callback, and  $l_r$  is the abstract address of the queue object which the return value of a callback fulfills. Notice that this domain is finite; thus, the analysis always terminates.

**Example:** 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 the 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()`  $\rightarrow$  `ff1()`  $\rightarrow$  `foo()`  $\rightarrow$  `ff2()`  $\rightarrow$  `foo()`  $\rightarrow$  `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 also 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 extends *TAJS* [18, 19, 17]; a state-of-the-art static analyzer for JavaScript. *TAJS* analysis is implemented as an instance of the abstract interpretation framework [2], and it is designed to be sound. It uses a lattice specifically designed for JavaScript which 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 [18].

```

1  function open(filename, flags, mode, callback) {
2      TAJJS_makeContextSensitive(open, 3);
3      var err = TAJJS_join(TAJJS_make("Undef"), TAJJS_makeGenericError());
4      var fd = TAJJS_join(TAJJS_make("Undef"), TAJJS_make("AnyNum"));
5      TAJJS_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 `TAJJS_` 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, `TAJJS_make("AnyStr")` evaluates to a value that can be any string.

Prior to our extensions, TAJJS 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 TAJJS. Then, we provide models for promises written in Java by faithfully following the ECMAScript specification. Recall again that our models exploit the  $\lambda_q$  calculus presented in Section 2 and they produce side-effects that over-approximate the behavior of JavaScript promises. Beyond that, we implement models for the special constructs of  $\lambda_q$ , i.e., `addTimerCallback`, `addIOCallback` (Recall Section 2.1.4), which 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 function `open()` of the `Node.js` module `fs`. 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` which is not `undefined` if there is an error during I/O, 2) `fd` which is an integer indicating the file descriptor of the opened file. Note that `fd` is `undefined`, if any error occurs. Our model first makes `open()` parameter-sensitive on the third argument which corresponds to the callback provided by the programmer. Then, at lines 3 and 4, it initializes the arguments of the callback, i.e., `err` and `fd` respectively. 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 `TAJJS_addAsyncIOCallback()`, which registers the given callback on the queue object responsible for I/O operations, implementing the semantics of the `addIOCallback` primitive from our  $\lambda_q$  calculus.

## 4 Empirical 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

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

■ **Table 1** List of 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.).

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<sup>2</sup>, and V8 engine<sup>3</sup>[18, 20, 21]. However, those benchmarks are not asynchronous; thus, they are not suitable for evaluating our analysis. 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<sup>4</sup>. The second option employed the Github API<sup>5</sup> to find JavaScript repositories which are related to promises. We then investigated the Github repositories which we collected at the first phase by computing various metrics such as lines of code, number of promise-, timer- and asynchronous IO-related statements. We manually selected the 6 JavaScript modules presented in Table 1. Most of them are libraries for performing HTTP requests or file system operations.

We experiment with 4 different analyses: 1) an analysis which 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). 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. 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 [20]. The fewer type errors an analysis reports, the more precise it is. The same applies to the number of callbacks inspected by the analysis; fewer callbacks indicate a more accurate analysis. To compute the performance characteristics of every analysis, we re-run every experiment ten times in order to receive reliable measurements. All the experiments were run on a machine with an Intel i7 2.4GHz quad-core processor and 8GB of RAM.

## 4.2 Results

**Micro-benchmarks.** Table 2 shows how precise every analysis is on every micro-benchmark.

<sup>2</sup> <https://browserbench.org/JetStream/>

<sup>3</sup> <http://www.netchain.com/Tools/v8/>

<sup>4</sup> <https://www.npmjs.com/browse/depended>

<sup>5</sup> <https://developer.github.com/v3/>

Benchmark	Analyzed Callbacks				Callback Graph Precision				Type Errors			
	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

■ **Table 2** Precision on micro-benchmarks.

Starting with callback-insensitive analyses (i.e., columns NC-No and NC-QR), 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 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 compared to the QR-sensitive analyses which 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

Benchmark	Analyzed Callbacks				Callback Graph Precision				Type Errors			
	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	6	6
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	31	30

■ **Table 3** Precision on macro-benchmarks.

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

■ **Table 4** Times of different analyses in seconds.

number of the analyzed callbacks from 169 to 158. Notice that if callback-sensitivity 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 of 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 16.7% fewer type errors in total. Additionally, if 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.

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. Neither callback- nor QR-sensitivity is effective for that kind of benchmarks, 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. 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). Therefore, keeping a more precise state does not lead to a more precise callback graph.

Table 4 gives the running times of every analysis on macro-benchmarks. We notice that in some benchmarks (such as `fetch`) a more precise analysis may decrease 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 which suggests that precision might lead to a faster analysis in some cases [32]. On the other hand, we observe a non-trivial decrease in the analysis performance in only benchmark. Specifically, the analysis sensitivity increased the running times of `pixiv-client` by 12%–30.6%. However, such an increase seems to be

```

1  function consumed(body) {
2      if (body.bodyUsed) {
3          return Promise.reject(new
4              TypeError("Already read"));
5      }
6      body.bodyUsed = true;
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 (
16             Blob.prototype.isPrototypeOf(
17                 body)) {
18             this._bodyBlob = body;
19         }
20         ...
21     }
22     this.text = function text() {
23         var rejected = consumed(this);
24         if (rejected) {
25             return rejected;
26         }
27         if (this._bodyBlob) {
28             return readBlobAsText(
29                 this._bodyBlob);
30         } else if (this._bodyArrayBuffer
31             ) {
32             return Promise.resolve(
33                 readArrayBufferAsText(
34                     this._bodyArrayBuffer))
35         }
36     };
37     ...
38     this.formData = function formData() {
39         return this.text().then(decode);
40     }
41     ...
42     function Response(body) {
43         ...
44         this._bodyInit(body);
45     }
46     Body.call(Response.prototype);
47     ...
48     function fetch(input, init) {
49         return new Promise(function (resolve,
50             reject) {
51             ...
52             var xhr = new XMLHttpRequest();
53             xhr.onload = function onLoad() {
54                 ...
55                 resolve(new Response(xhr.response)
56                     );
57             };
58         });
59     }
60 }

```

■ **Figure 13** Code fragment taken from `fetch`.

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`<sup>6</sup>. 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). If 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 fulfilled promise depending on the type of the body of the provided 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<sup>7</sup> through the call of the function `decode()`. The function `fetch()` (lines 47–56) makes a new asynchronous request. When the request completes successfully, the callback `onLoad()` is executed asynchronously (line 51).

<sup>6</sup> <https://github.com/github/fetch>

<sup>7</sup> <https://developer.mozilla.org/en-US/docs/Web/API/FormData>

## ■ 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.

That callback finally fulfills the promise returned by `fetch()` with a new response object initialized with 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()`, which 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 which 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 `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 which has a string type (line 2). 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 5 of Listing 2, we allocate a fresh response object whose body is an instance of `Blob`<sup>8</sup>. 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

<sup>8</sup> <https://developer.mozilla.org/en-US/docs/Web/API/Blob>

by the function `readBlobAsText()` (Figure 13, line 26). A QR-sensitive analysis—which creates a context according to the queue object on which a callback is registered—is capable of separating the two invocations of `decode()` because the first call of `decode()` is registered on the promise object which comes from line 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 another. 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 which 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 an existing analyzer, i.e., TAJIS. Therefore, the precision and performance of TAJIS 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 `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 which stem from the interaction with those APIs, leading to unsoundness [14, 32].
- Our macro-benchmarks consist of JavaScript libraries. Therefore, we needed to write some test cases that invoke the API functions of those benchmarks. We provided both hand-written test cases and test cases or examples taken from their documentation, trying to test the main APIs that exercise asynchrony in JavaScript.

## 5 Related Work

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

**Semantics.** Maffeis et al. [27] 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. [15] expressed the semantics of the 3rd edition of ECMAScript through a different approach; they developed a lambda calculus called  $\lambda_{JS}$ , and provided a desugaring mechanism for converting JavaScript code into  $\lambda_{JS}$ . We used  $\lambda_{JS}$  as a base for modeling asynchronous JavaScript. Later, Gardner et al. [12] introduced a program logic for reasoning about client-side JavaScript programs which support ECMAScript 3. They presented big-step operational semantics on the basis of that proposed by [27], and they introduced inference rules for program reasoning which are highly inspired from separation

logic [34]. More recently, Madsen et al. [25] and Loring et al. [24] extended  $\lambda_{JS}$  for modeling promises and asynchronous JavaScript respectively. Our model is a variation of their works; our modifications enable us to model almost all the sources of asynchrony found in JavaScript—some of them are not handled by their models.

**Static Analysis for JavaScript.** Guarnieri et al. [14] proposed one of the first pointer analyses 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 works that managed to model some of the peculiar features of JavaScript such as prototype-based inheritance. TAJs [18, 19, 17] is a typer analyzer for JavaScript which is implemented as a classical dataflow analysis. Our work is implemented as an extension of TAJs. SAFE [22] is a static analysis framework, which 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 which is plugged after the construction of CFG. This analysis adopts a similar approach with that of TAJs, i.e., a flow- and context-sensitive analysis which operates on top of CFG. JSAI [20] implements an analysis through the abstract interpretation framework [2]. 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 [9], 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 widening operator used in the fix-point calculation [16].

Existing static analyses provide sufficient support for precisely modeling browser environment. Jensen et al. [17] modeled HTML DOM by creating a hierarchy of abstract states which reflect the actual HTML object hierarchy. Before the analysis begins, an initial heap is constructed which contains the set of the abstract objects corresponding to the HTML code of the page. Park et al. [32] followed a similar approach for modeling HTML DOM. They also provided a more precise model which 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 [18, 22, 20]—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. [26] 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 which 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 which 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. [39] 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 which pre-processes an AJAX

request and the callback invoked when the response of the server is received. A subsequent work [33] 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 functions, where there is not any happens-before relation to each other. However, their approach introduced a lot of false positives because most data races did not lead to severe concurrency bugs. Mutlu et al. [30] 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 which 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  $\lambda_q$  calculus for modeling asynchrony in JavaScript. Our calculus  $\lambda_q$  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  $\lambda_q$  which over-approximates the semantics of our calculus.

By exploiting that abstract version, we designed and implemented what is, to the best of our knowledge, the first static analysis for dealing with a wide range of asynchrony-related features. At the same time, we introduced the concept of callback graph; a directed acyclic graph which represents the temporal relations between the execution of asynchronous callbacks, 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 flavor 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 using our approach. The analysis sensitivity (i.e., both callback- and context-sensitivity) is able to ameliorate the analysis precision without highly sacrificing performance. Specifically, as observed in the real-world modules, our analysis achieves a 79% precision for the callback graph, on average. If we combine callback- and QR-sensitivity, we can further improve the callback graph precision by up to 28.5%. Also, the callback- and QR-sensitive analysis achieves a 88% callback graph precision on average, and reduces the total number of type errors by 16.7%.

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 [38, 4]. 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 can inspect it to identify callbacks whose execution might be non-deterministic, i.e., unconnected nodes in the callback graph.

---

### References

- 1 S. Alimadadi, D. Zhong, M. Madsen, and F. Tip. Finding broken promises in asynchronous JavaScript programs. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):162, 2018.

- 2 P. Cousot and R. 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*, pages 238–252. ACM, 1977.
- 3 F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 186–189. ACM, 2002.
- 4 J. Davis, A. Thekumparampil, and D. Lee. Node.fz: Fuzzing the server-side event-driven architecture. EuroSys '17, pages 145–160. ACM, Apr 23, 2017.
- 5 A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip. Tool-supported refactoring for JavaScript. pages 119–137, 2011. cited By 17.
- 6 A. Feldthaus and A. Møller. Semi-automatic rename refactoring for JavaScript. pages 323–337, 2013. cited By 11.
- 7 A. Feldthaus, M. Schafer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for JavaScript IDE services. pages 752–761, 2013. cited By 38.
- 8 M. Felleisen, R. B. Findler, and M. Flatt. *Semantics engineering with PLT Redex*. MIT Press, 2009.
- 9 M. Felleisen and D. Friedman. A calculus for assignments in higher-order languages. volume Part F130236, pages 314–325, 1987. cited By 36.
- 10 K. Gallaba, Q. Hanam, A. Mesbah, and I. Beschastnikh. Refactoring asynchrony in JavaScript. pages 353–363, 2017.
- 11 K. Gallaba, A. Mesbah, and I. Beschastnikh. Don't Call Us, We'll Call You: Characterizing Callbacks in Javascript. volume 2015-November, pages 247–256, 2015. cited By 14.
- 12 P. Gardner, S. Maffeis, and G. Smith. Towards a Program Logic for JavaScript. In J. Field and M. Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 31–44. ACM, Jan. 2012.
- 13 Github. GitHub Octoverse 2017 | Highlights from the last twelve months. <https://octoverse.github.com/>, 2017. [Online; accessed 08-January-2019].
- 14 S. Guarnieri and V. B. Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *USENIX Security Symposium*, volume 10, pages 78–85, 2009.
- 15 A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of javascript. In T. D'Hondt, editor, *ECOOP 2010 – Object-Oriented Programming*, pages 126–150, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- 16 B. Hardekopf, B. Wiedermann, B. Churchill, and V. Kashyap. Widening for control-flow. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8318 LNCS:472–491, 2014. cited By 2.
- 17 S. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript Web Applications. pages 59–69, 2011. cited By 47.
- 18 S. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5673 LNCS:238–255, 2009. cited By 85.
- 19 S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural analysis with lazy propagation. In *Proc. 17th International Static Analysis Symposium (SAS)*, volume 6337 of LNCS. Springer-Verlag, September 2010.
- 20 V. Kashyap, K. Dewey, E. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. JSAT: A static analysis platform for JavaScript. volume 16-21-November-2014, pages 121–132, 2014. cited By 37.
- 21 Y. Ko, H. Lee, J. Dolby, and S. Ryu. Practically tunable static analysis framework for large-scale JavaScript applications. pages 541–551, 2016. cited By 10.

- 22 H. Lee, S. Won, J. Jin, J. Cho, and S. 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. Citeseer, 2012.
- 23 O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Transactions on Software Engineering and Methodology*, 18(1), 2008. cited By 57.
- 24 M. C. Loring, M. Marron, and D. Leijen. Semantics of asynchronous JavaScript. *DLS 2017*, pages 51–62. ACM, Oct 24, 2017.
- 25 M. Madsen, O. Lhoták, and F. Tip. A model for reasoning about JavaScript promises. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–24, Oct 12, 2017.
- 26 M. Madsen, F. Tip, and O. Lhoták. Static analysis of event-driven Node.js JavaScript applications. *ACM SIGPLAN Notices*, 50(10):505–519, Oct 23, 2015.
- 27 S. Maffeis, J. Mitchell, and A. Taly. An operational semantics for JavaScript. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5356 LNCS:307–325, 2008. cited By 48.
- 28 S. Maffeis and A. Taly. Language-based isolation of untrusted JavaScript. pages 77–91, 2009. cited By 47.
- 29 A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. pages 1–11, 2002. cited By 104.
- 30 E. Mutlu, S. Tasiran, and B. Livshits. Detecting JavaScript races that matter. pages 381–392, 2015. cited By 6.
- 31 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].
- 32 C. Park, S. Won, J. Jin, and S. Ryu. Static analysis of JavaScript web applications in the wild via practical DOM modeling. pages 552–562, 2016. cited By 12.
- 33 B. Petrov, M. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. pages 251–261, 2012. cited By 24.
- 34 J. Reynolds. Separation logic: A logic for shared mutable data structures. pages 55–74, 2002. cited By 1083.
- 35 G. Richards, S. Lebresne, B. Burg, and J. Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. *ACM SIGPLAN NOTICES*, 45(6):1–12, JUN 2010. ACM SIGPLAN Conference on Programming Language Design and Implementation, Toronto, CANADA, JUN 05-10, 2010.
- 36 C.-A. Staicu, M. Pradel, and B. Livshits. Synode: Understanding and Automatically Preventing Injection Attacks on Node.js. In *Network and Distributed System Security (NDSS)*, 2018.
- 37 K. Sun and S. Ryu. Analysis of JavaScript programs: Challenges and research trends. *ACM Computing Surveys*, 50(4), 2017. cited By 0.
- 38 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. pages 520–531, 2017. cited By 2.
- 39 Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. *WWW ’11*, pages 805–814. ACM, Mar 28, 2011.