A Model for Reasoning About JavaScript Promises

MAGNUS MADSEN, University of Waterloo, Canada
ONDŘEJ LHOTÁK, University of Waterloo, Canada
FRANK TIP, Northeastern University, USA

In JavaScript programs, asynchrony arises in situations such as web-based user-interfaces, communicating with servers through HTTP requests, and non-blocking I/O. Event-based programming is the most popular approach for managing asynchrony, but suffers from problems such as lost events and event races, and results in code that is hard to understand and debug. Recently, ECMAScript 6 has added support for *promises*, an alternative mechanism for managing asynchrony that enables programmers to chain asynchronous computations while supporting proper error handling. However, promises are complex and error-prone in their own right, so programmers would benefit from techniques that can reason about the correctness of promise-based code.

Since the ECMAScript 6 specification is informal and intended for implementers of JavaScript engines, it does not provide a suitable basis for formal reasoning. This paper presents λ_p, a core calculus that captures the essence of ECMAScript 6 promises. Based on λ_p, we introduce the *promise graph*, a program representation that can assist programmers with debugging of promise-based code. We then report on a case study in which we investigate how the promise graph can be helpful for debugging errors related to promises in code fragments posted to the StackOverflow website.

CCS Concepts: • **Theory of computation → Operational semantics**; **Program reasoning**; • **Software and its engineering → Object oriented languages**;

Additional Key Words and Phrases: EcmaScript 6, Promises, JavaScript, Formal Semantics, Promise Graph

ACM Reference Format:
Magnus Madsen, Ondřej Lhoták, and Frank Tip. 2017. A Model for Reasoning About JavaScript Promises. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 86 (October 2017), 24 pages. https://doi.org/10.1145/3133910

1 INTRODUCTION

Asynchronous control flow is widely used in the JavaScript community for a variety of tasks such as implementing web-based user-interfaces, communicating with servers through HTTP requests, and non-blocking I/O. The most popular approach for accommodating asynchrony in JavaScript applications is event-based programming. In this programming model, a callback function is associated with a specific type of event on a specific object. Events are typically emitted in response to some external activity, e.g., a button being clicked by a user, or a network response arriving, and enqueued for processing. In the main event loop, an event is selected from the queue and its associated callback is invoked and executed until completion. Unfortunately, event-based programming in JavaScript leads to highly convoluted and deeply nested control flow that is sometimes referred to as "callback hell". Such code is highly error-prone due to the non-obvious flow of control and lack of support for error handling. In particular, errors do not propagate between

Authors' addresses: Magnus Madsen, University of Waterloo, Canada, mmadsen@uwaterloo.ca; Ondřej Lhoták, University of Waterloo, Canada, olhotak@uwaterloo.ca; Frank Tip, Northeastern University, USA, f.tip@northeastern.edu.

2475-1421/2017/10-ART86
https://doi.org/10.1145/3133910

different event handlers, and when an exception occurs during the execution of one event handler, other event handlers may still be scheduled for execution. As a result, the effects of errors may not manifest themselves until much later during a program's execution, making the debugging of event-based code extremely challenging. Furthermore, several types of errors may occur that are specific to the event-driven programming model, such as *lost events* (situations where an event is emitted before a handler is registered) and *dead listeners* (situations where an event handler is never executed because the handler was registered too late or on the wrong object) [Madsen et al. 2015], and *event race errors*, i.e., situations where program behavior fails non-deterministically depending on the order in which event handlers are scheduled for execution [Adamsen et al. 2017; Hong et al. 2014; Jensen et al. 2015; Mutlu et al. 2015; Petrov et al. 2012; Raychev et al. 2013; Zhang and Wang 2017; Zheng et al. 2011].

In response to these problems, the JavaScript community has adopted *promises*, a programming model for asynchronous computing that was originally proposed by Friedman and Wise [1976]. A promise represents the value of an asynchronous computation, and is in one of three states (pending, fulfilled, or rejected). Promises come equipped with two functions, `resolve` and `reject`, which are used to resolve or reject the promise with a particular value, respectively. The `then` operation creates a new promise whose value is determined by how the original promise is resolved or rejected. This enables programmers to *chain* asynchronous computations and propagate errors from one asynchronously executed function to another. Several implementations of promises have been developed in the form of libraries, including BlueBird [Antonov 2013] and Q [Kowal 2010], and popular frameworks such as jQuery provide implementations of promises as well. More recently, ECMAScript 6 introduced a standard implementation of promises that has been gaining acceptance [ECMA 2015, Section 25.4]. However, the semantics of JavaScript promises are quite complex, and since the feature is implemented by way of ordinary function calls, there are no static checks to ensure correct usage. As a result, programmers often make mistakes in promise-based code that lead to pernicious errors, as is evident from many reported issues on forums such as StackOverflow. One of the key contributions of this paper is a classification of common types of errors related to JavaScript promises.

The long-term goal of our research is to develop tools for detecting errors in promise-based JavaScript code. However, the ECMAScript 6 standard specifies the semantics of promises informally and in operational terms, and is not a suitable basis for formal reasoning or program analysis. This paper overcomes this obstacle by presenting λ_p, a calculus in which the behavior of promises is expressed as an extension of λ_{JS} [Guha et al. 2010]. The λ_p calculus reflects the behavior of ECMAScript 6 promises, and includes its most important features, while omitting unnecessary and distracting details. Based on λ_p, we introduce the *promise graph* as a program artifact to help understand and debug promise-based programs. We demonstrate the usefulness of the promise graph by reporting on a case study in which we investigate how the promise graph can be used to understand promise-related problems reported on StackOverflow.

In summary, the paper makes the following contributions:

- a classification of common programming errors related to JavaScript promises.
- the λ_p calculus, which provides a formal semantics for a subset of JavaScript promises.
- the *promise graph*, a program artifact which can be used to detect promise-related errors.

2 MOTIVATION

This section presents a brief primer on JavaScript promises, discusses some common questions about promises, and presents an example that illustrates the types of errors that may arise in JavaScript programs that use them.

2.1 Brief Primer on JavaScript Promises

A *promise* is an *object* that represents the *result of an asynchronous operation.* A promise is always in one of three states:

Pending: The asynchronous operation has not yet completed and the promise holds no value. This is the initial state of every promise.

Fulfilled: The asynchronous operation has succeeded and the promise holds a result value.

Rejected: The asynchronous operation has failed and the promise holds an error value.

A promise is *settled* if it is either fulfilled or rejected.

Promises are constructed using a `Promise` constructor that takes a single callback function with two parameters. The parameters of this function are two functions (typically called `resolve` and `reject` in idiomatic JavaScript) that must be invoked to resolve or reject the promise, respectively. For example, the code fragment in Figure 1(a) creates a promise that is immediately resolved with the value 42. Once a promise is settled, subsequent calls to `resolve` or `reject` do not affect the promise's state.

The `then` function can be used to register *resolve reactions* and *reject reactions* with promises. Such reactions are ordinary functions that are invoked in the event loop when the promise they are associated with is resolved or rejected, respectively. However, the `then` function goes beyond merely executing the associated resolve reaction or reject reaction by creating a *dependent promise* that is resolved (or rejected) with the *result* computed by the resolve (reject) reaction. This enables the construction of a *chain* of promises, where computed values or error values are passed from one computation to the next. For example, the code in Figure 1(b) first creates a promise that is resolved with the value 42. Subsequently, a resolve reaction is associated with this promise that prints this value and that creates a dependent promise that is resolved with the value 84.

Note that, in the example of Figure 1(b), only a resolve reaction was supplied, but it is possible to provide both resolve and reject reactions, as shown in Figure 1(c)[1]. Here, if the promise assigned to `p1` is resolved with value 17, then the resolve reaction will print '`x = 17`' and the dependent

[1] Furthermore, reactions can in turn return promises, as we will see in the example of Figure 2.

```
var p = new Promise((resolve, reject) => {
    resolve(42);
});
```

(a)

```
var p = new Promise((resolve, reject) => {
    resolve(42);
});
var p2 = p.then(value => {
    // prints 42
    console.log(value)
    // resolves p2 with 84.
    return value + 42;
})
```

(b)

```
1  var p1 = new Promise((resolve, reject) => {
2      if (...) {
3          resolve(17);
4      } else {
5          reject(18);
6      };
7  });
8  var p2 = p.then(function(x) {
9      console.log("x = " + x);
10     return 19;
11 },
12 function(y) {
13     console.log("error");
14     return 20;
15 });
```

(c)

Fig. 1. Examples of JavaScript promises.

promise `p2` will be resolved with the value 19. If `p1` is rejected with the value 18, the reject reaction will be executed, causing "`error`" to be printed, and the dependent promise `p2` will be resolved with the value 20.[2] Thus, the `then` function can be used to construct a *chain* of promises of the form: `p1.then(...,...)..then(...,...).then(...,...)`.

The use of promises has several advantages over event-driven programming:

- Unlike event handlers, reactions will still be scheduled for execution when their association with a promise is created *after* that promise has already been settled. This prevents "lost event" errors that may arise in event-driven programs [Madsen et al. 2015].
- In a chain of promises, it is possible to propagate error values (e.g., exceptions, error values, or error messages) from a promise to a dependent promise. By contrast, in event-driven programs, exceptions propagate to the top-level and cannot be propagated to a subsequently executed event handler.
- The chaining of promises leads to code where subsequently executed reactions appear sequentially, at the same level. By contrast, traditional event-driven programming leads to deeply nested code that is difficult to understand, a situation that is sometimes referred to as "callback hell" or the "pyramid of doom".

2.2 Debugging a Program with Promises – StackOverflow Question Q42408234

Figure 2(a) shows a simplified program fragment from StackOverflow question Q42408234, entitled "*Difficulty handling Promises in javascript*". The example has considerable complexity, but in our experience, it is representative of the types of problems that programmers face with promise-based programming. In his question, the programmer writes: "*I'm not sure why the promise from a mongoose find is returning undefined.*" and provides two code fragments. Here, "mongoose" refers to a JavaScript interface for the MongoDB database.

The high-level structure of the code is as follows: The function `login` (lines 1–5) calls the function `validateLogin` (lines 7–32), passing the parameters `nameOrEmail` and `password`. The `login` function expects `validateLogin` to return a promise that is eventually resolved with an object that represents whether or not the login was successful. Unexpectedly, on line 3, the `result` parameter is `undefined` when the reaction is called. This is a classic debugging problem: Why does the parameter `result` have the value `undefined`? Is the bug in the library, as hinted at in the programmer's comments, or in his own code? The use of promises obscures both the control– and data flow, making it very difficult to understand from where the value `undefined` originates.

Let us manually try to debug the program. The implementation of `login` looks reasonable suggesting that the bug is in the function `validateLogin`. Inside `validateLogin`, lines 8–14 perform some simple validation of the values of the parameters `nameOrEmail` and `password` that seems unrelated to the problem at hand.

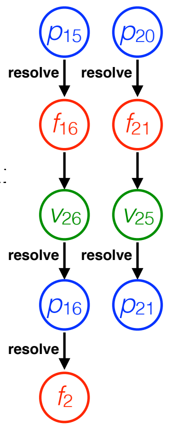
Line 15 contains a call `db.User.find(...)` to find a user in the database. This call returns a promise that is eventually resolved with an object that represents a user with the given `nameOrEmail` and `password`. On line 16, the programmer calls `then` on this promise to register the function on lines 16–31 as a resolve reaction with this promise. This creates a dependent promise that is the promise that is actually returned by `validateLogin`. At this point, it becomes clear that it is this dependent promise that is eventually resolved with the value `undefined`.

To understand why, consider the resolve reaction on lines 16–31, and observe that it is a big if-then-else statement. The else-branch (lines 27–30) always returns an object literal, so we conclude that the bug must be located somewhere in the then-branch. The then-branch (lines 18–26), contains a call on line 20 to `bcrypt.compare(...)`, which returns another promise that is eventually resolved

[2] Replacing "`return 20`" in the reject reaction with "`throw 20`" would cause the dependent promise to be *rejected* instead.

```
 1  function login(req, res) {
 2    validateLogin(req.nameOrEmail, req.password).then((result) => {
 3      // Question: Why is 'result' undefined here??
 4    }
 5  };
 6
 7  function validateLogin(nameOrEmail, password) {
 8    var errors = {};
 9    if (validator.isEmpty(nameOrEmail)) {
10      errors.nameOrEmail = 'username is required';
11    }
12    if (validator.isEmpty(password)) {
13      errors.password = 'password is required';
14    }
15    return db.User.find({$or:[{ username: nameOrEmail }, { email: nameOrEmai
16      .then(existingUser => {
17        if (existingUser.length > 0) {
18          // User exists, check if password matches hash
19          var user = existingUser[0];
20          bcrypt.compare(password, user.password_digest)
21            .then(valid => {
22              if (!valid){
23                errors.password = 'Invalid Password';
24              }
25              return { isValid: isEmpty(errors), errors };
26          })
27        } else {
28          errors.nameOrEmail = 'username or email does not exist';
29          return { isValid: isEmpty(errors), errors };
30        }
31    });
32  }
```

(a)



(b)

Fig. 2. (a) Program fragment from StackOverflow question Q42408234. (b) Promise graph for this program fragment.

with an object that indicates whether or not the correct password was entered. The call to `then` on line 21 registers the function on lines 21–26 as a resolve reaction for this promise. After some more validation steps, this function returns an object as expected, so what is wrong?

The problem has to do with the fact that the two return statements occur in different functions:

- The return on line 29 occurs in the function that spans lines 16–31. Note that this function is registered as a reaction for the promise returned by the call `db.User.find(...)` on line 15.
- The return on line 25 occurs in the function that spans lines 21–26. Note that this function is registered as a reaction for the promise returned by the call `bcrypt.compare(...)` on line 20.

At this point, the problem becomes clear. If the `then`-branch of the if-statement on line 17 is executed, the function on lines 16–31 does not explicitly return a value, so it implicitly returns `undefined`, and this explains why the promise created by the call to `then` on line 16 is resolved with the value `undefined`. As a related matter, one may observe that the value computed by the dependent promise created by the call to `then` on line 21 is not used anywhere: the promise and its result is "lost". Clearly, the programmer's intention was to return the promise created by the call to `then` on line 21, and this can be accomplished by explicitly returning the value computed by the expression `bcrypt.compare(password, user.password_digest).then(...)`.

The long-term goal of our research is to help programmers debug such issues. For that purpose, we introduce the *promise graph*. The relevant parts of the promise graph for the program of Figure 2(a) are shown in Figure 2(b). From this graph, one may observe that the function f_2, on line 2, receives the value v_{26} (the `undefined` value) from line 26 through the promise p_{16} created on line 16. However, what the programmer intended to happen was for this promise to receive the value v_{25} created on line 25. As the graph shows, the value v_{25} is returned by the function f_{21} on line 21 and used to resolve the promise p_{21} created on line 20, but, and this is crucial, the promise p_{21} is never used for anything and hence the value v_{25} is lost. At a high level, the problem is that the two promise chains are *disconnected*. The programmer wanted these chains to be connected by way of an edge $p_{21} \rightarrow p_{16}$. Specifically, the intention was to return the promise of `bcrypt` for resolving the promise created by `db.User.find(...).then`.

As a testament to the complexity of this problem, upon getting the answer, the programmer replied: "*Wonderful! What a major slip up by me. I spent a solid two hours trying to fix this.*" illustrating just how hard it can be to debug promise-based code.

3 SEMANTICS

We now turn our attention to a formal treatment of the semantics of promises in JavaScript. In this section, we develop a reduction semantics for promises as an extension of the λ_{JS} calculus.

Using a minimal calculus allows us to focus on the essence of promises, and avoid obscuring the important details by other complex JavaScript features, such as prototype-based inheritance, dynamic property access, implicit coercions and on-the-fly code evaluation with `eval`, to which much research has already been dedicated [Guha et al. 2010; Jensen et al. 2012, 2011, 2009; Kashyap et al. 2014; Madsen and Andreasen 2014; Madsen et al. 2013; Madsen and Møller 2014].

3.1 The ECMAScript Specification

The ECMAScript 6 specification [ECMA 2015] describes the semantics of promises in JavaScript. The description consists of approximately 3,500 words written in terse prose. It explains the semantics in pseudo-code as a sequence of steps. No motivation or informal commentary is provided. Furthermore, the semantics rests on many "specification specific" operations, such as `ReturnIfAbrupt`, `IsCallable` and data structures such as "records" and "internal slots" that do not exist in the surface syntax of JavaScript.

We now proceed as follows: We begin with a discussion of the runtime environment for promises as specified in ECMAScript 6. This gives us an idea of the complexities of the specification, but also a sense of what promises really are. We use this to motivate some design choices. Next, we present our syntax and runtime extensions of λ_{JS} before presenting the evaluation rules of our calculus λ_p.

Promise Objects and Functions. In ECMAScript 6, a `Promise` object has four *internal slots*:[3]

`[[PromiseState]]`: A string `"pending"`, `"fulfilled"` or `"rejected"` representing the state of the promise.

`[[PromiseResult]]`: The value that the promise has been settled (i.e., fulfilled or rejected) with. If the promise is pending the value is `undefined`.

`[[PromiseFulfillReactions]]`: A list of *promise reactions*. A fulfill reaction is a function that should be executed asynchronously when the promise is fulfilled.

`[[PromiseRejectReactions]]`: A list of *promise reactions*. A reject reaction is a function that should be executed asynchronously when the promise is rejected.

[3] An internal slot is a hidden property. Internal slots are written as `[[name]]`.

Informally, we can think of promise reactions as event handlers that are triggered when a promise is resolved or rejected. However, unlike event handlers, promise reactions also determine the values that are used to resolve or reject dependent promises, as will be explained shortly.

A `PromiseCapability` object has three internal slots:

`[[Promise]]`: A reference to a `Promise` object.

`[[Resolve]]`: A function that resolves the promise. This function expects a single argument, which is the value to resolve the promise with.

`[[Reject]]`: A function to reject the promise. This function also expects a single argument, which is the value (typically an exception value, an error value, or an error message) to reject the promise with.

A `PromiseCapability` is really what the programmer sees as a promise object. Its `[[Resolve]]` and `[[Reject]]` functions, which themselves are objects, have two internal slots:

`[[Promise]]`: A reference to a `Promise` object.

`[[AlreadyResolved]]`: A boolean indicating whether the promise has been *settled* (i.e., resolve or rejected). The name is confusing, it would have been more accurate to call it `AlreadySettled`.

Finally, we get to `PromiseReaction`, which has two slots:

`[[Capabilities]]`: A reference to a `PromiseCapability` object. Note that despite the name being plural, the slot only refers to a single capability.

`[[Handler]]`: This is either a function object, the string `"Identity"`, or the string `"Thrower"`. The latter serve to compactly represent situations where a reaction consists of the identity function, or a similar function that throws its argument.

Based on these data structures, ES6 defines the core operations on promises: `resolve`, `reject`, `then`, `catch`.

3.2 Design Decisions

We formulate the λ_p calculus as an extension of the λ_{JS} calculus [Guha et al. 2010]. λ_p has a small-step reduction semantics with additional runtime components to track, schedule, and execute promises, and operations to create, chain, and resolve or reject promises. Similar to JavaScript and λ_{JS}, we assume execution to be single-threaded and non-preemptive. Features such as prototype chains, coercions, and dynamic fields, are as specified in λ_{JS}. In defining λ_p, we made the following simplifying design decisions:

- *Explicit syntax* is introduced for creating a promise (`promisify(e)`), resolving a promise (`e.resolve(e)`) and rejecting a promise (`e.reject(e)`).
- ECMAScript 6 distinguishes between `Promise` and `PromiseCapability` objects, λ_p merges these two objects. Furthermore, we will allow any object to be turned into a promise.
- High-level operations such as `then` and `catch` are modeled using lower-level primitives. Specifically, `catch` is a special case of `then`, and `then` is expressed as a combination of the primitives `onResolve`, `resolve`, and `link`.
- The relationship between dependent promises is modeled using an explicit `link` operation.

3.3 Syntax of λ_p

λ_p extends the surface syntax of λ_{JS}[4] with six new expressions, as shown in Figure 3(a).

A *promisify* expression turns an object into a promise. As discussed, a promise is in one of three states: pending (`P`), fulfilled (`F`), or rejected (`R`). Upon creation, a promise is in the pending (`P`) state. A *resolve* expression fulfills a promise with a value and causes its resolve reactions to be

[4] For convenience, the syntax of λ_{JS} can be found in Appendix A.

$$\begin{array}{rcll} e \in \mathit{Exp} & = & \texttt{promisify}\,(e) & \text{[create promise]} \\ & | & e.\texttt{resolve}\,(e) & \text{[resolve promise]} \\ & | & e.\texttt{reject}\,(e) & \text{[reject promise]} \\ & | & e.\texttt{onResolve}\,(e) & \text{[chain promise]} \\ & | & e.\texttt{onReject}\,(e) & \text{[chain promise]} \\ & | & e.\texttt{link}\,(e) & \text{[link promises]} \end{array}$$

(a)

$$\begin{array}{rcl} E & = & \square \\ & | & \texttt{promisify}\,(E) \\ & | & E.\texttt{resolve}\,(e) \mid v.\texttt{resolve}\,(E) \\ & | & E.\texttt{reject}\,(e) \mid v.\texttt{reject}\,(E) \\ & | & E.\texttt{onResolve}\,(e) \mid v.\texttt{onResolve}\,(E) \\ & | & E.\texttt{onReject}\,(e) \mid v.\texttt{onReject}\,(E) \\ & | & E.\texttt{link}\,(e) \mid v.\texttt{link}\,(E) \end{array}$$

(b)

Fig. 3. (a) Syntax of λ_p. (b) Evaluation Contexts for λ_p.

$$\begin{array}{rcl} \sigma \in \mathit{Heap} & = & \mathit{Addr} \hookrightarrow \mathit{Val} \\ \psi \in \mathit{PromiseState} & = & \mathit{Addr} \hookrightarrow \mathit{PromiseValue} \\ f \in \mathit{FulfillReactions} & = & \mathit{Addr} \hookrightarrow (\mathit{Reaction} \times \mathit{Addr})^\star \\ r \in \mathit{RejectReactions} & = & \mathit{Addr} \hookrightarrow (\mathit{Reaction} \times \mathit{Addr})^\star \\ \pi \in \mathit{Queue} & = & (\mathit{PromiseValue} \times \mathit{Reaction} \times \mathit{Addr})^\star \\ \rho \in \mathit{Reaction} & = & \mathit{Lam} \mid \text{default} \\ \Psi \in \mathit{PromiseValue} & = & \{\mathsf{P},\ \mathsf{F}(\mathit{Val}),\ \mathsf{R}(\mathit{Val})\} \end{array}$$

Fig. 4. Runtime of λ_p.

scheduled for execution by the event loop. Similarly, a *reject* expression rejects a promise with a value and causes its reject reactions to be scheduled for execution by the event loop. An `onResolve` expression registers a resolve reaction on a promise and returns a dependent promise. Similarly, an `onReject` expression registers a reject reaction on a promise and returns a dependent promise. A `link` expression registers a dependency between two promises such that when the former is resolved (or rejected) the latter is resolved (or rejected) with the same value.

As in λ_{JS} [Guha et al. 2010], we use evaluation contexts [Felleisen et al. 2009] to explain how sub-expressions are evaluated. Figure 3(b) shows the evaluation contexts for the extended language. Here, the $\square$ symbol represents the hole in the evaluation context.

3.4 Runtime of λ_p

The runtime state of λ_p is similar to that of λ_{JS}, but extended with four additional components: A *promises state* ψ maps each address to an algebraic data type $\Psi \in \mathit{PromiseValue}$, which is one of pending P, fulfilled $\mathsf{F}(v)$, or rejected $\mathsf{R}(v)$ where v is the result value of the promise. The *fulfill reactions* f and *reject reactions* r each map an address to a list of *reaction* and *dependent promise* pairs for a pending promise. A reaction ρ is lambda or the special default reaction. For now, the default reaction can be thought of as the identity function. When a pending promise is resolved (or rejected) with a value v, then the reaction, i.e. lambda expression or default, is evaluated with argument v and the result of *that evaluation* is used to further resolve (or reject) the dependent promise. The final component is a *queue* π of scheduled reactions, i.e., promises that have been settled for which the reactions are awaiting asynchronous execution by the event loop.

A *program state* is a five tuple $\langle \sigma, \psi, f, r, \pi \rangle$ of these components together with a heap σ. Figure 4 shows the grammar for the runtime of λ_p.

3.5 Semantics of λ_p

We now present the reduction semantics of λ_p, which extends the semantics of λ_{JS} with the evaluation rules shown in Figure 5 and Figure 6. We have omitted a few rules related to `reject` since they are similar to the rules for `resolve`. The semantics consists of two evaluation relations:

$$\frac{e \hookrightarrow e'}{\langle \sigma, \psi, f, r, \pi, E[e] \rangle \to \langle \sigma', \psi', f', r', \pi', E[e'] \rangle} \quad \text{[E-Context]}$$

$$\frac{\begin{array}{c} a \in \mathit{Addr} \qquad a \in \mathsf{dom}(\sigma) \qquad a \notin \mathsf{dom}(\psi) \\ \psi' = \psi[a \mapsto \mathsf{P}] \qquad f' = f[a \mapsto \mathit{Nil}] \qquad r' = r[a \mapsto \mathit{Nil}] \end{array}}{\langle \sigma, \psi, f, r, \pi, E[\mathtt{promisify}(a)] \rangle \to \langle \sigma, \psi', f', r', \pi, E[\mathsf{undef}] \rangle} \quad \text{[E-Promisify]}$$

$$\frac{\begin{array}{c} a \in \mathit{Addr} \qquad a \in \mathsf{dom}(\sigma) \qquad \psi(a) = \mathsf{P} \\ a' \in \mathit{Addr} \qquad a' \notin \mathsf{dom}(\sigma) \qquad \psi' = \psi[a' \mapsto \mathsf{P}] \qquad \sigma' = \sigma[a' \mapsto \{\}] \\ f' = f[a \mapsto f(a) ::: (\lambda, a')][a' \mapsto \mathit{Nil}] \qquad r' = r[a' \mapsto \mathit{Nil}] \end{array}}{\langle \sigma, \psi, f, r, \pi, E[a.\mathtt{onResolve}(\lambda)] \rangle \to \langle \sigma', \psi', f', r', \pi, E[a'] \rangle} \quad \text{[E-OnResolve-Pending]}$$

$$\frac{\begin{array}{c} a \in \mathit{Addr} \qquad a \in \mathsf{dom}(\sigma) \qquad \psi(a) = \mathsf{F}(v) \\ a' \in \mathit{Addr} \qquad a' \notin \mathsf{dom}(\sigma) \qquad \psi' = \psi[a' \mapsto \mathsf{F}(v)] \qquad \sigma' = \sigma[a' \mapsto \{\}] \\ f' = f[a' \mapsto \mathit{Nil}] \qquad r' = r[a' \mapsto \mathit{Nil}] \\ \pi' = \pi ::: (\mathsf{F}(v),\ \lambda,\ a') \end{array}}{\langle \sigma, \psi, f, r, \pi, E[a.\mathtt{onResolve}(\lambda)] \rangle \to \langle \sigma', \psi', f', r', \pi', E[a'] \rangle} \quad \text{[E-OnResolve-Fulfilled]}$$

$$\frac{a \in \mathit{Addr} \qquad a \in \mathsf{dom}(\sigma) \qquad \psi(a) \in \{\mathsf{F}(v'),\ \mathsf{R}(v')\}}{\langle \sigma, \psi, f, r, \pi, E[a.\mathtt{resolve}(v)] \rangle \to \langle \sigma, \psi, f, r, \pi, E[\mathsf{undef}] \rangle} \quad \text{[E-Resolve-Settled]}$$

$$\frac{\begin{array}{c} a \in \mathit{Addr} \qquad a \in \mathsf{dom}(\sigma) \qquad \psi(a) = \mathsf{P} \\ f(a) = (\lambda_1, a_1) \cdots (\lambda_n, a_n) \qquad \pi' = \pi ::: (\mathsf{F}(v), \lambda_1, a_1) \cdots (\mathsf{F}(v),\ \lambda_n,\ a_n) \\ \psi' = \psi[a \mapsto \mathsf{F}(v)] \qquad f' = f[a \mapsto \mathit{Nil}] \qquad r' = r[a \mapsto \mathit{Nil}] \end{array}}{\langle \sigma, \psi, f, r, \pi, E[a.\mathtt{resolve}(v)] \rangle \to \langle \sigma, \psi', f', r', \pi', E[\mathsf{undef}] \rangle} \quad \text{[E-Resolve-Pending]}$$

$$\frac{\begin{array}{c} a_1 \in \mathit{Addr} \qquad a_1 \in \mathsf{dom}(\sigma) \qquad a_2 \in \mathit{Addr} \qquad a_2 \in \mathsf{dom}(\sigma) \qquad \psi(a_1) = \mathsf{P} \\ f' = f[a_1 \mapsto f(a_1) ::: (\mathsf{default}, a_2)] \qquad r' = f[a_1 \mapsto r(a_1) ::: (\mathsf{default}, a_2)] \end{array}}{\langle \sigma, \psi, f, r, \pi, E[a_1.\mathtt{link}(a_2)] \rangle \to \langle \sigma, \psi, f', r', \pi, E[\mathsf{undef}] \rangle} \quad \text{[E-Link-Pending]}$$

$$\frac{\begin{array}{c} a_1 \in \mathit{Addr} \qquad a_1 \in \mathsf{dom}(\sigma) \qquad a_2 \in \mathit{Addr} \qquad a_2 \in \mathsf{dom}(\sigma) \qquad \psi(a_1) = \mathsf{F}(v) \\ \pi' = \pi ::: (\mathsf{F}(v), \mathsf{default}, a_2) \end{array}}{\langle \sigma, \psi, f, r, \pi, E[a_1.\mathtt{link}(a_2)] \rangle \to \langle \sigma, \psi, f, r, \pi', E[\mathsf{undef}] \rangle} \quad \text{[E-Link-Fulfilled]}$$

Fig. 5. Semantics of λ_p. Here *Nil* is the empty list, the notation $x :: xs$ is cons, and the notion $xs ::: ys$ is append. The rules for [E-OnReject-Pending], [E-OnReject-Rejected], [E-Reject-Pending], [E-Reject-Settled], and [E-Link-Rejected] are not shown, but are conceptually similar to other rules.

The relation $\hookrightarrow$ represents expression-to-expression reductions, whereas the $\to$ relation represents state-to-state transitions. The rules we add are impure and are therefore defined as state transitions in terms of $\to$.

[E-Context]. This rule enables evaluation and recomposition of expressions according to the evaluation contexts in Figure 3(b). That is, if some expression e_1 can evaluate to e_2 in the single step relation $\hookrightarrow$, and e_1 occurs in the hole of the evaluation context, then we can plug the hole with e_2.

$$\frac{\pi = (\mathsf{F}(v'),\ \lambda_r, a) :: \pi'}{\langle \sigma, \psi, f, r, \pi, v\rangle \rightarrow \langle \sigma, \psi, f, r, \pi', a.\mathtt{resolve}(\lambda_r(v'))]\rangle} \quad \text{[E-Loop-Fulfilled-Lambda]}$$

$$\frac{\pi = (\mathsf{R}(v'),\ \lambda_r, a) :: \pi'}{\langle \sigma, \psi, f, r, \pi, v\rangle \rightarrow \langle \sigma, \psi, f, r, \pi', a.\mathtt{resolve}(\lambda_r(v'))]\rangle} \quad \text{[E-Loop-Rejected-Lambda]}$$

$$\frac{\pi = (\mathsf{F}(v'), \mathsf{default}, a) :: \pi'}{\langle \sigma, \psi, f, r, \pi, v\rangle \rightarrow \langle \sigma, \psi, f, r, \pi', a.\mathtt{resolve}(v')]\rangle} \quad \text{[E-Loop-Fulfilled-Default]}$$

$$\frac{\pi = (\mathsf{R}(v'), \mathsf{default}, a) :: \pi'}{\langle \sigma, \psi, f, r, \pi, v\rangle \rightarrow \langle \sigma, \psi, f, r, \pi', a.\mathtt{reject}(v')]\rangle} \quad \text{[E-Loop-Rejected-Default]}$$

Fig. 6. Loop Semantics of λ_p.

[E-Promisify]. This rule turns an address into a promise. The rule states that if there is an expression promisify(a) where a is an address in the heap and the address is not already in one of the three promise states, then we initialize the promise state to pending P and initialize the resolve and rejection reactions to the empty list.

[E-OnResolve-Pending]. This rule registers a fulfill reaction on a pending promise. Specifically, if the expression is a.onResolve(λ), where a is the address of an object allocated in the heap, λ is the fulfill reaction, and the promise is in the pending state, then the rule allocates a dependent promise with address a', initializes its reactions to empty list, and—most importantly—adds the pair (λ, a') to the fulfill reactions of the original promise a. The idea is that when a is eventually resolved, the function λ will be executed asynchronously by the event loop. Its return value will then be used to resolve the dependent promise a'.

[E-OnResolve-Fulfilled]. This rule handles the case when a fulfill reaction is registered on a promise that is already resolved. In this case, the fulfill reaction is immediately scheduled for execution by placing it in the queue. Specifically, if the expression is a.onResolve(λ) where, as before, a is an address allocated in the heap, and the promise has already been fulfilled with a value v according to the promise state map ψ, then we allocate a dependent promise a' but instead of adding the pair (λ, a') to the fulfill reaction of a we enqueue (append) the triple $(\mathsf{P}(v), \lambda, a')$ to the queue and return a'. The effect is that, despite the promise already having been resolved, the reaction being registered is scheduled for execution. This behavior differs from that of event handlers where registering a handler too late means that it will never be executed.

[E-Resolve-Settled]. This rule simply states that resolving a settled promise has no effect. Specifically, if the expression is a.resolve(v) but the state of the promise a according to the promise state map ψ indicates that the promise has already been resolved (or rejected), then the resolve has no effect.

[E-Resolve-Pending]. This rule handles the case when a pending promise is resolved. Resolving a promise requires multiple things to happen: (a) The fulfill reactions are extracted from the promise and enqueued with the value used to resolve the promise with. (b) The fulfill and reject reactions are cleared from the promise. (c) The promise state is changed from pending to fulfilled (or rejected). Specifically, if the expression is a.resolve(v), then a must be an address allocated in the heap and the promise state of a must be pending. Then the fulfill reactions of a are extracted. This is a list of

(λ_i, a_i) pairs where the λ is the reaction and a_i is its dependent promise. We then enqueue a triple $(\mathsf{F}(v), \lambda_i, a_i)$ for each such reaction. Finally, we clear out the reactions for a in the fulfill and reject maps f and r and update the promise state ψ to $\mathsf{F}(v)$.

[E-Link-Pending]. This rule causes one promise to be "linked" to another. This means that when the former is resolved (or rejected), the latter will be resolved (or rejected) with the same value. Specifically, if the expression is a_1.link(a_2), then the rule registers two reactions on a_1; a fulfill reaction (default, a_2) and a reject reaction (default, a_2). The effect is that when a_1 is resolved (or rejected), then the registered reaction will be executed with what is effectively the identity function causing a_2 to be resolved (or rejected) with the same value as a_1.

[E-Link-Fulfilled]. This rule is similar to [E-Link-Pending] but handles the case where the parent promise is already settled. In this case, the dependent promise with the `default` reaction is immediately scheduled for execution in the event loop.

[E-Loop-Fulfilled-Lambda] and [E-Loop-Rejected-Lambda]. These two rules extracts a fulfill or reject reaction from the queue, executes it with the promise's value, and uses the returned value to *resolve* the dependent promise. Notice that the dependent promise is always *resolved*. This is what allows a reject reaction to "recover" from an error situation. For example, if the entire expression has been reduced to a value, and the event queue contains a triple $(\mathsf{R}(v), \lambda_r, a)$ of a rejected promise value v, a reject reaction λ_r and a dependent promise (address) a, then λ_r is evaluated with the value v and that result is used to resolve the dependent promise a.

[E-Loop-Fulfilled-Default] and [E-Loop-Reject-Default]. These two rules are conceptually similar to the two previous rules with an important difference. Each rule extracts a promise value (either fulfilled or rejected) v and a dependent promise a from the queue. The rule then uses the value v to resolve or reject the dependent promise *a preserving* the fulfilledness or rejectedness of the promise value. In other words, if the promise value was rejected then the dependent promise is *also* rejected.

The rules [E-OnReject-Pending], [E-OnReject-Rejected], [E-Reject-Pending], [E-Reject-Settled] and [E-Link-Rejected] are conceptually similar to the ones discussed above.

3.6 Encoding of Then, Catch, and Other Promise Operations

We now describe how to encode several promise operations in terms of lower-level primitives.

Then. The expression e.then(λ_f, λ_r) is a heavily overloaded operation that contains a lot of functionality. At a high level, then registers the reactions λ_f and λ_r on the promise e and returns a dependent promise. Figure 7 shows a sketch implementation of then in JavaScript syntax using the primitive promise operations of λ_p. The key details are as follows: The function expects two arguments `resolveReaction` and `rejectReaction`. If either is a non-function value, then implicitly uses the identity function instead. The function creates a new dependent promise `p` which is ultimately the result of the call to then. On the `this` value, then registers the resolve and reject reactions: The return value of `resolveReaction` and `rejectReaction` determine how the promise `p` is resolved (or rejected). Specifically, if the reaction returns a promise, then that promise is linked to the promise `p`. If, on the other hand, the reaction returns a regular value, then the promise `p` is resolved (or rejected) with that value.

Catch. The *catch* expression e.catch(λ_r) is simply syntactic sugar for e.then(id, λ_r).

Race. The expression race($a_1, \cdots, a_n$) takes a sequence of promise objects $a_1 \cdots a_n$ and returns a promise that is fulfilled (or rejected) with the value of the *first* promise that is resolved (or rejected).

```
Promise.prototype.then =
  function(resolveReaction, rejectReaction) {
    if (!resolveReaction instanceof Function)
      resolveReaction = identity
    if (!rejectReaction instanceof Function)
      rejectReaction = identity
    var p = promisify({});
    this.onResolve(function(value) {
      var x = resolveReaction(value)
      if (x instanceof Promise) {
        x.link(p);
      } else {
        p.resolve(x);
      }
    })
    // similarly for reject.
    return p;
  }
```

Fig. 7. A sketched implementation of then in λ_p

We can encode this functionality by creating a new promise and adding a reaction to each of the "racing" promises that will fulfill the newly created promise. Since fulfilling a promise twice has no effect, this implements the desired functionality.

All. The expression all($a_1, \cdots, a_n$) takes a sequence of promise objects $a_1 \cdots a_n$ and returns a promise that is fulfilled with an array of values when all the promises have been fulfilled, or rejected with the value of the *first* promise to reject. To this end, we create a new promise and add a counter to it. Whenever a promise is resolved, the counter is decremented. Once the counter reaches zero, we fulfill the promise. If a promise ever rejects, we immediately reject the created promise.

3.7 Promises & Exceptions

We have presented the core semantics without any reference to exceptions. As it turns out, exceptions are *not* purely orthogonal to promises. Whenever a fulfill or reject reaction *throws* an exception, that exception does not propagate upwards through the call stack nor is it left uncaught. Instead, the event loop catches the exception and uses it to *reject* the dependent promise. We can account for this by a suitable adjustment to the [E-Loop-Fulfilled-Lambda] and [E-Loop-Rejected-Lambda] rules. Specifically, these rules would catch an exception thrown by the reaction λ_r and then use the exceptional value to reject the dependent promise.

4 THE PROMISE GRAPH

The *promise graph* captures control- and dataflow in a promise-based program to represent the flow of values through promises, the execution of fulfill and reject reactions, and the dependencies between reactions and promises. The promise graph contains a *value node* $\textcircled{v}$ for every value allocation site in the program, a *promise node* $\textcircled{p}$ for every promise allocation site in the program, and a *function node* $\textcircled{f}$ for every lambda or named function in the program. The edges in the graph capture the following relationships between these nodes:

- A *resolve* or *reject* edge $\textcircled{v} \xrightarrow{\ell} \textcircled{p}$ from a value node $\textcircled{v}$ to a promise node $\textcircled{p}$, where the label ℓ is either "resolve" or "reject". Intuitively, such an edge represents that v is used to resolve or reject the promise p.
- A *registration* edge $\textcircled{p} \xrightarrow{\ell} \textcircled{f}$ from a promise node $\textcircled{p}$ to a function node $\textcircled{f}$ where the label ℓ is either "resolve" or "reject". Intuitively, the edge represents that the function f is

registered as fulfill or reject reaction on the promise p. In other words, it represents the fact that, when the promise p is resolved or rejected, the execution of f is triggered.

- A *link* edge $p_1 \rightarrow p_2$ from a promise node p_1 to a dependent promise node p_2. The edge represents the dependency that when the parent promise is resolved (or rejected) the dependent promise will be resolved (or rejected) with the same value.
- A *return* edge $f \rightarrow v$ from a function node f to a value node v. Intuitively, this represents that the function f returns the value allocated at v.

4.1 Examples

We now illustrate the promise graph with a few examples. In these examples, we will use subscripts, (e.g., v_{21}, p_{42}, f_{88}) to indicate the line on which a value, promise, or function was allocated.

Example I. The execution of the program shown below:

```
1   var p1 = promisify({});
2   var p2 = p1.onResolve(x => x + 1)
3   p1.resolve(42)
```

$$\begin{tikzcd}
v_3 \arrow[r, "\text{resolve}"] & p_1 \arrow[r, "\text{resolve}"] & f_2 \arrow[r] & v_2 \arrow[r, "\text{resolve}"] & p_2
\end{tikzcd}$$

gives rise to the promise graph on the right. The graph shows that the value allocated on line 3 is used to resolve the promise p_1 allocated on line 1, the value then flows into the fulfill reaction f_2 (the lambda function on line 2). Inside f_2, the value is used to compute a new value allocated on line 2 which is returned by the reaction and thus used to resolve the second promise p_2 allocated on line 2. Note that we can read the graph in a forwards manner starting from v_3 to understand the consequence of resolving the promise p_1. Similarly, we can read the graph backwards starting from p_2 to understand how and why it was resolved.

Example II. The execution of the program shown below:

```
1   var p1 = promisify({});
2   var p2 = p1.onResolve(x => x + 1)
3   var p3 = p1.onReject(x => x - 1)
4   p1.resolve(42)
```

$$\begin{tikzcd}
v_4 \arrow[r, "\text{resolve}"] & p_1 \arrow[r, "\text{resolve}"] \arrow[dr, "\text{reject}"'] & f_2 \arrow[r] & v_2 \arrow[r, "\text{resolve}"] & p_2 \\
 & & f_3 & & p_3
\end{tikzcd}$$

gives rise to the promise graph on the right. The graph shows that the value v_4 flows into the promise p_1 which causes the execution of the fulfill reaction function f_2. This function computes a new value v_2 which is then returned and used to resolve the promise p_2. More importantly, the promise p_1 also has a reject reaction function f_3 registered. But, since promise p_1 is never rejected, the function f_3 is never executed, and hence the dependent promise p_3 is never resolved or rejected.

Example III. The execution of the program:

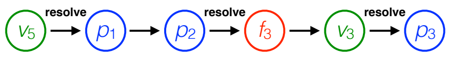
```
1   var p1 = promisify({});
2   var p2 = p1.onResolve(x => x + 1);
3   var p3 = p1.onResolve(y => y - 1);
4   p1.resolve(42);
```

$$\begin{tikzcd}
v_4 \arrow[r, "\text{resolve}"] & p_1 \arrow[r, "\text{resolve}"] \arrow[dr, "\text{resolve}"'] & f_2 \arrow[r] & v_2 \arrow[r, "\text{resolve}"] & p_2 \\
 & & f_3 \arrow[r] & v_3 \arrow[r, "\text{resolve}"] & p_3
\end{tikzcd}$$

gives rise to the promise graph on the right. The graph shows that the value v_4 flows into the promise p_1 which causes the execution of the fulfill reaction functions f_2 *and* f_3. These fulfill reactions compute new values v_2 and v_3, respectively, which are then used to resolve the dependent promises p_2 and p_3, respectively. This program and graph illustrates the scenario where multiple reactions are registered on the same promise.

Example IV. The execution of the program:

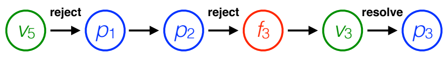
```
var p1 = promisify({});
var p2 = promisify({});
var p3 = p2.onResolve(x => x + 1)
p1.link(p2)
p1.resolve(42)
```


$$\begin{tikzcd} v_5 \arrow[r, "\text{resolve}"] & p_1 \arrow[r] & p_2 \arrow[r, "\text{resolve}"] & f_3 \arrow[r] & v_3 \arrow[r, "\text{resolve}"] & p_3 \end{tikzcd}$$

gives rise to the promise graph on the right. The graph shows that the value v_5 flows into the promise p_1 which, since it is linked to the promise p_2, causes p_2 to be resolved with the same value and that in turn causes the execution of the resolve reaction function f_3. This function computes the value v_3 which is returned and used to resolve the promise p_3.

Example V. The execution of the program:

```
var p1 = promisify({});
var p2 = promisify({});
var p3 = p2.onReject(x => x + 1)
p1.link(p2)
p1.reject(42)
```


$$\begin{tikzcd} v_5 \arrow[r, "\text{reject}"] & p_1 \arrow[r] & p_2 \arrow[r, "\text{reject}"] & f_3 \arrow[r] & v_3 \arrow[r, "\text{resolve}"] & p_3 \end{tikzcd}$$

gives rise to the promise graph on the right. The graph shows that the value v_5 flows into the promise p_1 which, since it is linked to the promise p_2, causes p_2 to be rejected with the same value and that in turn causes the execution of the reject reaction function f_3. This function computes the value v_3 which is returned and used to *resolve* the promise p_3. Note in particular that the reject reaction `x => x + 1` returns 43 and hence the dependent promise p_3 is *resolved* with 43.

4.2 Program Comprehension with the Promise Graph

We can use the promise graph to answer questions about the behavior of promise-based programs:

- Given a value, we can follow its outgoing resolve or reject edges to discover into which promises it can flow and consequently to which functions it can be passed as an argument.
- Given a fulfill or reject reaction, we can discover what values flow to its arguments.
- Given a promise, we can discover what values it is resolved or rejected with.
- We can discover promises that are never resolved or rejected.
- We can discover fulfill or reject reactions that are never executed.
- We can discover promise chains that are disconnected, whether intended or not.

4.3 Bug Finding with the Promise Graph

We now explain how to use the promise graph to detect bugs or suspicious behavior in promise-based programs. We consider three categories of behavior: (a) bugs that we can detect solely by inspection of the promise graph, (b) bugs we can detect based on the promise graph in combination with other information, e.g., a call graph or happens-before relation, and (c) bugs we can detect based on the promise graph and simple syntactic checks.

Dead Promise. We say that a promise is *dead* if an object enters the `Pending` state, either via an explicit `promisify` statement or as a dependent promise created by `onResolve` or `onReject` and it never transitions to either a `Fulfill` or `Reject` state. That is, a promise is created but it is never fulfilled nor rejected either explicitly or implicitly.

```
var p = new Promise((resolve, reject) => {
  // resolve and reject are not called.
});

// resolve and reject do not escape
// and are not called subsequently.

p.then(x => /* dead code */);
```

Consider the code fragment above, on the right. Here a promise `p` is created, but neither `resolve` nor `reject` associated with the promise is ever called. Consequently, the promise never leaves the

Pending state. Furthermore, any resolve or reject reactions associated with the promise are never executed and are thus dead code. In this case, the function on line 8 is never executed. We can use the promise graph to discover such issues. Specifically, whenever we have a promise with no *resolve* or *reject* edges nor any *link* edge then the promise is *dead*, and any reactions registered on it are never executed (by that promise).

Missing Resolve or Reject Reaction. We say that a promise is *missing a resolve (or reject) reaction* if an object enters the Pending state, the promise is resolved (or rejected) with a non-undefined value, and the promise lacks a fulfill (or reject) reaction.

```
var p = new Promise((resolve, reject) => {
  setTimeout(5000, resolve(42))
});
// p.then is never called, and
// consequently the value 42 is lost.
```

Consider the code fragment above, on the right. Here the promise p is created and after five seconds it is resolved with the value 42 with a call to resolve. However, no fulfill reactions are registered on the promise and hence the value is lost. We can use the promise graph to detect such situations by looking for promises that have no outgoing registration edges.

An important point, which was briefly mentioned earlier, is that—unlike event handlers—it is entirely acceptable for a fulfill (or reject) reaction to be registered *after* a promise has already been fulfilled (or rejected). In that case the reaction is simply scheduled for execution with the promise value. Thus, a promise could be fulfilled (or rejected) at a moment when it does not have any reactions, and as long as a reaction is registered at some point in the future, the value is not lost.

Missing Exceptional Reject Reaction. A special case of the previous bug pattern occurs when a *promise is implicitly rejected by throwing an exception.*

```
1  var p1 = new Promise((resolve, reject) => {
2    setTimeout(5000, resolve(42))
3  });
4  var p2 = p1.then(function(x) {
5    throw Error("...")
6  });
7  // a reject reaction is never registered
8  // on `p2` with either `then` or `catch`.
```

Consider the code fragment on the right. Here, the resolve reaction on lines 4-6 registered for the promise p1 throws an exception, which causes the second promise p2 to be rejected with the exceptional value. However, since no reject reaction is registered on p2, the exception is silently ignored. We can discover such situations, by looking for promises with a reject edge, but no reject registration edge.

Missing Return. A *missing return* occurs when a fulfill or reject reaction unintentionally returns undefined and this value is used to resolve or reject a dependent promises. Consider the program fragment on the right. The programmer intended to create a promise chain passing through an integer which is ultimately shown to the user in an alert box. However, the resolve reaction which logs the value x to the console has no explicit return statement. Hence the function returns undefined and *not* x as the programmer had in mind. Consequently, the dependent promise is resolved with undefined which is the value passed into the subsequent reaction and then shown to the user.

```
var p1 = new Promise((resolve, reject) => {
  resolve(42);
});
var p2 = p1.then(x => {
  console.log(x);
});
var p3 = p2.then(x => {
  alert(x);
});
```

Double Resolve or Reject. A *double resolve* or *double reject* occurs when a promise is created, resolve (or reject) is called on it and the promise enters the fulfilled or rejected state,

```
var p = new Promise((resolve, reject) => {
  resolve(42);
  resolve(21); // dead code, no effect
});
```

and later the promise is again resolved (or rejected). The second and any subsequent time a promise is resolved (or rejected) has no effect on the state of the promise and does not cause any of its reactions to be executed. Consider the program fragment above, on the right. We can use the promise-graph to detect double resolves (or rejects) by checking if there are multiple resolve (or reject) edges leading to the same promise. In some cases, some ingenuity by the programmer will be required to understand whether a double resolve (or reject) can occur. Here, information such as a happens-before relation could prove useful.

Unnecessary Promise. An *unnecessary promise* is a promise whose sole purpose is to act as an intermediary between two other promises[5]. Consider the code fragment on the right. The programmer wants the function g to return a promise. In order to do so he constructs a fresh promise p1, then calls f which returns a promise p2, and finally registers a reaction on p2 such that when it is fulfilled, it fulfills p1. However, the programmer could have simply returned the promise from f. Furthermore, since only a fulfill reaction is registered, any errors (rejects) from the promise p2 are silently lost. We can detect this pattern by inspecting the promise graph and the syntax of the fulfill reaction: (a) The unnecessary promise is not resolved (or rejected) from anywhere else and (b) The function passed to then is of the form `x => resolve(x)` where resolve is associated with the unnecessary promise.

```
function g() {
  var p1 = new Promise((resolve, reject) => {
    var p2 = f(); // assume f returns a promise
    p2.then(x => resolve(x));
  });
  return p1;
}
```

Broken Promise Chain. A *broken promise chain* occurs when the programmer inadvertently creates a fork in a promise chain. Consider the code fragment on the right. Inside the function f, the programmer calls asyncOperation, which returns a promise p. The programmer then calls p.then to chain the complexOperation to occur once the async operation completes. At the end of the function, the programmer mistakenly returns p instead of the promise returned by the call to p.then. The consequence is that when the programmer later calls p2.then the promise p2 will not be resolved (or rejected) with the value of complexOperation as intended, but instead with the value from asyncOperation. The promise graph cannot outright detect such issues, since they could be legitimate behavior, but the programmer can use the promise graph to inspect where the value of p2 comes from, to detect the broken promise chain.

```
function f() {
    var p = asyncOperation();
    p.then(function() {
        return complexOperation();
    });
    return p;
}
var p2 = f()
p2.then(x => /* ... */ )
```

5 CASE STUDY

We conducted a case study of questions posted to the popular StackOverflow forum to better understand: (a) the types of problems that JavaScript programmers encounter that are related to JavaScript promises, and (b) how the promise graph can help such developers understand and debug their programs.

5.1 Methodology

At the time of writing, there are 5,275 questions tagged with both promise and javascript on StackOverflow. Of these, 17 were posted in the last 24 hours. As it turns out, the majority of these posts are not really about promises at all, but more about typical JavaScript misunderstandings. It

[5]This pattern is also sometimes called the *Explicit-Constructor-Anti-Pattern*, although that name is archaic.

Table 1. StackOverflow Questions about JavaScript Promises.

Question	Bug Type	Root Cause	Extra Info
Q29210234	Broken Promise Chain	Unintended fork in promise chain.	-
Q41268953	Dead Promise	A promise is neither resolved nor rejected.	-
Q41488363	Dead Promise	A promise is neither resolved nor reject on all paths.	-
Q41512144	Missing Return	A promise is lost.	HB
Q41699046	Missing Return	A reaction unintentionally returns `undefined`.	-
Q41764399	Other	A reaction unintentionally replaces an error.	-
Q41993302	Other	A reject reaction swallows an error.	-
Q42076519	Other	A promise is passed to `then` and ignored.	-
Q42163367	Broken Promise Chain	An unintended fork in promise chain.	LU
Q42210068	Missing Return	A reaction unintentionally returns `undefined`.	-
Q42304958	Double Resolve	A promise is resolved multiple times.	HB + EV
Q42343372	Other	A value is passed to `then` and ignored.	-
Q42391252	Broken Promise Chain	A reject reaction is registered too late.	-
Q42408234	Missing Return	A reaction unintentionally returns `undefined`.	-
Q42551854	Double Resolve	A promise is resolved multiple times.	HB + EV
Q42577647	Missing Return	A promise is lost.	-
Q42672914	Dead Promise	A promise is neither resolved nor rejected.	-
Q42719050	Missing Return	A reaction unintentionally returns `undefined`.	-
Q42777771	Double Resolve	A promise is resolved multiple times.	LU
Q42788603	Missing Return	A reaction unintentionally returns `undefined`.	-
Q42828856	Missing Return	A reaction unintentionally returns `undefined`.	-

is common to see programmers struggle with type errors, non-lexical scoping, closures, exceptions, recursion, and even syntax errors, and then mistakenly believe these errors are related to promises.

We manually examined the newest 600 StackOverflow questions to find posts about promises where the promise graph could be useful for debugging. Specifically, we looked for posts that met the following criteria: (a) The post contained a code fragment with a discussion of expected or incorrect behavior, (b) The post had an answer, (c) The code fragment was not too long, at most 100 lines, (d) The root cause had to be related to promises, and (e) The post should not be a duplicate.

The case study was carried out by examining the StackOverflow post, looking at the code, manually constructing the promise graph, and inspecting it for errors, as discussed in Section 4.

5.2 Summary of Results

Table 1 summarizes the results of our case study. The first column is the StackOverflow identifier for the question. The second column is the bug type, as discussed in the previous section. The third column categorizes each question according to our subjective understanding of the root cause of the issue. The labels shown in the fourth column indicate what other information, in addition to the promise graph, was necessary to debug the issue. These labels should be interpreted as follows:

- HB *happens-before* information is necessary to debug the root cause (e.g., the knowledge that one statement in the program always executes before another).
- LU *loop-unrolling* is necessary to debug the root cause (e.g., the knowledge that a promise created in one iteration of a loop affects a promise created in the next iteration of the loop).
- EV *event reasoning* is necessary to debug the root cause (e.g., the knowledge that an event handler for a button can be fired multiple times).

In cases where no label is listed in the third column, the promise graph by itself was sufficient to understand and debug the issue.

5.3 Debugging using the Promise Graph

We now discuss how the promise graph can be used as an aid to understand and debug a representative subset of the StackOverflow questions listed in Table 1. The reader is referred to Appendix A for a discussion of the remaining StackOverflow questions.

Q41268953. The programmer creates a promise, but never resolves or rejects it. As a result, the reactions associated with the promise are never executed. The programmer gets lost in the details and mistakenly believes the bug to be elsewhere in the program. He or she then proceeds to add additional logging and reject reactions in all the wrong places and these are never executed. In this scenario, the promise graph immediately identifies that the initial promise in the promise chain is never resolved or rejected, and hence that no reaction on chain is ever executed.

Q41488363. The programmer returns a promise from a function meant to perform authorization. The authorization itself is delegated to another function that takes a callback. Inside this callback, the programmer inspects the result of the authorization and resolves or rejects the promise based on whether it was successful. However, on one path in the callback function where authorization fails, the programmer has inserted an early return to exit the function. Unfortunately, this leaves the promise in an unfulfilled state because the promise is not rejected, and hence the rest of the promise chain never executes. When this happens, the program gets stuck instead of reporting an error. In this case, the promise graph combined with happens-before information reveals that it is possible to return from the callback without the promise being resolved or rejected.

Q41699046. The programmer returns a promise from a function. The outcome of this promise is determined by a "die toss"; if the die shows six, then the promise is resolved, otherwise it is rejected. The programmer creates a promise chain of die tosses. In the middle of the chain, the programmer calls `then` to register two logging functions, `logSuccess` and `logFailure`, as reactions. Unexpectedly, the subsequent reactions in the promise chain report that the thrown die is `undefined`. The problem is that the two logging functions do not return the outcome of the die toss, but instead implicitly return `undefined` which is then used to resolve/reject the dependent promise. The promise graph shows that the dependent promise is resolved/rejected with an `undefined` value returned by `logSuccess` or `logFailure`.

Q41764399. The programmer creates a promise chain. Each reaction on this chain can fail for a different reason. To track which reaction failed, the programmer creates a series of errors objects for each type of error. At each step of the promise chain, the programmer registers a reject reaction to check if the previous operation failed and to return the appropriate error object. However, this does not achieve the desired effect. The problem is that each error object ends up masking the previous error. When an operation fails, the promise rejects with the correct error object, but this causes the next reject reaction to reject with its own error object, which in turn causes the next reject reaction to reject with its own error object, and so on. The result is that the entire promise chain always rejects with the last error object. In this case, the promise graph shows that each promise in the chain is rejected with the error object from the previous reject reaction.

Q41993302. The programmer creates a promise chain with a reject reaction. The reaction logs the error message, but does not return or throw any value. As a consequence, the reject reaction implicitly returns `undefined`, which causes the dependent promise to be (successfully) resolved. Hence, later in the promise chain, the error is unexpectedly lost and the `undefined` value appears. In this case, the promise graph shows that the reject reaction causes the dependent promise to be resolved with `undefined`.

Q42210068. The programmer creates a promise chain. In one of the fulfill reactions, the programmer returns the result of an expression of the form `someArray.forEach(f)`. The callback function `f` returns a promise. However, the `Array.prototype.forEach` function discards any value returned by the callback function `f` and always returns `undefined`. Hence, the promise is resolved with `undefined` instead of with the value expected by the programmer. In this case, the promise graph shows that the promise is resolved with the value `undefined` and that the value and promise inside the callback are lost.

Q42343372. The programmer creates a promise chain and passes the result of a function call into `then` instead of passing the function itself. The `then` operation expects two functions (the resolve and reject reactions), and when supplied with a value it simply discards it and uses the identity function. The result is that the function is executed too early and that the rest of the promise chain observes an unexpected value. The promise graph can help identify this problem by showing where the unexpected value originates and that the function is *not* registered as a reaction on the promise.

Q42391252. The programmer creates a promise chain and asks why a reject reaction is handling an error from one of the earlier promises. The problem is that the reject reaction is registered too late in the chain. If the programmer only wants to handle an error specific to a certain promise, then the reject reaction must be registered on that promise and not later in the chain. The promise graph can help debug this problem by showing the full promise chain and where on the chain each reject reaction is registered.

Q42577647. The programmer constructs a factory for delayed promises, i.e., promises that resolve after some specific timeout. Inside the factory, the programmer calls `setTimeout(f)` passing a function `f` that returns a fresh promise. However, `setTimeout` discards any value returned by the callback function `f` and as a consequence the fresh promise is lost and never linked to the current promise. The promise graph shows that (a) the value flowing into the fresh promise is lost and (b) the current promise is implicitly resolved with the value `undefined`.

Q42719050. The programmer creates a promise chain. Inside one of the later resolve reactions, the programmer passes a function `f` into the callback of a database operation `connection.query(f)`. Inside function `f`, the programmer performs various computations and then returns the computed result. The programmer expected this result to be passed down the chain. However, since the return happened inside the callback function `f`, and not in the reaction function, the value is simply discarded, and the reaction implicitly returns `undefined` which is then used to resolve the dependent promise. Here the problem is a matter of scoping: the programmer lost track of the fact that the return occurred inside a callback instead of inside a reaction. The promise graph can help pinpoint this problem by showing that the dependent promise is resolved with the `undefined` value, and not the value from inside the callback function `f`.

Discussion. As we have seen, reasoning about promises is a complicated problem that many JavaScript programmers struggle with. We have proposed the promise graph as an aid to understand and debug the intricate control- and data-flow of promises. The case study has shown that the promise graph often provides sufficient information to find the root cause of the issue. However, some ingenuity by the programmer is still required, and in some cases other information is needed too, for example the order in which statements can execute.

6 RELATED WORK

In this paper, we have used the term *promise* for an object that represents the result of an asynchronous operation. Other terms for the same concept are *future*, *delay*, and *deferred*. For simplicity, we will continue to use the term *promise* in our discussion of related work, although some authors use the other terms.

Promises. Promises were originally proposed by Friedman and Wise [1976]. Flanagan and Felleisen [1995, 1999] present several operational semantics for promises in the context of functional programming languages. A key contribution of their work is a program analysis to detect and eliminate redundant synchronization points, i.e., to avoid synchronization between threads for promises that are known to have already completed. In languages such as Java, where execution is multi-threaded and different threads may access and mutate the same memory location, the use of promises can lead to race conditions and other concurrency errors. Welc et al. [2005] present *safe futures*, an API and programming model for safely using futures in Java. A key property of their work is that a safe program, although executed concurrently, behaves as if it were executed sequentially. Unlike Java, JavaScript is inherently single-threaded, i.e., execution is non-preemptive and restricted to one thread. However, due to external events, e.g., the user pressing a button, it is non-deterministic when a promise is resolved (or rejected) and hence when its reactions are scheduled for execution. This means that, even with one thread, it is possible to have multiple different interleavings of promise reactions. There has been notable recent work on the detection of event races in JavaScript (see, e.g., Jensen et al. [2015]; Mutlu et al. [2015]; Petrov et al. [2012]; Raychev et al. [2013]), but, to our knowledge, none of this work explicitly considers promises.

Semantics. Maffeis et al. [2008] proposed one of the first formal semantics for the JavaScript language. Their semantics was formulated as a small-step operational semantics capturing ECMAScript (3rd Edition). Later, Guha et al. [2010] proposed $\lambda_{\mathcal{JS}}$, a reduction-style semantics for a "core" JavaScript language. The key idea was to distill JavaScript into a small collection of constructs forming the core calculus. In this way, many of the complex features of JavaScript could be handled by translation into these more basic operations. Madsen et al. [2015] presented an extension of $\lambda_{\mathcal{JS}}$ to formalize the registration and execution of event listeners in JavaScript. A part of this work discusses common programming errors related to events in Javascript, including *lost events* and *dead listeners*. In this work, we have shown how to extend the $\lambda_{\mathcal{JS}}$ calculus with promises, and how some of the high-level operations of ECMAScript 6 can be emulated by the basic constructs of λ_p.

Static Analysis. Guarnieri and Livshits [2009] proposed one of the first static analyses for JavaScript, a points-to analysis to detect potential harmful code in JavaScript widgets. Jensen et al. [2009] describe TAJS (Type Analysis for JavaScript), a whole-program dataflow and points-to analysis for JavaScript with the goal of finding type related errors. Later, this work was extended with support for browser-based applications [Jensen et al. 2011], jQuery [Andreasen and Møller 2014], and handling `eval` [Jensen et al. 2012]. Madsen et al. [2015] presents a static analysis for detecting event-related bugs, including *dead listeners* and *dead promises*, as discussed above. In this work, we have sketched how the technique by Horn and Might [2010] can be used to automatically derive a static analysis from a reduction semantics. The key idea behind this technique is to derive a series of abstract machines, starting from the control-environment-continuation (CEK) machine, and through a series of other machines, until a final "abstract" abstract machine is obtained. The most important step is the allocation of environments and continuations in the heap, and then approximation of the heap to ensure that the analysis terminates.

7 CONCLUSIONS AND FUTURE WORK

Recently, ECMAScript 6 added support for *promises*, an alternative mechanism for managing asynchrony that enables programmers to chain asynchronous computations while supporting proper error handling. Unfortunately, JavaScript promises are complex and error-prone in their own right, as is evident from a classification of promise-related errors that we compiled from a manual study of 600 issues related to promises that were reported on the StackOverflow forum. From this study, we concluded that programmers would benefit from techniques that can reason about the correctness of promise-based code.

This paper presents a foundational step towards this goal, by presenting λ_p, a core calculus that captures the essence of ECMAScript 6 promises. Based on λ_p, we introduced the *promise graph*, a program representation that can assist programmers with the debugging of promise-based code. We then reported on a case study in which we investigated how the promise graph can be helpful for debugging errors in promise-based code fragments reported on the StackOverflow forum.

As future work, we plan to derive static analyses from λ_p that compute sound over-approximations of the promise graph using the mechanical technique of Abstracting Abstract Machines (AAM) of Horn and Might [2010]. In this context, soundness means that the computed promise graph captures *all possible* behaviors of the program under analysis. The over-approximate promise graph must capture (i) all possible flows of values into promises via `resolve`/`reject`, (ii) all possible reaction registrations via `onResolve`/`onReject`, and (iii) all possible promise dependencies created via `link`. To apply the AAM technique, we must select an abstract domain to model each component of the state in the dynamic semantics that was presented in Section 3. Here, the choice of abstract domains will determine the precision and cost of the static analysis. In addition, we plan to extend the abstract state with a representation of the promise graph, and instrument the semantics to record the relevant operations on promises in that graph. From this, the AAM approach then mechanically derives static transfer functions that are guaranteed to be sound by construction so that the resulting static analysis soundly over-approximates the runtime behavior of the program and generates a sound over-approximation of the promise graph.

A CASE STUDY (CONTINUED)

Q41512144. The programmer presents a complicated program fragment with four levels of nested promises and five uses of `then` operation. The programmer explains that this a simplification of a larger piece of the code. In this specific piece of the code, the programmer is expecting one value to be passed to the last reaction in the promise chain, but is observing an earlier value from the chain. Looking at the code fragment, with its many levels of nesting and number of promises, it is hard to understand the flow of values. The promise graph, however, reveals that in one of the last calls to `then` a (resolved) promise is being passed, but in actuality `then` expects a *function* which either *returns* a regular value or a promise value. Passing the promise object directly to `then` has no effect and is silently ignored[6], hence the value of that promise is lost, and the previous value is used. The promise graph identifies this problem by showing that the value flowing into the promise is lost and more specifically that it does not flow into some other promise as expected.

Q42076519. The programmer creates two promises p_1 and p_2. Each promise is resolved after a specified timeout. The programmer then writes `p1.then(p2).then(f)` expecting that p_1 will be resolved followed by p_2 and that the value of p_2 will be passed to f. However, the `then` operation requires one or two function arguments, i.e. the fulfill and reject reactions. Passing a promise to `then` is silently ignored and treated as if the identity function was passed. Consequently, there is

[6]Technically it is treated as if the identity function was passed.

there is no causal relationship between p_1 and p_2. The promise graph shows that the promises p_1 and p_2 are not related and that the value passed to f originates from p_1.

Q42163367. The programmer asks why two program fragments do not exhibit the same behavior. In the first fragment, the programmer has created a promise chain with a "delayed promise" inserted between every fulfill reaction. In the second fragment, the programmer is trying to construct the same promise chain, but by using a loop. Unexpectedly, this code creates a tree of promises instead of a single chain. The problem is that at each iteration of the loop, the dependent promise is constructed from the first promise in the chain rather than from the promise created in the previous iteration of the loop. The promise graph, if instantiated in a loop-unrolled version, shows that the first program gives rise to a chain, whereas the second program gives rise to a tree.

Q42304958. The programmer creates a single global promise. The promise is resolved upon the completion of an AJAX request which is triggered by the user submitting a form. As the programmer reports, this only works the first time the form is submitted. The problem is that a promise can only ever be resolved or rejected *once*. The programmer should create a new promise for every AJAX request. The promise graph can help identify this problem by showing that the same promise is resolved multiple times.

Q42408234. We discussed this StackOverflow question in the motivation (see Section 2).

Q42551854. The programmer has a web page with two buttons: "success" and "failure". When a button is pressed, a promise is either resolved or rejected, respectively. The promise has a resolve and a rejection reaction to log to the console when something happens. The programmer asks why nothing is printed to the console the second time either button is pressed. The problem is that a promise can only be resolved or rejected once. Resolving or rejecting a promise a second time has no effect. The promise graph can help identify this problem by showing that the promise is resolved multiple times.

Q42672914. The programmer constructs a promise and performs various application-specific tasks inside the promise constructor. In one case the promise is rejected, but there is no path on which the promise is resolved. Hence the promise (and its chain) is dead. In this case, the promise graph shows that the promise is only ever rejected, but never resolved.

Q42777771. The programmer creates a promise and resolves it inside a loop. However, only the first call to `resolve` has any effect; the subsequent calls to `resolve` have no effect and their values are lost. They are effectively dead code. In this case, the promise graph can show that the promise is resolved multiple times.

Q42788603. The programmer creates a promise chain. In each resolve reaction the programmer performs some computation, but forgets to return any value, and consequently `undefined` is implicitly returned. Hence each dependent promise is resolved with `undefined`. The promise graph shows this situation.

Q42828856. The programmer creates a promise chain. In each resolve reaction the programmer performs an asynchronous operation and returns a promise. Except in the last reaction, where the programmer has forgotten the return. Hence the program runs correctly, except the very last promise is implicitly resolved with `undefined`. In this case, the promise graph shows that the value of the pen-ultimate promise is lost, and that the last promise is resolved with `undefined`.

Q29210234. The programmer asks why two programmer fragments do not exhibit the same behavior. The answer is that in the first fragment the programmer (correctly) creates a single promise chain. But in the second fragment, the programmer unintentionally creates a tree of promises, hence a promise is resolved not with the value of the "previous" resolve reaction, as expected, but with the value of the root. The promise graph shows this difference in structure.

B SYNTAX OF λ_{JS}

$$
\begin{array}{rcll}
c \in \mathit{Cst} & = & \mathit{bool} \mid \mathit{num} \mid \mathit{str} \mid \texttt{null} \mid \texttt{undef} & \text{[constant]} \\
\\
v \in \mathit{Val} & = & c & \text{[literal]} \\
 & \mid & a & \text{[address]} \\
 & \mid & \{\mathit{str} : v \cdots\} & \text{[object]} \\
 & \mid & \lambda\,(x \cdots)\ e & \text{[function]} \\
\\
e \in \mathit{Exp} & = & v & \text{[value]} \\
 & \mid & x & \text{[variable]} \\
 & \mid & e\ ;\ e & \text{[sequence]} \\
 & \mid & e = e & \text{[assignment]} \\
 & \mid & \texttt{let}\ (x = e)\ e & \text{[binding]} \\
 & \mid & e\,(e \cdots) & \text{[call]} \\
 & \mid & e.f & \text{[field load]} \\
 & \mid & e.f = e & \text{[field store]} \\
 & \mid & \texttt{ref}\ e & \text{[address of]} \\
 & \mid & \texttt{deref}\ e & \text{[value at]} \\
\\
x \in \mathit{Var} & = & \text{is a finite set of variable names.} & \\
f \in \mathit{Fld} & = & \text{is a finite set of field names.} & \\
a \in \mathit{Addr} & = & \text{is an infinite set of memory addresses.} & \\
\lambda \in \mathit{Lam} & = & \text{is the set of all lambda expressions.} &
\end{array}
$$

Fig. 8. Syntax of λ_{JS}.

ACKNOWLEDGMENTS

This research was supported by the Natural Sciences and Engineering Research Council of Canada. This research was supported in part by NSF grant CCF-1715153.

REFERENCES

Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. 2017. Repairing Event Race Errors by Controlling Nondeterminism. In *Proc. 39th International Conference on Software Engineering (ICSE).*

Esben Andreasen and Anders Møller. 2014. Determinacy in static analysis for jQuery. In *Proc. 29th ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages, and Applications (OOPSLA).*

Petka Antonov. 2013. bluebird. https://github.com/petkaantonov/bluebird. (2013). Accessed: 2016-10-27.

ECMA. 2015. 262: ECMAScript Language Specification. *European Association for Standardizing Information and Communication Systems (ECMA)* (2015).

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex.* The MIT Press.

Cormac Flanagan and Matthias Felleisen. 1995. The Semantics of Future and its use in Program Optimization. In *Proc. 22nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

Cormac Flanagan and Matthias Felleisen. 1999. The Semantics of Future and an Application. *Journal of Functional Programming* (1999).

Daniel Friedman and David Wise. 1976. The Impact of Applicative Programming on Multiprocessing. In *International Conference on Parallel Processing*.

Salvatore Guarnieri and Benjamin Livshits. 2009. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *Proc. 18th Usenix Security Symposium*.

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. In *Proc. 24th European Conference on Object-oriented Programming (ECOOP)*.

Shin Hong, Yongbae Park, and Moonzoo Kim. 2014. Detecting Concurrency Errors in Client-Side Java Script Web Applications. In *Proc. 7th IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *Proc. 15th ACM International Conference on Functional Programming (ICFP)*.

Casper Svenning Jensen, Anders Møller, Veselin Raychev, Dimitar Dimitrov, and Martin T. Vechev. 2015. Stateless Model Checking of Event-Driven Applications. In *Proc. 30th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Simon Holm Jensen, Peter Jonsson, and Anders Møller. 2012. Remedying the Eval That Men Do. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*.

Simon Holm Jensen, Magnus Madsen, and Anders Møller. 2011. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In *Proc. 8th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium (SAS)*.

Vineeth Kashyap, Kyle Dewey, Ethan Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A Static Analysis Platform for JavaScript. In *Proc. 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*.

Kris Kowal. 2010. Q. https://github.com/kriskowal/q. (2010). Accessed: 2016-10-27.

Magnus Madsen and Esben Andreasen. 2014. String Analysis for Dynamic Field Access. In *Proc. 23rd International Conference on Compiler Construction (CC)*.

Magnus Madsen, Benjamin Livshits, and Michael Fanning. 2013. Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries. In *Proc. European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

Magnus Madsen and Anders Møller. 2014. Sparse Dataflow Analysis with Pointers and Reachability. In *Proc. 21st International Static Analysis Symposium (SAS)*.

Magnus Madsen, Frank Tip, and Ondřej Lhoták. 2015. Static Analysis of Event-driven Node.Js JavaScript Applications. In *Proc. ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2008. An Operational Semantics for JavaScript. In *Asian Symposium on Programming Languages and Systems*.

Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. 2015. Detecting JavaScript Races that Matter. In *Proc. 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*.

Boris Petrov, Martin T. Vechev, Manu Sridharan, and Julian Dolby. 2012. Race Detection for Web Applications. In *Proc. 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Veselin Raychev, Martin T. Vechev, and Manu Sridharan. 2013. Effective Race Detection for Event-Driven Programs. In *Proc. 28th ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages, and Applications (OOPSLA)*.

Adam Welc, Suresh Jagannathan, and Antony Hosking. 2005. Safe Futures for Java. *Proc. 20th ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages, and Applications (OOPSLA)* (2005).

Lu Zhang and Chao Wang. 2017. RClassify: Classifying Race Conditions in Web Applications via Deterministic Replay. In *Proc. 39th International Conference on Software Engineering (ICSE)*.

Yunhui Zheng, Tao Bao, and Xiangyu Zhang. 2011. Statically Locating Web Application Bugs Caused by Asynchronous Calls. In *Proc. 20th International Conference on World Wide Web (WWW)*.