Characterizing and refactoring asynchronous JavaScript callbacks

by

Keheliya Gallaba

B.Sc. Eng. (Hons), University of Moratuwa, 2011

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Applied Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

December 2015

Abstract

Modern web applications make extensive use of JavaScript, which is now estimated to be one of the most widely used languages in the world. Callbacks are a popular language feature in JavaScript. However, they are also a source of comprehension and maintainability issues. We studied several features of callback usage across a large number of JavaScript applications and found out that over 43% of all callback- accepting function call sites are anonymous, the majority of callbacks are nested, and more than half of all callbacks are invoked asynchronously.

Promises have been introduced as an alternative to callbacks for composing complex asynchronous execution flow and as a robust mechanism for error checking in JavaScript. We use our observations of callback usage to build a developer tool that refactors asynchronous callbacks into Promises. We show that our technique and tool is broadly applicable to a wide range of JavaScript applications.

Preface

The work presented in this thesis was conducted by the author under the supervision of Professor Ali Mesbah and Professor Ivan Beschastnikh.

I was responsible for implementing the tools, running the experiments, evaluating and analyzing the results, and writing the manuscript. My supervisors guided me with the creation of the experimental methodology, and the analysis of results, as well as editing and writing portions of the manuscript. Quinn Hanam and Amin Milani Fard also helped me by editing and writing portions of the manuscript.

The results of the empirical study of JavaScript callbacks were published in the Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)[26] in 2015. We are also proud to announce that our paper received the **Best Full Paper Award** at ESEM 2015.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank my advisors Ivan Beschastnikh and Ali Mesbah for their invaluable support. This work would not have been possible without the guidance, motivation, inspiration, and support provided by them.

I would also like to thank my family specially my wife, Thamali, My sister, Sathya and my parents for their unlimited support and love. Thank you for encouraging me in all of my endeavors and inspiring me to follow my dreams. Knowing that you always wanted the best for me helped me to stay motivated.

To my friends and colleagues in SALT Lab, thank you for listening, offering me advice, and supporting me in various ways through this entire program.

Last but not least, I would also like to thank Prof. Karthik Pattabirman for accepting to be a part of my defence committee.

Chapter 1

Introduction

Callbacks, or higher-order functions, are available in many programming languages, for instance, as function-valued arguments (e.g., Python), function pointers (e.g., C++), and lambda expressions (e.g., Lisp). In this thesis we study JavaScript callbacks since JavaScript is the dominant language for building web applications. For example, a recent survey of more than 26K developers conducted by Stack Overflow found that JavaScript is the most-used programming language [56]. The callback language feature in JavaScript is an important factor to its success. For instance, JavaScript callbacks are used to responsively handle events on the client-side by executing functions asynchronously. And, in Node.js[1], a popular JavaScript-based framework, callbacks are used on the server-side to service multiple concurrent client requests.

Listing 1.1 illustrates three common challenges with callbacks. First, the three callbacks in this example (on lines 2, 3, and 5) are *anonymous*. This makes the callbacks difficult to reuse and to understand as they are not descriptive. Second, the callbacks in the example are *nested* to three levels, making it challenging to reason about the flow of control in the code. Finally, in line 5 there is a call to `conn.query`, which invokes the second callback parameter *asynchronously*. That is, the execution of the inner-most anonymous function (lines 6–7) is deferred until some later time. As a result, most of the complexity in this small example rests on extensive use of callbacks; in particular, the use of anonymous, nested, and

[1] https://nodejs.org

```
var db = require('somedatabaseprovider');
http.get('/recentposts', function(req, res){
    db.openConnection('host', creds, function(err, conn){
        res.param['posts'].forEach(function(post) {
            conn.query('select * from users where id=' + post['user'],
                function(err, results){
                    conn.close();
                    res.send(results[0]);
            });
        });
    });
});
```

Listing 1.1: A representative JavaScript snippet illustrating the comprehension and maintainability challenges associated with nested, anonymous callbacks and asynchronous callback scheduling.

asynchronous callbacks.

Though the issues outlined above have not been studied in detail, they are well-known to developers. Searching for "callback hell" on the web brings up many articles with best practices on callback usage in JavaScript. For example, a prominent problem with asynchronous callbacks in JavaScript is error-handling. In JavaScript, an error that occurs during the execution of an asynchronous task cannot be handled with the traditional `try/catch` mechanism because the asynchronous task is run outside the existing call stack. For example, consider the code in Listing 1.2.

```
try {
  setTimeout(function() {
              throw new Error("Uh oh!");
            }, 2000);
} catch (e) {
  console.log("Caught the error: " + e.message);
}
```

Listing 1.2: A JavaScript snippet illustrating that try/catch statements are ineffective for handling errors in asynchronous callbacks.

An exception generated during `setTimeout` will not be caught inside the `catch` block. Therefore, an error generated by an asynchronous function, such as `setTimeout`, can only be handled by passing it as a parameter to the callback function. The JavaScript community has come up with a convention for error propagation in asynchronous contexts called the *error-first protocol*. In this idiom-based protocol, the first parameter of the callback is reserved for communicating errors and the other parameters are used for passing data.

```
fs.readFile('/foo.txt', function(err, result) {
  if (err) {
    console.log('Unknown Error');
    return;
  }
  console.log(result);
});
```

Listing 1.3: A JavaScript snippet illustrating the error-first protocol.

Consider Listing 1.3: an error generated by the asynchronous function `fs.readFile` is passed to the callback as an argument (`err`) and the callback must include appropriate error-handling code. A key limitation of the error-first protocol is that it is merely a convention and developers are not obligated to use it. As a result, developers manually check to see whether a function follows the protocol, which makes it error-prone. As this is a best practice and adhering to this protocol is optional, it is also not clear to what extent developers use it in practice.

Promises are a new feature of ECMAScript6 that are designed to help with the error handling and nesting problems associated with asynchronous callbacks. The ECMAScript6 specification has been approved [7] in 2015 and all the major JavaScript runtimes support promises [10]. Promises explicitly register handlers for executions that are successful and executions that produce errors. This removes the need for the error-first convention and separates the success handler from the error handler. Promises can also be chained together, which flattens nested callbacks and makes them easier to understand.

1.1 Objectives

The main objective of our research is twofold:

- **Gaining an understanding of JavaScript callback usage in practice**
  Although callbacks are a key JavaScript feature, they have not received much attention in the research community. We think that this is a critical omission

as the usage of callbacks is an important factor in developer comprehension and maintenance of JavaScript code.

- **Devising automated JavaScript refactoring techniques to mitigate callback-related challenges**
  New JavaScript language features, such as Promises, are being proposed to help with problems associated with asynchronous callbacks. But there is no mention or use of refactoring tools in the developer community to transform exisiting callbacks to Promises. We investigate the feasibility of a novel automated JavaScript refactoring technique for this purpose.

1.2 Contributions

This thesis makes the following main contributions:

- A systematic methodology and tool, for analyzing JavaScript code statically to identify callbacks and to measure their various features (Section 4.3)
- An empirical study to characterize JavaScript callback usage across 138 large JavaScript projects. These include 86 Node.js modules from the NPM public package registry used in server-side code and 62 subject systems from a broad spectrum of categories, such as JavaScript MVC frameworks, games, and data visualization libraries. We found that on average, every 10th function definition takes a callback argument, and that over 43% of all callback-accepting function callsites are anonymous. Furthermore, the majority of callbacks are nested, more than half of all callbacks are asynchronous, and asynchronous callbacks, on average, appear more frequently in client-side code (72%) than server-side (55%). (Section 4.4)
- A discussion of the implications of our empirical findings
- An exploratory study in which we search for and examine several GitHub issues and pull-requests containing terms related to refactoring of asynchronous callbacks into promises. We found that developers frequently want to refactor existing code that uses asynchronous callbacks into code that uses promises (GitHub search returned over 4K issues related to this topic). GitHub search

returned only 451 pull-requests related to this topic (a small number of actual transformations as compared to the number of requests). This study provides support for the utility of an automated refactoring tool (Section 5.2)

- A set of static analysis techniques and a tool that support automated refactoring by: (1) discovering instances of asynchronous callbacks and (2) transforming instances of asynchronous callbacks into promises. (Section 5.3)
- An evaluation of the refactoring tool pointing to the real-world relevance and efficacy of our techniques. (Section 5.6)

1.3 Thesis Organization

In Chapter 2 of this thesis, we provide background information regarding JavaScript applications, particularly with respect to the use of callbacks in JavaScript, along with the motivation to conduct this research. Chapter 3 discusses the related work in this area of study. Chapter 4 describes in detail the empirical srudy we carried out to characterize the real-world use of callbacks in different types of JavaScript programs, the results found from this study, and the implications these results can have with respect to web developers, tool developers and the research community. In Chapter 5, we present the tool we developed to detect instances of asynchronous callbacks and to refactor such callbacks. The evaluation results of the tool on a wide range of JavaScript applications is also presented in this chapter. Finally, Chapter 6 concludes our work and presents future research directions.

Chapter 2

Background

A callback is a function that is passed as an argument to another function, which is expected to invoke it either immediately or at some point in the future. Callbacks can be seen as a form of the continuation-passing style (CPS) [57], in which control is passed explicitly in the form of a continuation; in this case the callback passed as an argument represents a continuation.

Synchronous and asynchronous callbacks. There are two types of callbacks. A callback passed to a function f can be invoked *synchronously* before f returns, or it can be deferred to execute *asynchronously* some time after f returns.

JavaScript uses an event-driven model with a single thread of execution. Programming with callbacks is especially useful when a caller does not want to wait until the callee completes. To this end, the desired non-blocking operation is scheduled as a callback and the main thread continues its synchronous execution. When the operation completes, a message is enqueued into a task queue along with the provided callback. The event loop in JavaScript prioritizes the single thread to execute the call stack first; when the stack is empty, the event loop dequeues a message from the task queue and executes the corresponding callback function. Figure 2.1 illustrates the event loop model of JavaScript for a non-blocking HTTP `get` call with a callback named `cb`.

Named and anonymous callbacks. JavaScript callbacks can be *named* functions or *anonymous* functions (e.g., lines 2, 3, or 5 of Listing 1.1). Each approach has its tradeoffs. Named callbacks can be reused and are easily identified in stack traces

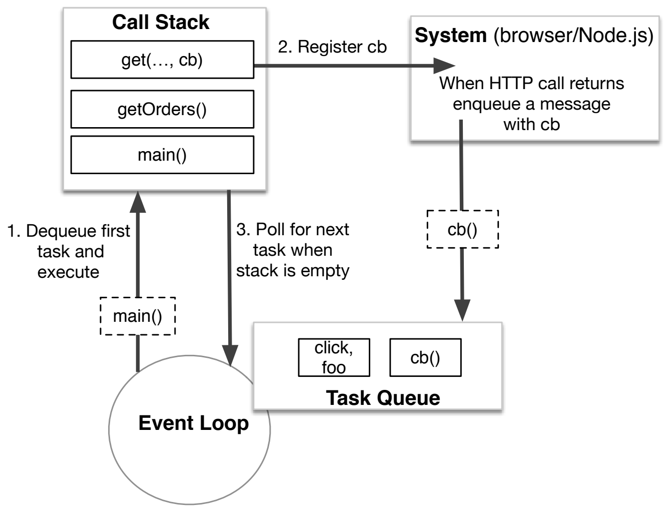

Call Stack
get(…, cb)
getOrders()
main()
2. Register cb
System (browser/Node.js)
When HTTP call returns enqueue a message with cb
cb()
1. Dequeue first task and execute
3. Poll for next task when stack is empty
main()
click, foo
cb()
Task Queue
Event Loop

Figure 2.1: The JavaScript event loop model.

or breakpoints during debugging activities. However naming causes the callback function to persist in memory and prevents it from being garbage collected. On the other hand, anonymous callbacks are more resource-friendly because they are marked for garbage collection immediately after being executed. However, anonymous callbacks are not reusable and may be difficult to maintain, test, or debug.

Nested callbacks. Developers often need to combine several callback-accepting functions together to achieve a certain task. For example, two callbacks have to be nested if the result of the first callback needs to be passed into the second callback in a non-blocking way (see lines 2-8 in Listing 2.2). This structure becomes more complex when the callbacks need to be conditionally nested. Control-flow composition with nested callbacks increases the complexity of the code. The term 'callback hell' [49] has been coined by developers to voice their common frustration with this complexity.

Error-first callbacks. In synchronous JavaScript code the `throw` keyword can be used to signal an error and `try/catch` can be used to handle the error. When there is asynchrony, however, it may not be possible to handle an error in the context it is thrown. Instead, the error must be propagated asynchronously to an error handler in a different context. Callbacks are the basic mechanism for delivering errors asynchronously in JavaScript. Because there is no explicit language support for asynchronous error signaling, the developer community has proposed a convention — dedicate the first argument in the callback to be a permanent place-holder for error signaling.

More exactly, the *error-first callback* protocol specifies that during a callback invocation, either the error argument is non-null (indicating an error), or the first argument is null and the other arguments contain data (indicating success), but not both [1]. Listing 2.1 shows an example of this protocol. If an error occurs while reading the file (Line 4), the anonymous function will be called with error as the first argument. For this error to be handled, it will be propagated by passing it as the first argument of the callback (in line 5). The error can then be handled at a more appropriate location (lines 17–21). When there is no error, the program continues (line 8) and invokes the callback with the first (error) argument set to null.

```
 1  var fs = require('fs');
 2  // read a file
 3  function read_the_file(filename, callback) {
 4      fs.readFile(filename, function (err, contents) {
 5          if (err) return callback(err);

 7          // if no error, continue
 8          read_data_from_db(null, contents, callback);
 9      });
10  }

12  function read_data_from_db(err, contents, callback) {
13      //some long running task
14  }

16  read_the_file('/some/file', function (err, result) {
17      if (err) {
18          //handle the error
19          console.log(err);
20          return;
21      }
22      // do something with the result
23  });
```

Listing 2.1: Error-first callback protocol.

The error-first callback protocol is intended to simplify exception handling for developers; if there is an error, it will always propagate as the first argument through the API, and API clients can always check the first argument for errors.

But, there is no automatic checking of the error-first protocol in JavaScript. It is therefore unclear how frequently developers adhere to this protocol in practice.

Handling callbacks. A quick search in the NPM repository[1] reveals that there are over 4,500 modules to help developers with asynchronous JavaScript programming. Two solutions that are gaining traction in helping developers handle callbacks are libraries, such as *Async.js* [40] and new language features such as *Promises* [12]. We now briefly detail these two approaches.

The Async.js library exposes an API to help developers manage callbacks. For example, Listing 2.2 shows how nested callbacks in vanilla JavaScript (lines 1–8) can be expressed using the `waterfall` method available in Async.js (11–18).

Promises are a JavaScript language extension. A Promise-based function takes some input and returns a promise object representing the result of an asynchronous operation. A promise object can be queried by the developer to answer questions like "were there any errors while executing the async call?" or "has the data from the async call arrived yet?" A promise object, once fulfilled, can notify any function that depends on its data. Listing 2.2 illustrates how nested callbacks in vanilla JavaScript (lines 1–8) can be re-expressed using Promises (20–24).

Promises are described further in detail in the Section 5.1.

[1] https://www.npmjs.com/

```
// Before: nested callbacks
$("#button").click(function() {
    promptUserForTwitterHandle(function(handle) {
        twitter.getTweetsFor(handle, function(tweets) {
            ui.show(tweets);
        });
    });
});

// After: Using Async.js waterfall method
$("#button").click(function() {
    async.waterfall([
        promptUserForTwitterHandle,
        twitter.getTweetsFor,
        ui.show
    ]
    , handleError);
});

// After: sequential join of callbacks with Promises
$("#button").clickPromise()
    .then(promptUserForTwitterHandle)
    .then(twitter.getTweetsFor)
    .then(ui.show);
```

Listing 2.2: Rewriting nested callbacks using Async.js or Promises.

Chapter 3

Related Work

Callback-related issues are a recurrent discussion topic among developers [49]. However, to the best of our knowledge, there have been no empirical studies of callback usage in practice.

JavaScript applications. The dynamic behaviour of JavaScript applications was studied by Richards et al. [52]. They found that commonly made assumptions about dynamism in JavaScript are violated in at least some real-world code. A similar study was conducted by Martinsen et al. [39]. Richards et al. [53] studied the prevalence of `eval`. They found `eval` to be pervasive, and argued that in most usage scenarios, it could be replaced with equivalent and safer code or language extensions.

Ocariza et al. [47] conducted an empirical study to characterize root causes of client-side JavaScript bugs. Since this study, server-side JavaScript, on top of Node.js, has gained traction among developers. Our study considers callback usage in both client- and server-side JavaScript code.

Security vulnerabilities in JavaScript code have also been studied. Examples include studies on remote JavaScript inclusions [46, 62], cross-site scripting (XSS) [61], and privacy-violating information flows [33]. Parallelism in JavaScript code was studied by Fortuna et al. [25].

Milani Fard et al. [43] studied code smells in JavaScript code. In their list of JavaScript smells, they included nested callbacks, but only focus on callbacks in client-side code. Decofun [20] is a JavaScript function de-anonymizer. It parses the code and names any detected anonymous function according to its context.

Although JavaScript is a challenging language for software engineering, recent research advances have made the use of static analysis on JavaScript more practical [15, 23, 34, 35, 38, 45, 48, 55]. Other techniques mitigate the analysis challenges by using a dynamic or hybrid approach [14, 28, 60]. Others have considered to improve the core language through abstraction layers [59].

Asynchronous programming. Okur et al. [50] recently conducted a large-scale study on the usage of asynchronous programming in C# applications. They found that callback-based asynchronous idioms are heavily used, but new idioms that can take advantage of the `async/await` keywords are rarely used in practice. They have studied how developers (mis)use some of these new language constructs. Our study similarly covers the usage of callbacks and new language features such as Promises to enhance asynchronous programming. However, our work considers JavaScript code and delves deeper. For example, we characterize the usage of callback nesting and anonymous callbacks, which are known to cause maintenance problems.

Concurrency bug detection. EventRacer [51] detects data races in JavaScript applications. Zheng et al. [63] propose a static analysis method for detecting concurrency bugs caused by asynchronous calls in web applications. Similarly, WAVE [32] is a tool for identifying concurrency bugs by looking for the same sequence of user events leading to different final DOM-trees of the application.

Program comprehension. Clematis [13] is a technique for helping developers understand complex event-based and asynchronous interactions in JavaScript code by capturing low-level interactions and presenting those as higher-level behavioral models. Theseus [37] is an IDE extension that helps developers to navigate asynchronous and dynamic JavaScript execution. Theseus has some limitations; for example, it does not support named callbacks. Our study demonstrates that over half of all callbacks are named, indicating that many applications will be negatively impacted by similar limitations in existing tools.

JavaScript Refactoring Tools. The closest work to ours is by Brodu et al. [17] who propose a compiler for converting nested callbacks (or an imbrication of continuations) into a sequence of Dues, which is a simpler version of Promises. This compiler does not support the critical promise notions of rejection and fulfillment and re-writes the error-first protocol callbacks without using the error-handling body of the callback. In addition, there are several drawbacks to this approach: (1) the source code does not change, so it does not eliminate the issues with understandability, (2) Dues do not support the critical notions of rejection and resolution in promises and can therefore only re-write the error-first protocol in a simplified notation, (3) their approach requires developer intervention to specify asynchronous callbacks that can are suitable candidates to be converted. This approach also requires the developer to manually specify asynchronous callbacks to be converted into Promises. In contrast we introduce a tool which is completely automated.

A number of other JavaScript refactoring tools have been previously proposed. For example, Meawad et al. [41] proposed a tool to refactor `eval` statements into safer code, and Feldthaus et al. [21, 22] developed a technique for semi-automatic refactoring with a focus on renaming. None of these works focuses on detecting or refactoring asynchronous callbacks to Promises.

Chapter 4

Empirical Study

4.1 Methodology

To characterize callback usage in JavaScript applications, we focus on the following three research questions.

RQ1: How prevalent are callbacks?

RQ2: How are callbacks programmed and used?

RQ3: Do developers use external APIs to handle callbacks?

Our analyses are open source [4] and all of our empirical data is available for download [9].

4.2 Subject Systems

We study 138 popular open source JavaScript subject systems from six distinct categories: NPM modules (86), web applications (16), game engines (16), client-side frameworks (8), visualization libraries (6), and games (6). NPM modules are used only on the server-side. Client-side frameworks, visualization libraries, and games include only client-side code. Web applications and game engines include both client-side and server-side code. Table 4.1 presents these categories, whether or not the category includes client-side and server-side application code, and the aggregate number of JavaScript files and lines of JavaScript code that we *analyze* for each applications category.

Table 4.1: JavaScript subject systems in our study

Category	**Subject systems**	**Client side**	**Server side**	**Total files**	**Total LOC**
NPM Modules	86		✓	4,983	1,228,271
Web Apps.	16	✓	✓	1,779	494,621
Game Engines	16	✓	✓	1,740	1,726,122
Frameworks	8	✓		2,374	711,172
DataViz Libs.	6	✓		3,731	958,983
Games	6	✓		347	119,279
Total	**138**	✓	✓	**14,954**	**5,238,448**

The 86 NPM modules we study are the most depended-on modules in the NPM repository [2]. The other subject systems were selected from *GitHub Showcases* [3], where popular and trending open source repositories are organized around different topics. The subject systems we consider are JavaScript-only systems. Those systems that contain server-side components are written in Node.js[1], a popular server-side JavaScript framework. Overall, we study callbacks in over 5 million lines of JavaScript code.

4.3 Analysis

To address the three research questions, we have developed a static analyzer to search for different patterns of callbacks in JavaScript code[2].

Our static analyzer builds on prior JavaScript analysis tools, such as Esprima [31] to parse and build an AST, Estraverse [58] to traverse the AST, and TernJS [30], a type inference technique by Hackett and Guo [29], to query for function type arguments. We also developed a custom set of analyses to identify callbacks and to measure their various properties of interest.

Duplicate code is an issue for any source code analysis tool that measures the prevalence of some language feature. Our analysis maintains a set of dependencies

[1] https://nodejs.org

[2] Our static analysis approach considers most program paths, but it does not handle cases like `eval` that require dynamic analysis.

for a subject system and guarantees that each dependency is analyzed exactly once. To resolve dependencies expressed with the `require` keyword, we use TernJS and its *Node* plugin.

In the rest of this section we detail our analysis for each of the three research questions.

Prevalence of callbacks (RQ1). To investigate the prevalence of callbacks in JavaScript code we consider all function definitions and function callsites in each subject system. For each subject, we compute (1) the percentage of function definitions that accept callbacks as arguments, and (2) the percentage of function callsites that accept callbacks as arguments.

We say that f is a *callback-accepting function* if we find that at least one argument to f is used as a callback.

To determine whether a parameter p of a function f definition is a callback argument, we use a three-step process: (1) if p is invoked as a function in the body of f then p is a callback; (2) if p is passed to a known callback-accepting function (e.g., `setTimeout`) then p is a callback; (3) if p is used as an argument to an unknown function f', then recursively check if p is a callback parameter in f'.

Note that for f to be a callback-accepting function, it is insufficient to find an invocation of f with a function argument p. To be a callback-accepting function, the argument p must be invoked inside f, or p must be passed down to some other function where it is invoked.

We do not analyze a subject's dependencies, such as libraries, to find callbacks[3]. The only time we analyze external libraries is when a subject system calls into a library and passes a function as an argument. In this case our analysis check whether the passed function is invoked as a callback in the library code.

We also study whether callback usage patterns are different between server-side and client-side JavaScript code. Categorizing the projects known as purely client-side (MVC frameworks like Ember, Angular) or purely server-side (NPM modules) is easy. But, some projects contain both server-side and client-side JavaScript code. To distinguish client-side code from server-side code, we use the project directory structure. We assume that client-side code is stored in a directory named *www*,

[3]For instance, JavaScript files under the *node_modules* directory are excluded.

public, *static*, or *client*. We also identify client-side code through developer code annotations (e.g., `/* jshint browser:true, jquery:true */`).

```
1  function getRecord(id, callback) {
2    http.get('http://foo/' + id, function (err, doc) {
3      if (err) {
4        return callback(err);
5      }
6        return callback(null, doc);
7    });
8  }

10 var logStuff = function () { ... }
11 getRecord('007', logStuff);
```

Listing 4.1: An example of an asynchronous anonymous callback.

For example, consider the code in Listing 4.1. To check if the `getRecord()` function invocation in line 11 takes a callback, we analyze its definition in line 1. We find that there is a path from the start of `getRecord()` to the `callback` invocation (which happens to be the `logStuff` argument provided to the `getRecord()` invocation). The path is: $getRecord() \rightarrow http.get() \rightarrow Anonymous1() \rightarrow callback()$. Therefore, `logStuff()` is a callback function because it is passed as a function argument and it is invoked in that function. This makes `getRecord()` a callback-accepting function.

We label a *callsite* as callback-accepting if it corresponds to (1) a function that we determine to be callback-accepting, as described above, or (2) a function known a-priori to be callback-accepting (e.g., `setTimeout()`). The `getRecord` callsite in line 11 is callback-accepting because the function `getRecord` was determined to be callback-accepting.

Callback usage in practice (RQ2). Developers use callbacks in different ways. Some of these are known to be problematic [49] for comprehension and maintenance (e.g., see Chapter 2). We characterize callback usage in three ways: we compute (1) the percentage of callbacks that are anonymous versus named, (2) the percentage of callbacks that are asynchronous, and (3) the callback nesting depth.

Anonymous versus named callbacks. If a function callsite is identified as callback-accepting, and an anonymous function expression is used as an argument, we call it an instance of an anonymous callback.

Asynchronous callbacks. A callback passed into a function can be deferred and invoked at a later time. This deferral happens through known system APIs, which deal with the task queue in common browsers and Node.js environments. Our analysis detects a variety of APIs, including DOM events, network calls, timers, and I/O. Table 4.2 lists examples of these asynchronous APIs.

Table 4.2: Example Asynchronous APIs available to JavaScript programs

Category	**Examples**	**Availability**
DOM events	`addEventListener`, `onclick`	Browser
Network calls	`XMLHTTPRequest.open`	Browser
Timers (macro-Task)	`setImmediate()`, `setTimeout()`, `setInterval()`	Browser, Node.js
Timers (micro-task)	`process.nextTick()`	Node.js
I/O	APIs of `fs`, `net`	Node.js

For each function definition, if a callback argument is passed into a known deferring function call, we label this callback as asynchronous.

Callback nesting. The number of callbacks nested inside one after the other is defined as the callback depth. According to this definition, Listing 1.1 has a callback depth of three because of callbacks at lines 2, 3, and 5.

In cases where there are multiple instances of nested callbacks in a function, we count the maximum depth of nesting, including nesting in conditional branches. This under-approximates callback nesting. For example, Listing 4.2 shows an example code snippet from an open source application[4]. There are two sets of nested callbacks in this example, namely, at Lines 1, 4, 5 (depth of three) and a second set at Lines 1, 12, 13, 15 (depth of four). Our analysis computes the maximum nesting depth for this function, which is four.

Handling callbacks (RQ3). There are a number of solutions to help with the complexities associated with callbacks. We consider three well-known solutions:

[4]https://github.com/NodeBB/NodeBB

```
define('admin/general/dashboard', 'semver', function(semver) {
  var Admin = {};

  $('#logout-link').on('click', function() {
    $.post(RELATIVE_PATH + '/logout', function() {
      window.location.href = RELATIVE_PATH + '/';
    });
  });

  ...

  $('.restart').on('click', function() {
    bootbox.confirm('Are you sure you wish to restart NodeBB?', function(
        confirm) {
      if (confirm) {
        $(window).one('action:reconnected', function() {
          app.alert({ alert_id: 'instance_restart', });
        });

        socket.emit('admin.restart');
      }
    });
  });
  return Admin;
});
```

Listing 4.2: Example of multiple nested callbacks in one function.

(1) the prevalence of the error-first callback convention, (2) the prevalence and usage of Async.js [40], a popular control flow library, and (3) the prevalence of Promises [12], a recent language extension, which provides an alternative to using callbacks. For each solution, we characterize its usage — are developers using the solution and to what extent.

Error-first callbacks. To detect error-first callbacks (see Chapter 2) we use a heuristic. We check if the first parameter p of a function f definition has the name '*error*' or '*err*'. Then, we check if f's callsites also contain '*error*' or '*err*' as their first argument. Thus, our analysis counts the percentage of function definitions that accept an error as the first argument, as well as the percentage of function callsites that are invoked with an error passed as the first argument.

Async.js. If a subject system has a dependency on Async.js (e.g., in their `package.json`), we count all invocations of the Async.js APIs in the subject's code.

Promises. We count instances of Promise creation and consumption for each subject system. If a new expression returns a Promise object (e.g., `new Promise()`), it is counted as a Promise creation. Invocations of the `then()` method on a Promise object are counted as Promise consumption — this method is used to attach callbacks to a promise. These callbacks are then invoked when a promise changes its state (i.e., evaluates to success or an error).

4.4 Results

4.4.1 Prevalence of Callbacks (RQ1)

In the subject systems that we studied, on average, 10% of all function definitions and 19% of all function callsites were callback-accepting. Figure 4.1 depicts the percentage of callback-accepting function definitions and callsites, per category of systems (Table 4.1), and in aggregate. The figure also shows how these are distributed across client-side and server-side, indicating that server-side code generally contains more functions that take callbacks than client-side code.

Finding 1*: On average, every 10th function definition takes a callback argument. Callback-accepting function* definitions *are more prevalent in server-side code (10%) than in client-side code (4.5%).*

Finding 2*: On average, every 5th function callsite takes a callback argument. Callback-accepting function* callsites *are more prevalent in server-side code (24%) than in client-side code (9%).*

Implications. Callbacks are utilized across all subject categories we considered. Some categories contain a higher degree of callback usage. For example, the web applications category contained a higher fraction of both callback-accepting function definitions and invocations of such functions. The inter-category differences in callback usage were not large, however, and we believe that these differences can be ascribed to the fact that some categories contain more server-side code than others. We believe the more extensive usage of callbacks in server-side code can be attributed to the continuation-passing-style of programming that was advocated in the Node.js community from its inception.

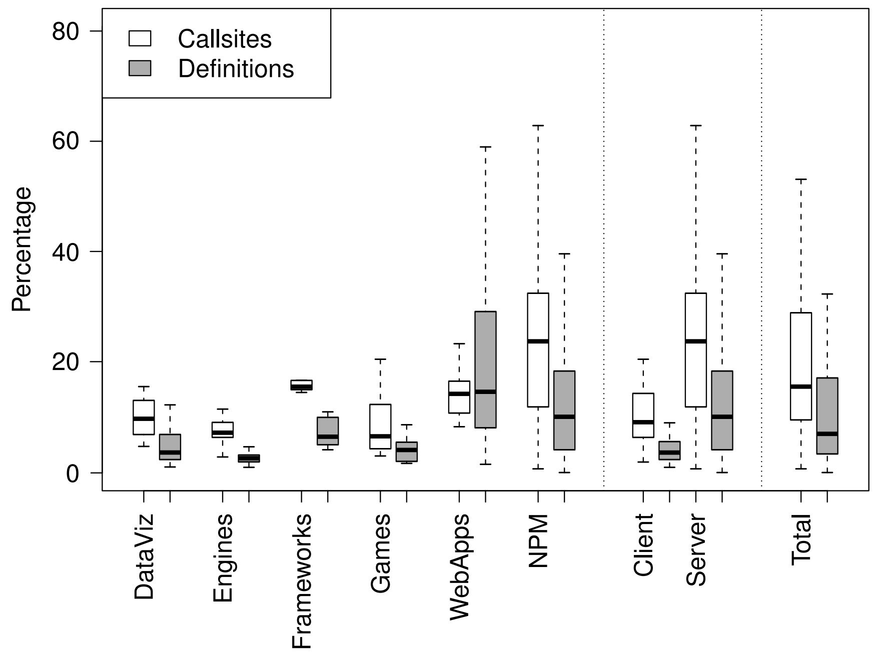


Figure 4.1: Boxplots for percentage of callback-accepting function definitions and callsites per category, across client/server, and in total.

4.4.2 Callback Usage (RQ2)

Asynchronous Callbacks

As a reminder, an asynchronous callback is a callback that is eventually passed to an asynchronous API call like `setTimeout()` (see Section 4.3 for more details).

Figure 4.2 shows the prevalence of asynchronous callback accepting function callsites. Across all applications there was a median of 56% and a mean of 56% of callsites with asynchronous callbacks. The figure also partitions the data into the client-side and server-side categories. We find that usage of asynchronous callbacks is higher in client-side code. Of all callsites in client-side code, 72% were asynchronous. On the server-side, asynchronous callbacks usage is 55% on average.

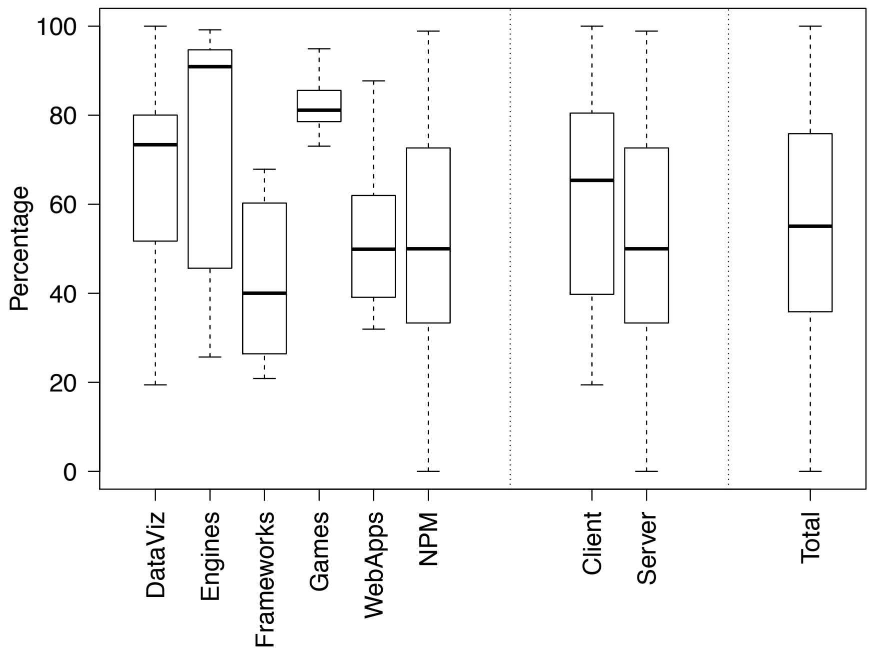


Figure 4.2: Boxplots for percentage of asynchronous callback-accepting function callsites.

***Finding 3**: More than half of all callbacks are asynchronous. Asynchronous callbacks, on average, appear more frequently in client-side code (72%) than server-side code (55%).*

Implications. The extensive use of asynchrony in the subjects we studied indicates that program analyses techniques that ignore the presence of asynchrony are inapplicable and may lead to poor results. Analyses of JavaScript must account for asynchrony.

The extensive use of asynchronous scheduling paired with callbacks surprised us. The asynchronous programming style significantly complicates program control flow and impedes program comprehension [13]. Yet there are few tools to help with this. We think that the problem of helping developers reason about large JavaScript code bases containing asynchronous callbacks, both on the client-side and the server-side, deserves more attention from the research community.

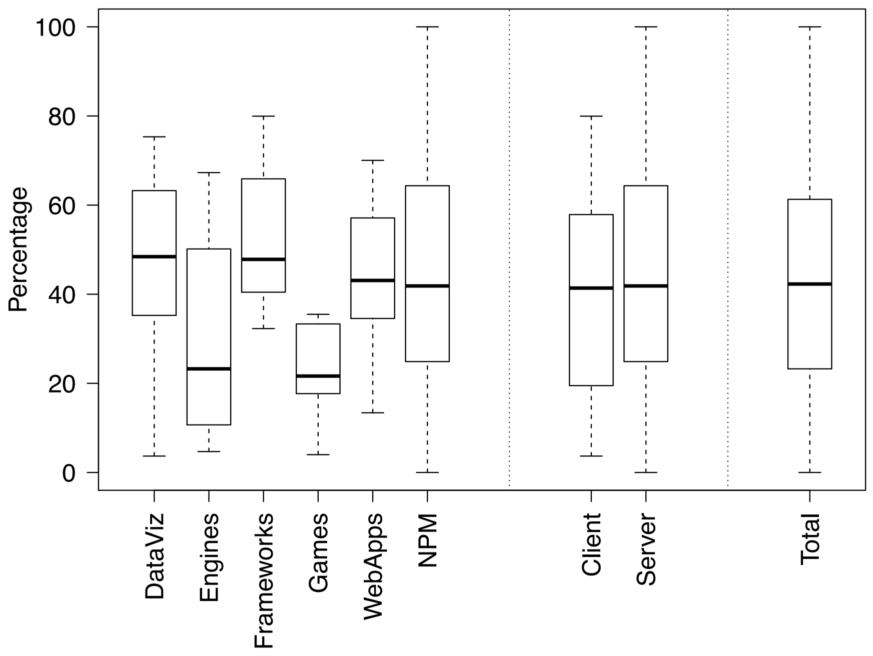


Figure 4.3: Boxplots for percentage of anonymous callback-accepting function callsites per category, across client/server, and in total.

Anonymous Callbacks

Figure 4.3 shows the prevalence of anonymous callback- accepting function callsites. The median percentage across the categories ranges from 23% to 48%, which is fairly high considering that anonymous callbacks are difficult to understand and maintain. Figure 4.3 also shows the same data partitioned between client-side and server-side code, and indicates that server-side code contains a slightly higher percentage of anonymous callbacks than client-side code.

***Finding 4**: Over 43% of all callback-accepting function callsites are invoked with at least one anonymous callback. There is little difference between client-side and server-side code in the extent to which they use anonymous callbacks.*

Implications. This finding indicates that in a large fraction of cases, a callback is used once (anonymously) and is never re-used again. It seems that developers find anonymous callbacks useful in spite of the associated comprehension, debugging, and testing challenges. We think that this topic deserves further study — it is important to understand why developers use anonymous callbacks and prefer them over named callbacks. Possible reasons for using anonymous callbacks could be code brevity, or creating temporary local scopes (e.g., in closures). We also think that the high fraction of anonymous callbacks indicates that this popular language feature is here to stay. Therefore, it is worthwhile for the research community to invest time in developing tools that will support developers in handling anonymous callbacks.

Nested Callbacks

Figure 4.4 presents our results for the total number of instances of nested callbacks at each observed nesting level. We found that the majority of callbacks nest two levels deep. Figure 4.4 shows this unusual peak at nesting level of 2. We also found that callbacks are nested up to a depth of 8 (there were 29 instances of nesting at this level). In these extreme cases developers compose sequences of asynchronous callbacks with result values that flow from one callback into the next. These extreme nesting examples are available as part of our dataset [9].

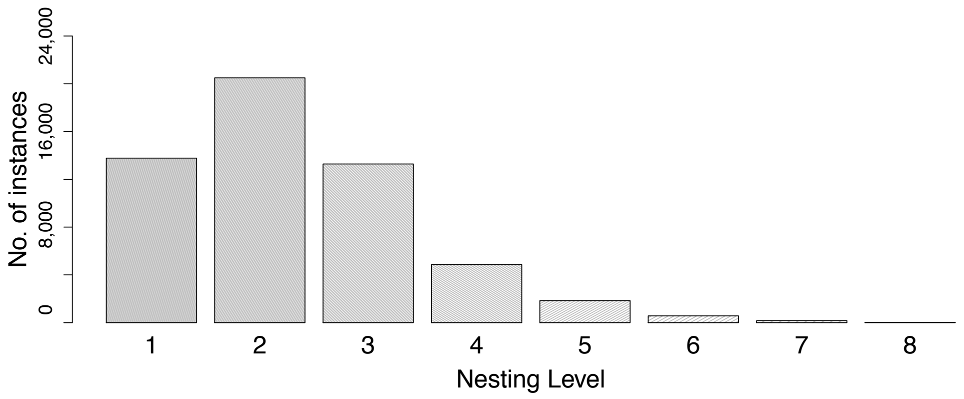


Figure 4.4: Instances of nested callbacks for a particular nesting level.

Finding 5: *Callbacks are nested up to a depth of 8. There is a peak at nesting level of 2.*

Implications. As with anonymous and asynchronous callbacks, callback nesting

```
1 $(document).ready(function () {
2   $('.star').click(function (e) {
3     ...
4   })
5 })
```

Listing 4.3: Example of a nested callback introduced by the wrapping `$(document).ready()` function from the jQuery library.

taxes code readability and comprehension. We find that nesting is widely used in practice and note that developers lack tools to manage callback nesting. We believe that there is ample opportunity in this area for tool builders and software analysis experts. The number of instances decreases from level 1 to 8, except at level 2. Based on our investigation of numerous level 2 callback nesting examples, we believe that the peak at level 2 is due to a common JavaScript practice in which project code is surrounded with an anonymous function from an external library. This is used, for example, for module exporting, loading, or to wait for the DOM to be fully loaded on the client-side. Due to the extra callback surrounding the project code, in these type of projects, callbacks begin nesting at level 2. Listing 4.3 lists an example of this kind of nesting with the `$(document).ready()` function (line 1) from the popular jQuery library. This function waits for the DOM to load. It increases the callback nesting in the rest of the code by 1 (e.g., the callback on line 2 has a nesting level of 2).

4.4.3 Solutions (RQ3)

Error-first Protocol

We found that 20% of all function definitions follow the error-first protocol. The median percentage across the categories ranges from 4% to 50%. The fraction of function definitions that adhere to the error-first protocol is almost twice as high in the server-side code (30%) than in the client-side code (16%). In addition, the error-first protocol was the most common solution among the three solutions we considered. For example, 73% (63 out of 86) NPM modules and 93% (15 out of 16) web applications had instances of the error-first protocol.

***Finding 6**: Overall, every 5th function definition adheres to the error-first protocol. The error-first protocol is used twice as often in server-side code than in client-side code*

Implications. Although we found that a non-trivial fraction of JavaScript functions rely on the error-first protocol, it remains an ad-hoc solution that is loosely applied. The relatively low and highly variable use of the error-first protocol means that developers must check adherence manually and cannot depend on APIs and libraries to enforce it. Such idiom-based strategies for handling exceptions are known to be error-prone in other languages, such as C [18]. It would be interesting to study if this is also the case for JavaScript functions that follow the error-first callback idiom.

Async.js

To study Async.js, we considered subject systems that use this library. We found that only systems in the web applications and NPM modules categories used this library. Our results show that 9 of 16 (56%) web applications and just 9 of 85 (11%) NPM modules use the Async.js library to manage asynchronous control flow. Table 4.3 shows the top 10 used functions from the Async.js API (by number of callsites) for these two categories of subject systems.

This table indicates that the sets of functions used in the top 10 list are similar. But, there are notable differences: for example, `nextTick` was the most used Async.js method in web applications and just the 9th most used method in NPM modules. The `nextTick` method in Async.js is used to delay the invocation of the callback until a later tick of the event loop, which allows other events to precede the execution of the callback. In Node.js code the `nextTick` is implemented using the `process.nextTick()` method in the runtime. In browsers this call is implemented using `setImmediate(callback)` or `setTimeout(callback, 0)`. In this case Async.js provides a single interface for developers to achieve the same functionality in both client-side and server-side code.

In NPM modules `parallel` is the most widely used Async.js method (it is the 6th most popular among web applications). This call is used to run an array of independent functions in parallel.

Table 4.3: Top 10 Async.js invoked methods in JavaScript web applications (left) and NPM modules (right). The * symbol denotes calls that do not appear in both tables.

Rank	Method	Count	Rank	Method	Count
1	nextTick	18	1	parallel	189
2	queue*	16	2	apply	81
3	each	14	3	waterfall	72
3	setImmediate*	14	4	series	61
3	series	14	5	each	48
6	auto*	11	6	map	37
6	waterfall	11	7	eachSeries*	20
6	parallel	11	8	eachLimit*	12
9	map	10	9	whilst*	10
9	apply	10	9	nextTick	10

As a final example, the second-most used call in web-applications, `queue`, does not appear in the top ten calls used by NPM modules. The `queue` method functionality is similar to that of `parallel`, except that tasks can be added to the queue at a later time and the progress of tasks in the queue can be monitored.

We should note that because JavaScript is single-threaded, both the `parallel` and `queue` Async.js calls do not expose true *parallelism*. Here, parallel execution means that there may be points in time when two or more tasks have started, but have not yet completed.

There are significant differences between web applications and NPM modules in terms of the Async.js API usage. To characterize this difference, we first ranked the API functions according to the number of times they were used in web applications and NPM modules. Then we analyzed the difference between the ranks of each function in the two categories. For example, the rank of `nextTick` is 9 and 1 in the NPM modules and web applications, respectively, making the absolute difference 8. Overall, the rank differences had a mean of 6.2, a median of 5.5, and a variance of 23.8. This indicates that the Async.js library is used differently in these two categories of subject systems.

***Finding 7**: More than half of the web applications (56%) use the Async.js library to manage asynchronous control flow. The usage is much lower (11%) in the NPM modules. In addition, the Async.js library is used differently (rank variance of 23.8) in these two categories of subject systems.*

Implications. Libraries, such as Async.js, provide one means of coping with the complexity of callbacks. However, because library solutions are not provided natively by the language runtime, the developer community can be divided on which library to use, especially as there are many alternatives (see Chapter 2). We think that the difference in Async.js API usage by developers of web applications (that include both client-side code and server-side code) and NPM modules (exclusively server-side code) indicates different underlying concerns around callbacks and their management. We think that this usage deserves further study and can inform the design of future libraries and proposals for language extensions [36, 42, 59].

Promises

Table 4.3 shows the percentage of subjects that create promises and the percentage of subjects that use promises.

Table 4.4: Percentage of subject systems creating and using Promises

Category	Subjects creating Promises (%)	Subjects using Promises (%)
DataViz libraries	6	31
Game Engines	0	25
Frameworks	50	75
Games	0	17
Web Applications	13	50
NPM Modules	3	12
Total	8	26

Figure 4.5 shows box plots for the number of Promise creating instances using `new Promise()` constructs and Promise usage, e.g., `then()`, in the different sub-

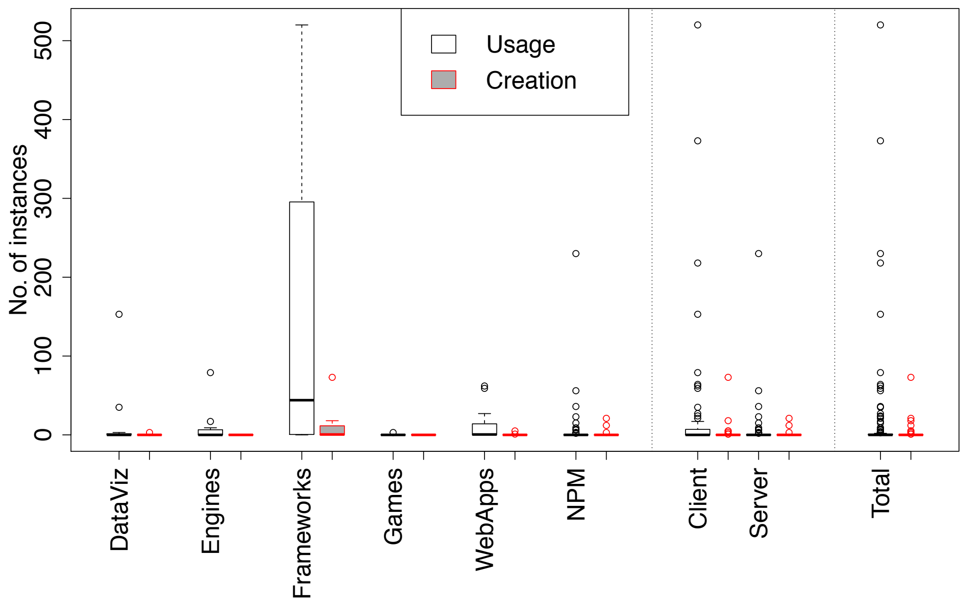


Figure 4.5: The distribution of total Promise usage and creation instances by category, across client/server, and in total.

ject categories, partitioned across client/server, and in total. It should be noted that not all application access a Promise through the `new Promise()` statement; some invoke library functions that return Promises.

In aggregate, we found that 37 of 138 (27%) applications use Promises. They were predominantly used by client-side frameworks (75%), with a maximum of 513 usage instances (across all frameworks) and a standard deviation of 343 usage instances. In all the other subject systems, usage of Promises was rare, with a mean close to zero. There was one outlier, the bluebird NPM module[5], that had 2,032 Promise usage instances. This module implements the Promises specification as a library. We therefore omit it from our results.

***Finding 8**: 27% of subject systems use Promises. This usage is concentrated in client-side code, particularly in JavaScript frameworks.*

Implications. Although Promises is a promising language-based approach to resolving many of the challenges related to callbacks, such as nesting and error han-

[5] https://www.npmjs.com/package/bluebird

dling, we have not observed a significant uptake of Promises among the systems we studied. This could be because Promises is a relatively new addition to browsers and Node.js. It would be interesting to study how this adoption evolves and whether Promises lead to higher quality and more maintainable JavaScript code. Tools that automatically refactor callbacks into Promises would help developers to migrate existing large projects to use Promises.

4.5 Threats to Validity

There are both internal and external threats to validity for our study. We overview these threats in this section.

Internal threats. Our JavaScript analyses rely on a number of development conventions. For example, our error-first callback analysis depends on a naming heuristic — the assumption that code adhering to the error-first protocol will name the first argument of a function as *err* or *error*. A threat is that we may be under-counting error-first protocol adherence by missing cases where the protocol is followed but a different argument name is used. And, we may also be over-counting adherence, since an argument name does not necessarily mean that the code uses error-first protocol, or that it properly follows the protocol. We also rely on directory naming conventions and use code annotations as hints to identify client-side and server-side code.

We decided to count features of callback usage in particular ways. For example, we count callback nesting by taking the maximum depth of callback nesting for a function. This can provide an under approximation of the number of instances of nested callbacks.

Our analyses are static. This limits the kinds of JavaScript behaviors that we can analyze. For example, we do not handle code in `eval` statements in our study.

External threats. Although we study over 5 million lines of JavaScript code, our sample might not be representative, in a number of ways. First, it comes from open source projects of a particular size and maturity. Second, we consider projects that use JavaScript and are primarily mono-lingual. For example, we do not consider projects that use JavaScript on the client-side and Java on the server-side. As a result, our findings may not generalize to other types of JavaScript projects.

However, the subject systems in our study represent five different categories and as the first study of its kind, we believe our characterization study of JavaScript callback usage in practice is worthwhile, and hope that it will lead to other studies that consider a broader variety of subject systems.

All our empirical data and toolset are publicly available; since the subject systems are all open source, our study should be repeatable.

4.6 Conclusions

All modern JavaScript applications that handle and respond to events use callbacks. However, developers are frequently frustrated by “callback hell” — the comprehension and maintainability challenges associated with nested, anonymous callbacks and asynchronous callback scheduling. In an empirical study of callbacks usage in practice, we study over 5 million lines of JavaScript code in 138 subject systems that span a variety of categories. We report on the prevalence of callbacks, their usage, and the prevalence of solutions that help to manage the complexity associated with callbacks. We hope that our study will inform the design of future JavaScript analysis and code comprehension tools. Our analysis [4] and empirical results [9] are available online.

Chapter 5

Refactoring

5.1 Background

In this section, we elaborate on the key background of the JavaScript language, its runtime, and promises, which are necessary to understand the rest of the thesis.

Callback Nesting and Error Handling in Practice. JavaScript programs frequently contain sequences of asynchronous tasks that need to be completed sequentially (e.g., when a click event is fired, send data to the server, and then on response from the server, update an element in the DOM). Using callbacks to handle the control flow in these situations results in *nested callbacks*, which increase code complexity. Because each callback adds a new function definition and indentation level, this affects the understandability of the program.

In our empirical study (Section 4.4), callback depth was defined to be the number of callbacks nested inside one after the other. Using static analysis we computed the maximum nesting depth for each function and found that the majority of callbacks are nested to two levels, nesting can happen as deep as 8 levels, and that the number of nested instances decreases from level 1 to 8, except at level 2.

We also found that 20% of all callback accepting function definitions use the 'error-first callbacks' convention to propagate errors asynchronously.

To understand whether nested callbacks use the error-first protocol, we focused on a subset of 16 subject systems (having a total of 494 KLOC) from the previous study and counted the number of occurrences of the error-first protocol in nested callbacks across these systems. The results for these 16 web applications are shown

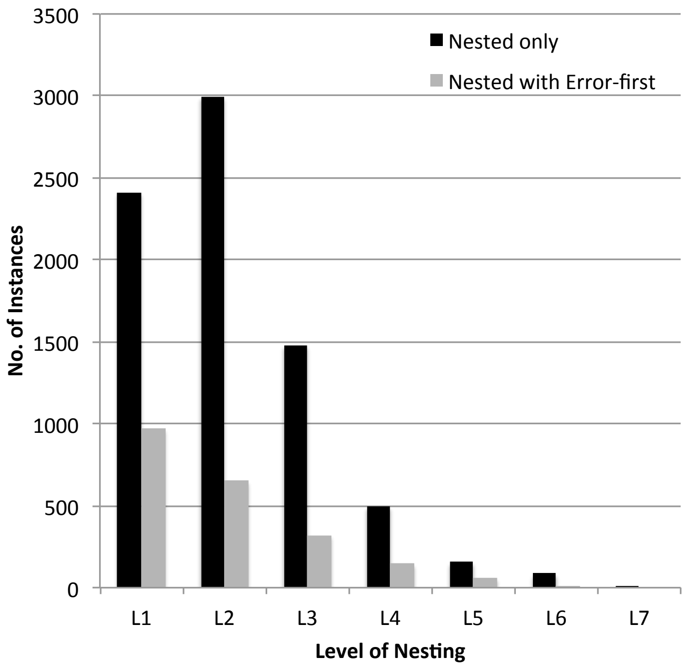


Figure 5.1: Count of nested callbacks with at least 1 error-first callback in web applications.

in the Figure 5.1. It shows the number of nested callbacks for each nesting level as black bars and number of error-first protocol inside nested callbacks as light grey bars. We found out a large number of nested callbacks included at least one instance of the error-first protocol. On average 28% of nested callbacks use the error-first protocol.

Uncaught errors. Because each callback is executed with a new call stack, uncaught errors cannot be propagated up the call stack to the original caller of the asynchronous function. Consider Listing 5.1: if an error is raised in `handleError` at lines 3, 7 or 10, the programmer cannot handle these errors with one `catch` block because each of these statements are executed within a fresh call stack. The programmer needs three `catch` blocks to handle uncaught exceptions from the three callback functions. To avoid missing error handlers, developers occasionally use a

```
getUser('jackson', function(error, user) {
  try{
    if (error) {
      handleError(error);
    } else {
      getNewTweets(user, function(error, tweets) {
        try{
          if (error) {
            handleError(error);
          } else {
            updateTimeline(tweets, function(error) {
              try{
                if (error) handleError(error);
              } catch(e) {
                globalErrorHandler(e);
              }
            });
          }
        } catch(e) {
          globalErrorHandler(e);
        }
      });
    }
  } catch(e) {
    globalErrorHandler(e);
  }
});
```

Listing 5.1: A sequence of asynchronous operations.

global catch-all `uncaughtException` handler as a hack, as shown below.

```
process.on('uncaughtException', function (err) {
  console.error(' uncaughtException:', err.message)
  console.error(err.stack)
  process.exit(1)
})
```

Synchronization. Callbacks do not have built in synchronization and are not naturally idempotent. This causes problems when, for example, the same callback is accidentally invoked multiple times inside another continuation. Consider Listing 5.2: because there is no return statement within the `if` block, `cb` is executed twice if `foo` is truthy. Based on our experience, this is a common mistake among JavaScript developers.

Promises. A *promise* is a design pattern that handles asynchronous events and solves many of the callback-related problems described previously. While promises have been used for some time in JavaScript with third party libraries, the next

```
handler(cb, foo) {
  if(foo) cb(foo);
  cb(foo);
}
```

Listing 5.2: A JavaScript snippet illustrating that callbacks are vulnerable to synchronization bugs.

ECMA specification (version 6) [6] of the language has promises built in.

With the promises design pattern, instead of accepting a callback as the continuation function, an asynchronous function returns a `Promise` instance. This instance represents a value that will be available sometime in the future, for example, after a deferred task has completed. A promise can be in one of three states:

Pending. The deferred task has not yet completed, so the outcome of the Promise has not been determined.

Fulfilled. The deferred task has completed successfully and its result is available.

Rejected. The deferred task has failed and a corresponding error is available.

A promise's initial state is `pending`, and will either transition to `fulfilled` or `rejected` depending on the outcome of the deferred task. Once the caller of the asynchronous function has retrieved a promise, it can register two continuation functions with the promise: a *success handler* and an optional *error handler*. The success handler is called when the promise enters the fulfilled state. The error handler is called when the promise enters the rejected state.

```
getUser('jackson')
  .then(getNewTweets, handleError)
  .then(updateTimeline, handleError)
  .catch(globalErrorHandler);
```

Listing 5.3: A sequence of asynchronous operations composed with Promises.

Promises solve many of the problems associated with callbacks. Consider Listing 5.3, which is semantically equivalent to Listing 5.1, but uses promises. Callback nesting is eliminated by chaining promises. Error handling is separated into a success handler (the first parameter of `.then`) and an error handler (the second parameter of `.then`). Uncaught errors are handled with `.catch`. Basic synchronization is handled automatically because promises guarantee that the error and success handlers only execute once.

5.2 Exploratory Study: Refactoring Callbacks to Promises

We carried out an exploratory study to better understand the extent and manner in which developers refactor callbacks into promises. Our exploratory study consists of three parts: (1) a manual inspection of issues on GitHub related to promises, (2) a manual inspection of pull requests on GitHub related to promises, and (3) an automated mining of commits that refactored asynchronous callbacks to promises.

5.2.1 Finding Issues Related to Callbacks and Promises

The first part of our study explored posts on GitHub's issue tracking system. GitHub is one of the most popular collaborative software-development platforms among JavaScript developers [24] and provides the largest publicly available dataset including developer discussions and development history. Because of its popularity and depth of content, we used GitHub's search feature to find issues related to refactoring of asynchronous callbacks to promises.

We first used the query string: "`promise callback language:JavaScript stars:>30 comments:>5 type:issue`" to search for GitHub issue discussions that were non-trivial (containing at least 5 comments), associated with projects that were popular (starred by at least 30 users) and contained the terms `promise` and `callback`. This search resulted in 4,342 issues. We randomly sampled 11 issues from this set and manually inspected the discussions associated with the issues. We found that in majority of issues (8 out of 11), the final consensus was to refactor the code to use promises instead of using asynchronous callbacks. Many discussion participants agreed that using promises would be beneficial to the project, however, the main reasons for reluctance to migrate to promises were that (1) promises may cause significant changes to existing APIs and (2) the development costs associated with the change was prohibitive.

Many users who showed interest in moving to promises, emphasized various benefits of Promises:

"I've recently solved various problems using promises, e.g., polyglot data migrations, browser/node client libraries for an HTTP API, kv store interfaces & controller code. I've both written completely new code and rewrapped callback-style code. Personally I'm very pleased with the amount of additional safety and expressiveness I've gained by using promises. I'm not dismissing callbacks per se, but personally I find it much simpler to reason about code using promises than code using callbacks & utility libraries like async." [1]

"In general, I think promises provide a good foundation for async: they're fairly simple and lightweight, but they make it easy to build higher level async constructs. For example, when'js's when.map and when.reduce are surprisingly compact, but quite powerful. I see that as an advantage of promises, they are both useful in and of themselves, and also make good building blocks. I also feel that promises make for very clean API design. There is no need for callbacks [...] to be added to every function signature in your API. You can just return a promise instead." [2]

We then narrowed down the search by including the term `refactor`.[3] This resulted in 351 issues, many of which indicated strong demand to refactor code to use promises. For example, one participant stated: *"So this is something that is purely for devs but I think it is about time to do this. i.e. git-task is a great candidate to take full advantage of promises and it would have made implementation of #602 much easier."* [4]

Many of these requests came from users of JavaScript libraries who wanted promises as part of the library API: *"Are there any plans for Promise support, alongside the callbacks and streams? Proper Promise support in any-db and any-*

[1] https://github.com/share/ShareJS/issues/268

[2] https://github.com/gladiusjs/gladius-core/issues/127

[3]Complete search string: "`Refactor promises language:JavaScript stars:>20 type:issue`" We lowered the number of stars to capture more projects.

[4] https://github.com/FredrikNoren/ungit/issues/603

db-transaction would be really nice :)"[5], and *"Add promise API option?"* [6]. Some of the users encouraged the move to promises by sharing their own experiences of using promises: *"We've recently converted pretty large internal codebases from async.js to promises and the code became smaller, more declarative, and cleaner at the same time."* [7]

5.2.2 Exploring Refactoring Pull Requests

The second part of our study explored pull requests on GitHub in order to determine whether developers acted on suggestions for refactoring asynchronous callbacks to promises. We did this by searching GitHub for pull requests associated with refactoring asynchronous callbacks to promises and manually inspecting the results. Our search used the following query string: "`Refactor promises language:Javascript stars:>20 type:pr`". The search resulted in 451 pull requests. We observed that most of these pull requests were submitted as improvements to the project and involved replacing callbacks with Promises. These were either native Promises supported by the runtime or ones provided by third-party libraries like *Bluebird, Q*, or *RSVP*. We found that developers act on the desire for promises over asynchronous callbacks and do perform this type of refactoring in practice. A more detailed listing of the discussions we explored in our study, along with listings of relevant quotes, can be found online [11].

5.2.3 Mining Commits

In the third part of our exploratory study, we mined 134 Node.js applications and NPM modules to look for examples of asynchronous callback to promise refactoring in practice. We discovered 39 instances of asynchronous callback to promise rafactorings across 9 projects. This indicates that developers are interested in performing this refactoring in practice. We manually inspected each of these instances and found that five of these instances conform to the standard refactoring pattern that matches what is recommended in developer blog posts [19].

This exploratory study demonstrates that the developers see many advantages

[5] https://github.com/grncdr/node-any-db/issues/66
[6] https://github.com/addyosmani/psi/issues/56
[7] https://github.com/meetfinch/decking/issues/18

in migrating to promises. However, because of the complex control flow associated with callbacks and promises, refactoring code to use promises instead of asynchronous callbacks is difficult. Without tool support, the development costs associated with the change can become the major barrier of promises adoption. We believe there is a strong demand among JavaScript developers for automated tool support to perform these refactorings.

In this chapter, our goal is to develop an approach that can automatically refactor asynchronous callbacks to promises. The approach should have the following properties: (1) candidates for refactoring can be detected automatically, (2) the approach can refactor most asynchronous callbacks, (3) the approach produces code and (4) the understandability of the source code is improved. In the following section we describe our approach.

5.3 Approach

In this section we formally specify our proposed program refactorings. Throughout we use the following notation: we use *async* to denote an asynchronous, built-in, JavaScript API function, such as `process.nextTick` or `fs.readFile`. We use *cb* to denote a callback, or a function that is passed as an argument to other functions. For example, Listing 5.4 gives an abstract example of function f that uses an asynchronous callback cb_f; Line 16 in the listing contains a callsite to f.

```
1  function f(cb_f) {
2    async(function cb_async(error, data) {
3              if(error) cb_f(null, data);
4              else cb_f(error, null);
5  });

7  function cb_f(error, data) {
8    if(error) {
9      // Handle error
10   }
11   else {
12     // Handle data
13   }
14 }

16 f(cb_f);
```

Listing 5.4: Abstract functions and callsites in the original program *P*.

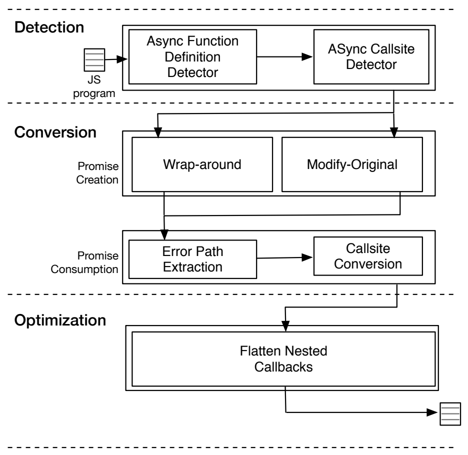


Figure 5.2: Overview of our approach.

We transform P into P' by transforming sub-elements of P. Figure 5.2 illustrates this process. In Sections 5.3.1 and 5.3.2 we describe a process to automatically discover instances of f that can be refactored using our method. In Section 5.3.3 we describe a process to derive a new asynchronous function f' that returns a promise from f. In Section 5.3.4 we describe a process to derive *succ* and *err*, the success and error handlers for the promise from cb_f. In Section 5.3.5 we describe a process to derive the new call site of f' from the original call site of f. Finally, in Section 5.4 we demonstrate that P is semantically equivalent to P'.

5.3.1 Identifying Functions with Asynchronous Callbacks

Our first contribution is in helping developers to automatically identify instances of f to refactor. We consider a function to be an instance of f if *async* is directly invoked inside the body of the function and if one of the following is true:

1. $cb_f = cb_{async}$, or
2. cb_f is invoked inside the closure of cb_{async}

This rule ensures that f is asynchronous and f accepts a callback as a parameter and that callback is deferred to be invoked when the asynchronous API call finishes.

We identify instances of *async* using a whitelist of calls that we know to be asynchronous. This whitelist includes a variety of APIs, including DOM events, network calls, timers, and I/O. The complete list is available online [11].

5.3.2 Choosing a Refactoring Strategy

Our next contribution is to identify which of our automated refactoring techniques (if any) can transform f to f'. We describe two strategies for transforming callbacks into promises: *modify-original* and *wrap-around*. Table 5.1 shows the advantages and disadvantages of each strategy, which are discussed below.

Table 5.1: Refactoring strategies.

Strategy	**Advantages**	**Disadvantages**
modify-original	Produces code similar to how developers would refactor	Transforms some instances
wrap-around	Transforms most instances	Produces code that can be more complex than the original

Modify-original

In our exploratory study (Section 5.2), we observed the relative frequency of different kinds of promise refactorings performed by developers. The modify-original strategy is based on the most frequent type of refactoring that we observed. The limitation of modify-original is that it cannot transform more complex asynchronous callbacks. Candidate instances for the modify-original refactoring must meet the following preconditions. The rationale behind these preconditions is also provided below:

1. **cb_f is invoked inside the closure of cb_{async}**
   This ensures that $cb_f \neq cb_{async}$. This would require a different transformation for cb_f that we do not currently support.
2. **cb_f is not used in cb_{async} other than in instances of (1)**
   This ensures that f does not use cb_f for anything other than as a callback function. Because this parameter is removed during the transformation, cb_f is no longer available to f at runtime.
3. **f always returns `void`**
   This precondition eliminates cases where an alternate synchronization method

is used. We found that in most cases, when an asynchronous function returns a value, the return value is used as an identifier for synchronization. Custom synchronization strategies require detailed knowledge of their implementation to produce a valid refactoring and are therefore not handled by either of our transformation strategies.

4. **cb_f is *splittable***
   This requires that cb_f has a success path and an error path that do not interact with each other (i.e., that cb_f is splittable). For example, if cb_f is using the error-first protocol, the error parameter cannot be used on the success path and the data parameter cannot be used on the error path. This is because promises separate the success and error handlers, so any interaction between the two paths cannot occur in a promises implementation.
5. **f has exactly one *async***
   This is needed because only one promise is returned and the handler for a promise can only be invoked once. If more than one *async* is invoked, a significantly more complex refactoring is needed.
6. **invocations of cb_f provide zero or one argument, or follow the error-first protocol**
   It eliminates cases where more than one non-null argument is given to cb_f. This is a restriction of the current implementation of promises in JavaScript, which only accepts one argument in both the `resolve` and `reject` handlers.
7. **cb_{async} does not use variables named `resolve` or `reject`**
   This prevents problems caused by variable hiding, since our method uses these variable names inside cb_{async}.
8. **f is not contained in a third-party library**
   Finally, this precondition prevents library code from being refactored.

Wrap-around

Because the modify-original strategy cannot transform asynchronous callbacks that do not satisfy one or more of the above preconditions, we also provide a strategy which (unlike modify-original) does not modify the body of f. This strategy is able to refactor a larger number of asynchronous callback functions than modify-original. This strategy, however, comes at the cost of simplicity and comprehension, as this method produces more code than the original and introduces a new function. Candidates for the wrap-around refactoring must satisfy the following preconditions:

1. $cb_f = cb_{async}$ OR cb_f is invoked inside the closure of cb_{async}
2. cb_f is not used in f other than in instances of (1)
3. f always returns `void`
4. cb_f is *splittable*
5. f has exactly one *async*
6. invocations of cb_f provide zero or one arguments, or follow the error first protocol
7. f cannot be refactored by modify-original

The preconditions for the wrap-around strategy are more relaxed than the preconditions for the modify-original strategy. The wrap-around strategy does not include a number of preconditions found in the modify-original strategy. This means that the wrap-around strategy can support the following additional cases: cases where $cb_f = cb_{async}$ (modify-original condition #1), cases where cb_{async} uses variables named `resolve` or `reject` (modify-original condition #7) and cases where f is contained in a third-party library (modify-original condition #8).

The rationale for the preconditions in the wrap-around strategy are as follows: Precondition (1) is the same as our precondition for identifying instances of f in Section 5.3.1. Preconditions (2-6) are the same as the modify-original preconditions. Precondition (7) ensures that the modify-original strategy is selected first, because it produces more understandable code.

5.3.3 Transforming the Asynchronous Function

In this subsection, we specify our transformations for deriving f' from f.

Modify-original

In this strategy, we modify the body of f in the same way that a developer would be likely to perform this transformation. First, our technique creates a new f' that returns a promise:

```
function f'() {
  return new Promise();
});
```

The `Promise` constructor takes one argument: the factory function for the promise. To build this, we declare an anonymous function that wraps the body of f:

```
function f'() {
  return new Promise(function (resolve, reject){
    async(function cb'_async(data) {
      if(error) cb_f(null, data);
      else cb_f(error, null);
    });
  });
});
```

Next, we replace invocations of cb_f with invocations of `resolve` and `reject`. Invocations of cb_f that pass a non-null error argument are converted into invocations of `reject`, which calls *err*. We look for arguments that use the error-first protocol or match the regular expression `e|err|error` to find these invocations. All other invocations of cb_f are converted into invocations of `resolve`, which calls *succ*.

```
function f'() {
  return new Promise(function (resolve, reject){
    async(function cb'_async(data) {
      if(error) reject(null, data);
      else resolve(error, null);
  });
});
```

Finally, in P' we replace f with f'.

The listing 5.5 and listing 5.6 shows how an asynchronous callback instance in a real-world JavaScript program is refactored to use Promises using PROMISESLAND. The refactored version of the function `addTranslations`, does not accept a callback, and instead returns a Promise. The invocations of the callback (lines 5 and 15 in listing 5.5) have been changed to `reject` and `resolve` (lines 6 and 16 in listing 5.6) depending whether an error occurred or not.

Wrap-around

In this strategy, we do not modify f. Instead, we wrap all of the calls to f inside a new function. We create this new function f', which creates and returns a `Promise`.

```
function addTranslations(translations, callback) {
  translations = JSON.parse(translations);
  fs.readdir(dirname + '/../client/src/translations/', function (err, pofiles
      ) {
    if (err) {
      return callback(err);
    }
    var vars = [];
    pofiles.forEach(function (file) {
      var locale = file.slice(0, −3);
      if ((file.slice(−3) === '.po') && (locale !== 'template')) {
        vars.push({tag: locale, language: translations[locale]});
      }
    });

    return callback(vars);
  });
}

addTranslations(trans, jobComplete);
```

Listing 5.5: An example of an asynchronous callback before refactoring to Promises

A call to f is inserted into the body of the factory method for the promise:

```
function f′() {
  return new Promise(function (resolve, reject) {
    f();
  });
}

function f(cb_f) {
  async(function cb_async(error, data) {
    if(error) cb_f(null, data);
    else cb_f(error, null);
  });
}
```

A new anonymous function is created as the continuation function for f. If cb_f follows the error-first protocol, the continuation function provides branches that direct the error parameter to `reject` and the data parameter to `resolve`:

```
function f′() {
  return new Promise(function (resolve, reject) {
    f(function(err,data){
      if(err !== null)
        return reject(err);
      resolve(data);
```

```
function addTranslations(translations) {
  return new Promise(function (resolve, reject) {
    translations = JSON.parse(translations);
    fs.readdir(__dirname + '/../client/src/translations/', function (err,
        pofiles) {
      if (err) {
        reject(err);
      }
      var vars = [];
      pofiles.forEach(function (file) {
        var locale = file.slice(0, -3);
        if ((file.slice(-3) === '.po') && (locale !== 'template')) {
          vars.push({tag: locale, language: translations[locale]});
        }
      });

      resolve(vars);
    });
  });
}

addTranslations(trans).then(jobComplete);
```

Listing 5.6: An example of an asynchronous callback after refactoring to Promises using *Modify-original* strategy

```
    });
  });
}

function f(cb_f) {
  async(function cb_async(error, data) {
    if(error) cb_f(null, data);
    else cb_f(error, null);
  });
}
```

5.3.4 Transforming the Callback Function

From Section 5.3.3, we have a new asynchronous function f' that returns a promise. We now transform all call sites of f to use the promise produced by f'. The first step is to identify call sites of f. We rely on existing static analysis of TernJS [30], a type inference technique based on the work by Hackett and Guo [29] to determine the points-to relationships between call sites of f and the declaration of f.

Next, we convert all call sites to use f'. Consider c, a call site of f. c has a callback function cb_f, which handles both successful and unsuccessful executions of f. However, f' requires a separate handler for successful and unsuccessful executions. From cb_f we derive two functions: the success handler *succ* and the error handler *err*. *succ* is the success-handling path of cb_f, while *err* is the error-handling path of cb_f. We therefore declare a success handler and an error handler for the promise. The code that is executed along the success path in cb_f gets copied into *succ*, while all code that is executed along the error path in cb_f is copied into *err*. Any conditional statements that cause control flow to branch to the success or error paths in cb_f are omitted from the handlers.

```
function succ(data) {
  // Handle data
}
function err(error) {
  // Handle error
}
```

To determine the success path and error path of cb_f, we use a heuristic: we look for an `if` statement that checks if a parameter matching `e|err|error` is non-null. The branch where the parameter is null we consider to be the success path and the branch where the parameter is non-null we consider to be the error path. This is based on the typical usage of the error-first protocol. Finally, in P' we replace cb_f with *succ* and *err*.

5.3.5 Transforming the Call Site

Our last step to have a working program is to transform the call sites of f to invoke f' instead. First, if the wrap-around strategy was used, the name of f is changed to f'. If the modify-original strategy was used, the name remains unchanged.

```
f'(cb_f);
```

Next, since f' does not accept a continuation function as an argument, we remove cb_f as an argument from our call to f'. Because our call to f' produces a promise, we pass *succ* and *err* to the promise by appending the call `.then(succ, err)` to the promise returned by f'.

```
f'().then(succ, err);
```

In some cases, no *err* exists. Either there was no error handling path in cb_f or one was not recognized by our heuristic. In this case, we add a comment in place of *err* as the second argument of `then`, which makes a recommendation to the developer to create an error handler.

5.3.6 Flattening Promise Consumers

After a set of nested callbacks are converted into promises, the result is a set of nested promise consumers. Because `[Promise].then` also returns a `Promise`, we can improve readability by converting nested promises to a flat sequence of chained promises that are semantically equivalent. For example, Listing 5.7 has a set of nested promises that can be refactored to the chain of promises in Listing 5.8

```
getLocationDataNew("jackson").then(function (data) {
  getCoordinatesNew(data.address, data.country).then(function (longLat) {
    getNearbyATMsNew(longLat).then(function (atms) {
      console.log('Closest ATM is at: ' + atms[0]);
    });
  });
});
```

Listing 5.7: Nested promises calls.

```
getLocationDataNew("jackson").then(function (data) {
  return getCoordinatesNew(data.address, data.country);
}).then(function (longLat) {
  return getNearbyATMsNew(longLat);
}).then(function (atms) {
  return console.log('Closest ATM is at: ' + atms[0]);
});
```

Listing 5.8: Chained promises after they are flattened.

We have two preconditions for flattening promise consumers (below, v is a variable):

1. $\forall v$ declared in $succ, v$ is not used inside a closure
2. only one nested promise is consumed inside *succ*

The first condition checks that no variable declared in *succ* is also used in one of the asynchronous handlers declared in *succ*. This condition is necessary because if we add the handler for the nested promise through a promise chain, then *v*, which is declared in *succ* will no longer be available to the nested promise's handler through closure. This is illustrated in Listing 5.9. We cannot flatten these nested promises because the parameter `data` is used by the success handler of `getNearbyATMsNew`.

```
getLocationDataNew("jackson").then(function (data) {
  getCoordinatesNew(data.address, data.country).then(function (longLat) {
    return getNearbyATMsNew(longLat).then(function (atms) {
      console.log("The closest ATM to " + data.address + " is: " + atms[0]);
    });
  });
});
```

Listing 5.9: Nested promises which cannot be flattened

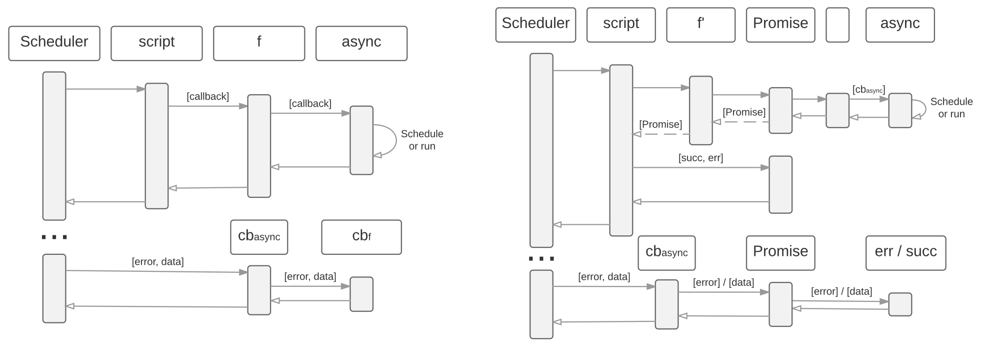
The second condition checks that there is just one asynchronous call inside of *succ* since promise chaining does not support executing multiple asynchronous functions in parallel.

If the two preconditions are met, to flatten a promise chain we perform two transformations:

1. Each handler is modified to return a promise.
2. For each handler that is not at the start of the chain, a new call to `[Promise].then` is created after the previous handler is registered. The handler is passed to the previous promise in the chain.

5.4 Semantic Equivalence of Transformations

Next we use induction to show that our transformations do not change a program's semantics. That is, we show that these transformations are behaviour-preserving program refactorings. For the base case, we need to show that the following five properties are semantically equivalent between P and P': (1) scheduling order, (2) function scope, (3) intra-procedural control flow, (4) inter-procedural control flow, and (5) data flow. Finally, we show that our transformation is semantically equivalent for (6) nested asynchronous callbacks.



(a) (b)

Figure 5.3: The sequence diagram on the left (a) shows the sequence for an asynchronous function that accepts a callback. The sequence diagram on the right (b) shows the sequence for an asynchronous function that returns a promise.

5.4.1 Scheduling Order Equivalence

Consider the sequence diagrams in Figure 5.3. Diagram (a) shows the sequence for an asynchronous callback, while diagram (b) shows the sequence for an asynchronous function that returns a promise. From the diagrams we can see that both versions execute cb_{async} in the same tick. Since we have only one *async* inside f, any scheduling decisions made by *async* will be the same.

5.4.2 Function Scope Equivalence

We consider the function pairs $f \Leftrightarrow f'$, $cb_{async} \Leftrightarrow cb'_{async}$ and $cb_f \Leftrightarrow \{succ, err\}$ separately.

1) $f \Leftrightarrow f'$**.** Since f is unchanged in the wrap-around strategy, the scope is also unchanged. In the modify-original strategy, there are three changes between f and f':

1. The body of f is nested in a new function within f'. This does not affect the scope because anything available in the body of f is available inside the nested function through closure.
2. The parameter cb_f is not present in f'. This affects the scope if cb_f is used anywhere other than as the callback function. This case is filtered by our

preconditions.

3. The parameters `resolve` and `reject` are added to the scope of f'. This causes a naming conflict or overwrites the scope if variables named `resolve` or `reject` exists in the scope of f. This case is also filtered by our preconditions.

2) $cb_{async} \Leftrightarrow cb'_{async}$**.** Because cb_{async} is nested within f, the same changes apply as the changes between f and f'. These changes do not affect the scope of cb'_{async} for the same reasons they do not affect f'. The only other change in cb'_{async} is that invocations of f' are replaced by invocations of `resolve` and `reject`. This change does not affect the scope.

3) $cb_f \Leftrightarrow \{succ, err\}$**.** Because *succ* and *err* are defined at the same level as cb_f, the closure is the same for all three functions. Variables declared within cb_f are copied to *succ* and *err* if they occur on the success path and error path respectively. As long as the success path and error path are correctly retrieved, these copies do not affect the semantics of the program because the variables are only used within the path that is being executed.

5.4.3 Intra-Procedural Control Flow Equivalence

We now show that the intra-procedural control flow of the functions in P is equivalent to the intra-procedural control flow of the functions in P'. We demonstrate that all transformations we perform on the elements in P produce new elements in P' that have control flow equivalent to their counterparts in P.

1) $f \Leftrightarrow f'$**.** For the modify-original strategy, we add a return statement and an invocation of a `Promise` constructor. The `Promise` does not semantically modify the program's behaviour and because the body of f is unchanged, statement order is preserved. For the wrap-around strategy, f is unchanged.

2) $cb_{async} \Leftrightarrow cb'_{async}$**.** For the modify-original strategy, statement order is preserved because we replace calls to cb_f with calls to either *succ* or *err*. These replacements do not change the control flow. For the wrap-around strategy, cb_{async} is unchanged.

3) $cb_f \Leftrightarrow \{succ, err\}$**.** Statement order is preserved because *succ* and *err* are built from the success path and error path of cb_f, which does not change the statement order on either path.

4) $c \Leftrightarrow c'$**.** Consider Figure 5.3. In this case statements are added to the control flow to create the promise, however, these statements do not affect the behaviour of the program and can be ignored. The execution order of statements in the call sites of f is maintained relative to the body of f.

5.4.4 Inter-Procedural Control Flow Equivalence

For intra-procedural control flow, we consider two cases: the control flow between call sites and *async* and the control flow between cb_{async} and cb_f.

1) $c \rightarrow async \iff c' \rightarrow async$**.** Consider Figure 5.3. For the modify-original strategy, the control flow between call sites of f and the body of f now passes through a `Promise` constructor and a factory function. Neither the `Promise` or the factory function semantically modify the program's behaviour and statement order between call sites of f and the body of f is preserved.

For the wrap-around strategy, the control flow between call sites of f and the body of f now passes through f', a `Promise` constructor and a factory function. Neither f', the `Promise` or the factory function semantically modify the program's behaviour and statement order between call sites of f and the body of f is preserved.

2) $cb_{async} \rightarrow cb_f \iff cb'_{async} \rightarrow \{succ, err\}$**.** Consider Figure 5.3. For the modify-original strategy, the control flow between cb'_{async} and $\{succ, err\}$ now passes through a `Promise`. The `Promise` does not semantically modify the program's behaviour and statement order between cb'_{async} and $\{succ, err\}$ is preserved.

For the wrap-around strategy, the control flow between cb_{async} and $\{succ, err\}$ now passes through f' and a `Promise`. Neither f' or the `Promise` semantically modify the program's behaviour and statement order between cb_{async} and $\{succ, err\}$ is preserved.

5.4.5 Data Flow Equivalence

Because the control flow is equivalent, we demonstrate data flow equivalence by showing that the data that is passed between c', f', *async*, cb'_{async}, *succ* and *err* remain equivalent to their counterparts in P.

1) $c \rightarrow async \iff c' \rightarrow async$**.** Data flow is preserved between c' and f' for all arguments besides cb_f since these arguments are unchanged. Because our pre-conditions state that cb_f is only used in f as the continuation function, the data flow is preserved for cb_f when we register *succ* and *err* through `.then(`*succ*`, `*err*`)`.

2) $cb_{async} \rightarrow cb_f \iff cb'_{async} \rightarrow \{succ, err\}$**.** Data flow is preserved between cb'_{async} and $\{succ, err\}$ because of the pre-condition of modify-original in Section 5.3.2. With this condition, we know that at most one argument is passed to *succ* or *err*. The `resolve` and `reject` handlers of promises each take one parameter and propagate them from cb'_{async} to $\{succ, err\}$.

5.4.6 Equivalence of Nested Asynchronous Callbacks

We now take the 'inductive step' and show that our transformation is semantically equivalent for nested asynchronous callbacks.

Because we have a precondition that *async* is the only asynchronous callback within f, we only need to consider asynchronous callbacks that are nested inside cb_f. Because cb_f is the last function in the control flow, we can ignore all other functions and refactorings. Our method will automatically refactor any asynchronous callbacks inside cb_f. After such a refactoring, since our base transformation produces code that is semantically equivalent, the nested asynchronous call will also be semantically equivalent. It follows that all nested asynchronous callbacks will be semantically equivalent.

5.5 Implementation: PROMISESLAND

We have implemented our approach in a tool called PROMISESLAND. It is composed of two components, namely, a static analyzer to search for refactoring opportunities in the form of patterns of asynchrony in JavaScript code and a transformation engine to refactor the detected opportunities into native Promises. PROMISESLAND builds on prior JavaScript analysis tools, such as Esprima [31] to parse and build an AST, Estraverse [58] to traverse the AST, and Escope [8] for scope analysis. We also use TernJS [30], a type inference technique based on the work by Hackett and Guo [29], to query for function type arguments.

Implementation Challenges. Library should not be refactored by default. If there are possible refactorings in module code used by the project code, the user can be notified via `INFO` logs. Heuristics like the following are used to select local dependencies of the project, but to ignore remote dependencies.

- Usually, external dependencies which are downloaded from the public *npm* registry are in the `node_modules` directory and are referred in the source like this:
  `require('module_foo')`
- But internal dependencies are kept separate from the third-party modules, under version control, and referred by a file system path like this:
  `require('./path/to/bar.js')`

Limitations. A limitation of our implementation is that it depends on the soundness of current static analysis techniques. For example, if a points-to relationship between c and f is not discovered by static analysis, our technique will not refactor c. While this is true for points-to relationships between all elements of the transformation, in Section 5.6 we show that this rarely occurs in practice.

5.6 Evaluation

Our goal is to evaluate the efficacy of our approach in terms of its refactoring opportunity detection accuracy, refactoring correctness, and efficiency. We address the following research questions in our empirical evaluation:

RQ1: Can PROMISESLAND accurately identify instances of asynchronous callbacks to be converted?

We consider PROMISESLAND as an automated technique that a developer can use to first identify refactoring opportunities in the code. Therefore, we asses how accurately PROMISESLAND can find asynchronous callbacks in JavaScript code.

RQ2: Can PROMISESLAND correctly refactor asynchronous callbacks to Promises?

The most important factor for adoption of refactoring tools is determined [54] to be confidence in the correctness. We consider a refactoring correct, if it preserves the behaviour after the transformation, which is critical in any refactoring technique.

RQ3: Is PROMISESLAND efficient?

Refactoring tools that are slow face adoption challenges [44] in practice. We evaluate the efficiency of PROMISESLAND in both the detection and transformation of asynchronous callbacks in terms of overhead and analysis speed.

We have made PROMISESLAND open source and all our empirical dataset is available for download [11].

5.6.1 Detection Accuracy (RQ1)

To find out whether PROMISESLAND can accurately identify refactoring candidates in the form of asynchronous callbacks, we first manually inspected four subject systems (see Table 5.2) and counted asynchronous callbacks that can be converted to Promises. This set of subject systems consists of heroku-bouncer,[8] a server-side middleware, moonridge,[9] an isomorphic library for MongoDB, timbits,[10] a client-side widget framework, and tingo-rest,[11] a REST-API wrapper for TingoDB. To limit the manual inspection effort, we included four systems for this evaluation, although we think these four are representative as they include server-side code, client-side code as well as isomorphic JavaScript (executed both on the client-side and server-side).

We then used PROMISESLAND to find refactoring instances, to measure precision and recall. We define precision as the percentage of asynchronous callbacks that can safely be refactored without leading to test failures, across all asynchronous callbacks that PROMISESLAND detects. Recall pertains to the percentage of asynchronous callbacks that PROMISESLAND detects, across all asynchronous callbacks that exist in the subject system.

Table 5.2 presents our results. PROMISESLAND did not report any false positives, providing a precision of 100%. This means that although the static analysis is not sound, in practical use, the tool does not detect any wrong instances, and the developer does not need to be concerned about refactoring incorrectly identified callbacks.

The recall was 83% on average. This is because PROMISESLAND missed a

[8] https://github.com/heroku/node-heroku-bouncer
[9] https://github.com/capaj/Moonridge
[10] https://github.com/postmedia/timbits
[11] https://github.com/lean-stack/node.tingo-rest

few instances of asynchronous callbacks. The reason is that our design is based on the premise that only if it can be guaranteed that all paths of a function execute the callback asynchronously, the callback can be considered to be semantically similar to a Promise (and thus it becomes a refactoring candidate). To ensure statically that the callback is executed asynchronously and exactly once, we follow a conservative approach that can miss some of the potential candidates for conversion. That is the reason that our recall is not 100%. This means in practice although PROMISESLAND detects and transforms most of the candidates, a few can be missed. We believe this can be further improved using more advanced data-flow analysis techniques.

Table 5.2: Detection accuracy of the tool.

Subject System	**LOC** (JS)	**Detected Instances**	**Refactored Instances**	**Precision** (%)	**Recall** (%)
heroku-bouncer	947	7	6	100	85.7
moonridge	3,760	19	14	100	73.6
timbits	1,226	17	15	100	88.2
tingo-rest	238	4	4	100	100
Total	6,171	39	47	100 (avg)	82.9 (avg)

5.6.2 Refactoring Correctness (RQ2)

In prior research, Brodu et al. [17] proposed a compiler-based technique to convert nested callbacks into a simpler specification of Promises called Dues [16]. To evaluate PROMISESLAND, we select the subject systems used by Brodu et al. and compare our results to theirs. This set of subject systems consist of 64 Node.js modules and is expected to be representative of a majority of commonly used JavaScript modules. We measure how many asynchronous callback instances can be detected and converted to Promises without leading to failures of the existing test cases of the subject systems.

Out of the 64 modules, we first selected the ones with non-failing tests. We then instrumented the code to filter out subject systems with test cases that covered the asynchronous callbacks in the code. This selection was needed to be able to verify behaviour preservation after the refactoring step. There were 21 subject systems that matched these two criteria, namely passing tests before refactoring and providing callback coverage. We use test cases for this purpose because prior research [27] has shown the effectiveness of test cases for providing an estimate of how reliable refactoring engines are for refactoring tasks on real software projects.

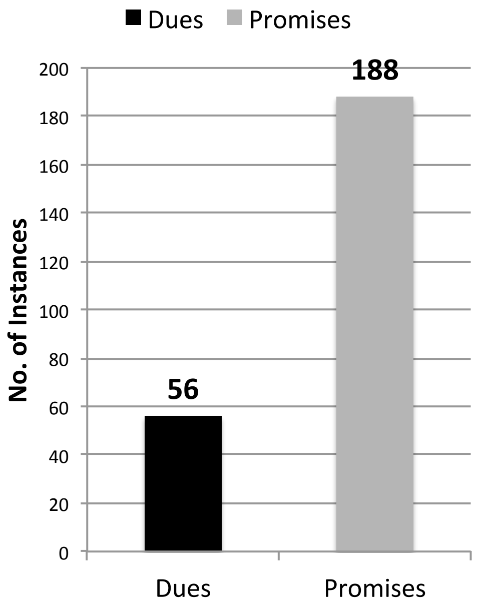
Then refactoring was performed with PROMISESLAND, which analyzed 438 JavaScript files and 108,615 lines of JavaScript code, across all the 21 subject systems. This included the identification of asynchronous callback instances and refactoring them to Promises. After each conversion, we ran the tests to verify whether the original behaviour is preserved.

The results of our comparison between the technique proposed by Brodu et al. [17] (indicated as Dues) and PROMISESLAND are shown in Figure 5.4. In all except two subject systems, PROMISESLAND was able to correctly transform more asynchronous callbacks right away than using the Dues transpiler. The exceptions, namely `express-user-couchdb` and `express-endpoint`, were not compatible with the Node.js 0.12 version, needed for native Promises support. After minor modifications in the dependency declarations of those two projects to depend on the *Bluebird* third party Promises implementation, instead of native Promises, PROMISESLAND was able to refactor with passing tests in these two projects as well.

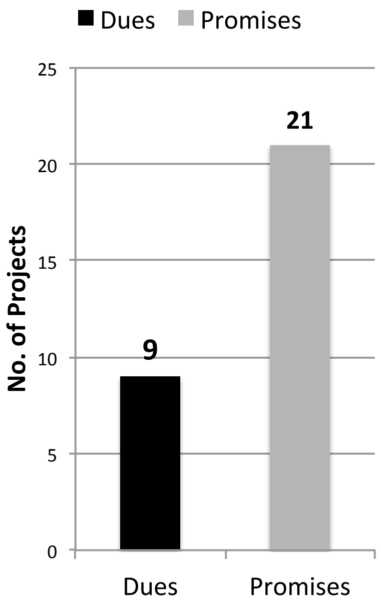
In total, Dues transpiler converted 56 instances, while PROMISESLAND converted 188 instances (including those 56).

Out of the 188 converted instances, 73 were converted using the modify-original strategy and 115 were converted using the wrap-around strategy. When detecting compatible continuations for refactoring, the Dues compiler restricted itself to choose error-first callbacks only. However, PROMISESLAND does not have such constrains and determines the suitability for conversion by analyzing the body of the function itself. Therefore, our approach can select a larger set of asynchronous callbacks for safe conversion.

These results show not only the ability of PROMISESLAND in detecting asynchronous callbacks, but also its correctness in transforming those into Promises.



(a)



(b)

Figure 5.4: The bar chart on the left (a) shows the number of asynchronous callbacks converted into Dues by using the tool from [17] versus the number of asynchronous callbacks converted into Promises with PROMISESLAND. The bar chart on the right (b) shows the number of subject systems in which the tool from [17] was able to detect asynchronous callbacks for converting into Dues versus the number of subject systems in which PROMISESLAND was able to detect asynchronous callbacks for converting into Promises.

5.6.3 Performance (RQ3)

Since the refactoring tool will typically be used in a development environment, the analysis and transformation of the source code are expected to be completed within an acceptable time frame, without keeping the developer idle for too long. In this step, to characterize the performance of PROMISESLAND, we measured the time to analyze and refactor asynchronous callbacks.

Table 5.3 shows the performance statistics of each phase of the refactoring. All the measurements were taken on a system with Dual-core 2.16 GHz CPU and 4GB of RAM, running Linux. In all cases the refactoring completed within 3 seconds. The last row of Table 5.3 shows the time taken for the complete refactoring process end-to-end. Since the migration from asynchronous callbacks to Promises will be a one time task in software maintenance, we believe the time taken by our technique is acceptable and does not hinder the developers' regular work-flow.

Table 5.3: Performance measurements of PROMISESLAND (in seconds)

Phase	**Min**	**Max**	**Mean**	**Median**
Async Function Detection	0.12	1.00	0.51	0.50
Promise Creation Conversion	0.10	0.49	0.29	0.29
Promise Consumption Conversion	0.11	0.47	0.27	0.30
Optimization and Re-writing	0.14	0.95	0.61	0.58
All Phases	0.97	2.57	1.69	1.64

5.7 Discussion

Evaluating PROMISESLAND. We evaluated the correctness of PROMISESLAND by running an application's tests after its code was refactored using the tool. This is a sanity check that the PROMISESLAND maintains program correctness. A more rigorous evaluation would require more formal techniques and is part of our future work.

Evaluating Promises. Although at least some developers prefer promises over asynchronous callbacks, we do not know of any research that considers whether the use of promises improves JavaScript code quality. Our work contributes two refactoring techniques and a tool, PROMISESLAND, that implements these techniques. In our evaluation, we focus on features of the tool, such as its precision and recall. Empirical evaluation of the promises language feature itself and its implications for software quality and developer productivity remains an open problem.

IDE Intergration. By default PROMISESLAND refactors all asynchronous callbacks that it finds in the source code of an application, though it can be also run on a single source file. We believe PROMISESLAND can be integrated into common JavaScript IDEs to make it more easily accessible to developers, which forms part of our future work.

Async and Wait. Promises are specified in the ECMAScript6 specification. ECMAScript7 [5], which is nearing completion, will provide a new option for handling asynchrony in the form of the `async` and `await` keywords. These will allow a linear programming style and permit traditional `try/catch` error handling, which is arguably more understandable than promises and will likely gain fast adoption among developers. However, our perspective is that, regardless of the underlying mechanism for managing asynchrony, the need for detecting and refactoring asynchronous callbacks will remain. The mechanisms described in this chapter and implemented as part of PromisesLand are a first step towards more powerful techniques. Promises explicitly encode success and failure paths, which are implicit in the error-first protocol. With the techniques developed in this chapter, if and when ECMAScript7 is standardized, we will be one step closer to automating the refactoring of JavaScript code to use `async` and `await`.

Backward Compatibility. Although all major JavaScript runtimes support promises, lack of backward-compatibility was a concern that we observed in discussions that we studied (Section 5.2). For example, one developer noted that *"[I] too believe Promises are the future, but it seems that you need to make the users aware of what Promise library they should use or native browser Promises if supported."*[12] In other words, refactoring a library to use promises requires all clients of the library to change their code. Fortunately, PROMISESLAND can be used for this to some extent, but clients must be made aware of this tool.

5.8 Conclusion

It is difficult to imagine a useful JavaScript application that does not use asynchronous callbacks; these are used by applications to respond to GUI events, receive network messages, schedule timers, etc. But, asynchronous callbacks present a number of software engineering challenges, including inability to properly catch and handle errors and callback nesting, which leads developers into "callback hell." In this chapter we presented two refactorings, modify-original and wrap-around, to refactor asynchronous callbacks into promises, a JavaScript language feature that resolves some of the issues with asynchronous callback. We implemented

[12] https://github.com/fixjs/define.js/issues/7

both refactorings as part of the PROMISESLAND tool and evaluate it on 21 large JavaScript applications. We found that PROMISESLAND correctly refactors asynchronous callbacks to promises, refactors 235% more callbacks than a tool from prior work, and runs in under three seconds on all of our evaluation targets.

Chapter 6

Conclusion and Future Work

In the first part of this thesis, we present an empirical study to characterize JavaScript callback usage across 138 large JavaScript projects. These include 86 Node.js modules from the NPM public package registry used in server-side code and 62 subject systems from a broad spectrum of categories, such as JavaScript MVC frameworks, games, and data visualization libraries. Analyzing JavaScript code statically to identify callbacks and to characterize their properties for such a study presents a number of challenges. For example, JavaScript is a loosely typed language and its functions are variadic, i.e., they accept a variable number of arguments. We developed new JavaScript analysis techniques, building on prior techniques and tools, to identify callbacks and to measure their various features.

The focus of our study is on gaining an understanding of callback usage in practice. We study questions such as, how often are callbacks used, how deep are callbacks nested, are anonymous callbacks more common than named callbacks, are callbacks used differently on the client-side as compared to the server-side, and so on. Finally we measure the extent to which developers rely on the "error-first protocol" best practice, and the adoption of two recent proposals to mitigate callback-related challenges, the Async.js library [40] and Promises [12].

The results of our study show that (1) callbacks are passed as arguments more than twice as often in server-side code than in client-side code, i.e., 24% of all server-side call-sites use a callback in server-side code, while only 9% use a callback in client-side code; (2) anonymous callbacks are used in 42% of all callback-accepting function call-sites; (3) there is extensive callback nesting, namely, most callbacks nest 2 levels, and some nest as deep as 8 levels, and (4) there is an extensive use of asynchrony associated with callbacks — 75% of all client-side callbacks were used in conjunction with built-in asynchronous JavaScript APIs.

These results indicate that existing JavaScript analyses and tools [37, 43] often make simplifying assumptions about JavaScript callbacks that might not be true in practice. For example, some of them ignore anonymous callbacks, asynchrony, and callback nesting altogether. Our work stresses the importance of empirically validating assumptions made in the designs of JavaScript analysis tools.

We believe that our characterization of the real-world use of callbacks in different types of JavaScript programs will be useful to tool builders who employ static and dynamic analyses (e.g., which language corner-cases to analyze). Our results will also make language designers more aware of how developers use callback-related language features in practice.

In the second part of this thesis we present a set of program analysis techniques to detect instances of asynchronous callbacks and to refactor such callbacks, including callbacks with the error-first protocol, into promises.

We started by explaining an exploratory study (Section 5.2) in which we examine several GitHub issues and pull-requests containing terms related to refactoring of asynchronous callbacks into promises. We found that developers frequently want to refactor existing code that uses asynchronous callbacks into code that uses promises (GitHub search returned over 4K issues related to this topic). Furthermore, based on our reading of a random sample of these issues, developers have a hard time understanding this refactoring process. GitHub search returned only 451 pull-requests related to this topic (a small number of actual transformations as compared to the number of requests). And, the pull-requests we studied reveal that the most common style of refactoring is project-independent and amenable to automation. Although our study is small, we found no mention or use of refactoring tools: it seems that currently developers manually refactor asynchronous callbacks.

Our exploratory study provides support for the utility of an automated refactoring tool. We propose a set of static analysis techniques that support automated refactoring by: (1) discovering instances of asynchronous callbacks and (2) transforming instances of asynchronous callbacks into promises. We implemented these techniques in a tool called PROMISESLAND and evaluated it on 21 open source JavaScript projects containing a total of 108,615 lines of code. We found that PROMISESLAND performs favorably against recent work [17] that transforms error-first protocol into Dues [16], a simpler (non-standard) form of promises. Specifically, when evaluating PROMISESLAND on projects evaluated in [17], we found that our technique is able to refactor 235% more asynchronous callbacks than the tool proposed in [17]. PROMISESLAND runs in under three seconds on all of the projects we evaluated and we verified the correctness of our implementation by testing all of the refactorings with the test-suites that are distributed with these projects: all the test cases passed after our refactorings, pointing to the behaviour preservation nature of our technique. We also manually studied the code of four of the projects to evaluate the precision and recall of PromisesLand. We found that it has average precision of 100% and average recall of 83%. We believe these results point to the real-world relevance and efficacy of our techniques.

6.1 Future Work

As future work, we plan to improve the techiniques we devised, to gain further insights about JavaScript applications and other areas in software engineering. For example we plan to do further investigations to explain the differences we observed in usages of callbacks in client- side vs server-side JavaScript. Another avenue of future work we are interested in is understanding why developers use different variations of callbacks and which type is used when. With paradigms like Functional Reactive Programming (FRP) gaining traction, we also plan to investigate whether the ways developers are using callbacks is changing. To complement the work of this thesis we also plan on studying how different usages of callbacks impact code quality and how error-prone are different callbacks by investigating correlation of callbacks with bug reports.

Much like in JavaScript, anonymous functions (or Lambda expressions) are present in other programming languages such as C#, Racket, Scheme, Python and Ruby, as well. We plan to extend the techiniques we developed to analyze programs of those other languages and characterize software engineering challenges related to asynchronous execution.

Bibliography

[1] Error Handling in Node.js. https://www.joyent.com/developers/node/design/errors, 2014. Accessed: 2015-11-30. → pages 8

[2] Most depended-upon NMP packages. https://www.npmjs.com/browse/depended, 2014. Accessed: 2015-11-30. → pages 15

[3] Github Showcases. https://github.com/showcases, 2014. Accessed: 2015-11-30. → pages 15

[4] CallMeBack. https://github.com/saltlab/callmeback, 2015. Accessed: 2015-11-30. → pages 14, 31

[5] Status, process, and documents for ECMA262. https://github.com/tc39/ecma262, 2015. Accessed: 2015-11-30. → pages 60

[6] The ECMAScript language specification. http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts, 2015. Accessed: 2015-11-30. → pages 35

[7] ECMA General Assembly Press Release. http://www.ecma-international.org/news/Publication%20of%20ECMA-262%206th%20edition.htm, 2015. Accessed: 2015-11-30. → pages 3

[8] Escope. https://github.com/estools/escope, 2015. Accessed: 2015-11-30. → pages 53

[9] Don't Call Us, We'll Call You: Characterizing Callbacks in JavaScript. Dataset release. http://salt.ece.ubc.ca/callback-study/, 2015. Accessed: 2015-11-30. → pages 14, 24, 31

[10] Can I use Promises? http://caniuse.com/#feat=promises, 2015. Accessed: 2015-11-30. → pages 3

[11] Promisland: implementation and empirical dataset. http://salt.ece.ubc.ca/software/promisland, 2015. Accessed: 2015-11-30. → pages 38, 41, 55

[12] Promises/A+ Promise Specification. https://promisesaplus.com, 2015. Accessed: 2015-11-30. → pages 9, 19, 62

[13] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding JavaScript Event-based Interactions. In *Proceedings of the Intl. Conf. on Software Engineering (ICSE)*, pages 367–377. ACM, 2014. → pages 12, 22

[14] S. Alimadadi, A. Mesbah, and K. Pattabiraman. Hybrid DOM-sensitive change impact analysis for JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 321–345, 2015. → pages 12

[15] E. Andreasen and A. Møller. Determinacy in static analysis for jQuery. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2014. → pages 12

[16] E. Brodu. Due. https://github.com/etnbrd/due, 2015. Accessed: 2015-11-30. → pages 56, 64

[17] E. Brodu, S. Frénot, and F. Oblé. Toward Automatic Update from Callbacks to Promises. In *Proc. of the Workshop on All-Web Real-Time Systems (AWeS)*, pages 1:1–1:8, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3477-8. doi:10.1145/2749215.2749216. → pages ix, 12, 56, 57, 58, 64

[18] M. Bruntink, A. van Deursen, and T. Tourwé. Discovering Faults in Idiom-based Exception Handling. In *Proceedings of the International Conf. on Software Engineering (ICSE)*, pages 242–251. ACM, 2006. → pages 26

[19] B. Cavalier. Async programming part 2: Promises. http://blog.briancavalier.com/async-programming-part-2-promises/, 2013. Accessed: 2015-11-30. → pages 38

[20] D. M. Clements. Decofun. https://github.com/davidmarkclements/decofun. Accessed: 2015-11-30. → pages 11

[21] A. Feldthaus and A. Møller. Semi-automatic rename refactoring for JavaScript. In *Proceedings of the ACM International Conference on Object*

Oriented Programming Systems Languages and Applications (OOPSLA), pages 323–338. ACM, 2013. → pages 13

[22] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip. Tool-supported refactoring for JavaScript. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 119–138, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi:10.1145/2048066.2048078. → pages 13

[23] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 752–761. IEEE, 2013. → pages 12

[24] K. Finley. Github has surpassed sourceforge and google code in popularity. 2011. http://readwrite.com/2011/06/02/github-has-passed-sourceforge. → pages 36

[25] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers. A limit study of JavaScript parallelism. In *Proceedings of Intl. Symposium on Workload Characterization (IISWC)*, pages 1–10, 2010. → pages 11

[26] K. Gallaba, A. Mesbah, and I. Beschastnikh. Don't call us, we'll call you: Characterizing callbacks in JavaScript. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE Computer Society, 2015. → pages iii

[27] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov. Systematic testing of refactoring engines on real software projects. In G. Castagna, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 7920 of *Lecture Notes in Computer Science*, pages 629–653. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-39037-1. doi:10.1007/978-3-642-39038-8_26. → pages 57

[28] L. Gong, M. Pradel, M. Sridharan, and K. Sen. Dlint: Dynamically checking bad coding practices in JavaScript. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 94–105. ACM, 2015. ISBN 978-1-4503-3620-8. doi:10.1145/2771783.2771809. → pages 12

[29] B. Hackett and S.-y. Guo. Fast and Precise Hybrid Type Inference for JavaScript. In *Proceedings of the Conference on Programming Language*

Design and Implementation (PLDI), pages 239–250. ACM, 2012. → pages 15, 46, 53

[30] M. Haverbeke. Tern. https://github.com/marijnh/tern, 2015. Accessed: 2015-11-30. → pages 15, 46, 53

[31] A. Hidayat. Esprima. https://github.com/jquery/esprima, 2015. Accessed: 2015-11-30. → pages 15, 53

[32] S. Hong, Y. Park, and M. Kim. Detecting Concurrency Errors in Client-Side JavaScript Web Applications. In *Proceedings of International Conference on Software Testing, Verification and Validation (ICST)*, pages 61–70. IEEE, 2014. → pages 12

[33] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *Proceedings of Conf. on Comp. and Communications Security*, pages 270–283. ACM, 2010. doi:http://doi.acm.org/10.1145/1866307.1866339. → pages 11

[34] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 34–44. ACM, 2012. ISBN 978-1-4503-1454-1. doi:10.1145/2338965.2336758. → pages 12

[35] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. JSAI: A static analysis platform for JavaScript. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 121–132. ACM, 2014. ISBN 978-1-4503-3056-5. doi:10.1145/2635868.2635904. → pages 12

[36] Y. P. Khoo, M. Hicks, J. S. Foster, and V. Sazawal. Directing JavaScript with arrows. *ACM Sigplan Notices*, 44(12):49–58, 2009. → pages 28

[37] T. Lieber. Theseus: understanding asynchronous code. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 2731–2736. ACM, 2013. → pages 12, 63

[38] M. Madsen, F. Tip, and O. Lhoták. Static analysis of event-driven node.js JavaScript applications. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015. → pages 12

[39] J. Martinsen, H. Grahn, and A. Isberg. A comparative evaluation of JavaScript execution behavior. In *Proceedings of Intl. Conf. on Web Engineering (ICWE)*, pages 399–402. Springer, 2011. → pages 11

[40] C. McMahon. Async.js. https://github.com/caolan/async. Accessed: 2015-11-30. → pages 9, 19, 62

[41] F. Meawad, G. Richards, F. Morandat, and J. Vitek. Eval begone!: semi-automated removal of eval from JavaScript programs. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 607–620. ACM, 2012. → pages 13

[42] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A Programming Language for Ajax Applications. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 1–20, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi:10.1145/1640089.1640091. → pages 28

[43] A. Milani Fard and A. Mesbah. JSNose: Detecting JavaScript Code Smells. In *Proceedings of the International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125. IEEE, 2013. → pages 11, 63

[44] E. Murphy-Hill and A. Black. Breaking the barriers to successful refactoring. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, pages 421–430, May 2008. doi:10.1145/1368088.1368146. → pages 55

[45] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Building call graphs for embedded client-side code in dynamic web applications. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 518–529. ACM, 2014. ISBN 978-1-4503-3056-5. doi:10.1145/2635868.2635928. → pages 12

[46] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In *Proceedings of the Conf. on Computer and Comm. Security*, pages 736–747. ACM, 2012. → pages 11

[47] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. An Empirical Study of Client-Side JavaScript Bugs. In *Proceedings of the ACM/IEEE*

International Symposium on Empirical Software Engineering and Measurement (ESEM), pages 55–64. IEEE, 2013. → pages 11

[48] F. Ocariza, K. Pattabiraman, and A. Mesbah. Detecting inconsistencies in JavaScript MVC applications. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 325–335. ACM, 2015. → pages 12

[49] M. Ogden. Callback Hell. http://callbackhell.com, 2015. Accessed: 2015-11-30. → pages 7, 11, 17

[50] S. Okur, D. L. Hartveld, D. Dig, and A. v. Deursen. A Study and Toolkit for Asynchronous Programming in C#. In *Proceedings of the Intl. Conf. on Software Engineering (ICSE)*, pages 1117–1127. ACM, 2014. → pages 12

[51] V. Raychev, M. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 151–166. ACM, 2013. → pages 12

[52] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM, 2010. → pages 11

[53] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 52–78. Springer, 2011. → pages 11

[54] G. Soares, R. Gheyi, D. Serey, and T. Massoni. Making program refactoring safer. *Software, IEEE*, 27(4):52–57, 2010. → pages 54

[55] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 435–458. Springer-Verlag, 2012. ISBN 978-3-642-31056-0. doi:10.1007/978-3-642-31057-7_20. → pages 12

[56] Stack Overflow. 2015 Developer Survey. http://stackoverflow.com/research/developer-survey-2015, 2015. Accessed: 2015-11-30. → pages 1

[57] G. J. Sussman and G. L. Steele Jr. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998. → pages 6

[58] Y. Suzuki. Estraverse. https://github.com/estools/estraverse, 2015. Accessed: 2015-11-30. → pages 15, 53

[59] TypeScript. TypeScript. http://www.typescriptlang.org, 2015. Accessed: 2015-11-30. → pages 12, 28

[60] S. Wei and B. G. Ryder. State-sensitive points-to analysis for the dynamic behavior of JavaScript objects. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 1–26. Springer, 2014. → pages 12

[61] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. An Empirical Analysis of XSS Sanitization in Web Application Frameworks. Technical Report EECS-2011-11, UC Berkeley, 2011. → pages 11

[62] C. Yue and H. Wang. Characterizing insecure JavaScript practices on the web. In *Proceedings of Intl. Conf. on World Wide Web (WWW)*, pages 961–970. ACM, 2009. doi:http://doi.acm.org/10.1145/1526709.1526838. → pages 11

[63] Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *Proceedings of the International Conference on World Wide Web (WWW)*, pages 805–814. ACM, 2011. → pages 12