Journal of Universal Computer Science, vol. 27, no. 9 (2021), 955-978
submitted: 15/12/2020, accepted: 20/6/2021, appeared: 28/9/2021 CC BY-ND 4.0

Continuations and Aspects to Tame Callback Hell on the Web

Paul Leger
(Escuela de Ingeniería, Universidad Católica del Norte, Coquimbo, Chile
https://orcid.org/0000-0003-0969-5139, pleger@ucn.cl)

Hiroaki Fukuda
(Shibaura Institute of Technology
https://orcid.org/0000-0003-1228-3186, hiroaki@shibaura-it.ac.jp)

Ismael Figueroa
(Pragmatics Lab, http://pragmaticslab.com
https://orcid.org/0000-0003-3661-4963, ifigueroap@gmail.com)

Abstract: JavaScript is one of the main programming languages to develop highly rich responsive and interactive Web applications. In these kinds of applications, the use of asynchronous operations that execute callbacks is crucial. However, the dependency among nested callbacks, known as callback hell, can make it difficult to understand and maintain them, which will eventually mix concerns. Unfortunately, current solutions for JavaScript do not fully address the aforementioned issue. This paper presents Sync/cc, a JavaScript package that works on modern browsers. This package is a proof-of-concept that uses continuations and aspects that allow developers to write event handlers that need nested callbacks in a synchronous style, preventing callback hell. Unlike current solutions, Sync/cc is modular, succinct, and customizable because it does not require ad-hoc and scattered constructs, code refactoring, or adding ad-hoc implementations such as state machines. In practice, our proposal uses a) continuations to only suspend the current handler execution until the asynchronous operation is resolved, and b) aspects to apply continuations in a non-intrusive way. We test Sync/cc with a management information system that administers courses at a university in Chile.

Keywords: Callback Hell, Aspect-Oriented Programming, Continuations, JavaScript
Categories: D.3, D.1, B.5.1
DOI: 10.3897/jucs.72205

1 Introduction

The software industry strongly focuses on rich interactive Web applications running atop the JavaScript engine of modern browsers. Indeed, for Web applications, the use of this programming language is close to 95% [W3 Techs, 2019] and the most used programming language as reported by the Developer Stack Overflow survey 2019 [StackoverFlow, 2019]. JavaScript, a dynamic prototype-based language with higher-order functions, is the *de facto* standard for developing these applications because most modern browsers support it. In the development of these applications, asynchronous programming has been used to hide network latency and improve responsiveness and user interaction. As an example, consider an interactive JavaScript application to read stories online. In this application, every option triggers a *handler*, and one of them (Handler 1) needs the network to retrieve data from a server.

In single-thread languages, such as JavaScript, asynchronous programming developers replace an invocation that *blocks* the whole program execution with a non-blocking invocation that immediately returns. Whenever the asynchronous operation finishes, a function passed as *callback* is executed. Apart from programming, asynchronous programming has been studied widely [Zheng et al., 2011, Tilkov and Vinoski, 2010]. Following our example, Listings 1 and 2 illustrate synchronous programming and contrast it with asynchronous programming using a handler of a Web application that allows users to read stories available on a server when a button is pushed.

```
1 function storyButtonHandler_SYNC() {
2   var URL = "http://storyServer.com?";
3   var idStory = getElementById("idStory").value;
4
5   var story = syncRequest(URL + "id=" + idStory);
6   display(story);
7 }
```

Listing 1: A synchronous version of the story button handler.

```
1 function storyButtonHandler_ASYNC() {
2   var URL = "http://storyServer.com?";
3   var idStory = getElementById("idStory").value;
4   function callback(story) {
5     display(story);
6   }
7   asyncRequest(URL + "id=" + idStory,callback);
8 }
```

Listing 2: An asynchronous version of the story button handler.

In the synchronous programming version of the button handler (Listing 1), the execution of syncRequest blocks the handler work until this execution finishes (Line 5). Instead, in the asynchronous version (Listing 2), when the content of the selected story is downloaded from the server, the callback function is executed to display the story (Line 8). Unlike synchronous programming, the use of asynchronous programming splits a set of statements into an asynchronous operation (asyncRequest) and a callback that receives the operation result (story) and executes the rest of the statements (display(story)). It is important to highlight that asyncRequest corresponds to the native browser mechanism for Ajax [Garrett, 2005], *i.e.,* XMLHttpRequest. Throughout this paper, we use asyncRequest as a generic name for any kind of asynchronous operation.

Although asynchronous programming is used in Listing 2, its implementation is still understandable because there is a single callback with no dependencies. However, this is not always the case in more complex software. For example, consider an application to read books that contain chapters and images:

```
function bookButtonHandler() {
  var URL = "http://bookServer.com?";
  var idBook = getElementById("idBook").value;

  asyncRequest(URL + "id=" + idBook, function(book) {
    //display description about the book on the HTML

    book.chaptersURL.forEach(function(chapURL) {
      asyncRequest(chapURL, function(chapter) {
        //display content of the chapter on the HTML

        chapter.imagesURL.forEach(function(imgURL) {
          asyncRequest(imgURL, function(img) {
            //display images of the chapter on the HTML
          });
        });
      });
    });
  });
}
```

Listing 3: Nested asynchronous calls inside a handler.

In `bookButtonHandler` of Listing 3, we can see nested callbacks because of their dependencies. This is because the two nested asynchronous calls can only be invoked when the response of the previous asynchronous operation is available. These callback dependencies, known as *callback hell* [Ogden, 2019], make it difficult to understand and maintain pieces of code [Loring et al., 2017]. In addition, these (coupled) callback dependencies eventually end up tangling concerns in complex software. For example, `bookButtonHandler` now tangles the download and display concerns. Finally, less obvious issues appear in callback hell: potentially unexpected order of callback executions and unclear control flow of an application execution [Alimadadi et al., 2016]. For example, fig. 1 shows the diagram sequence that a developer must bear in mind to understand the control flow of Listing 3, where each chapter may need to download and show a specific set of images, which should not *a*) change the order in which these images are downloaded and showed and *b*) mix with images of other chapters. Callback hell is widely known in the JavaScript community; indeed, we can find a Website that only focuses on this subject [Ogden, 2019], StackOverflow questions [Overflow, 2020a, Overflow, 2020b], and studies that evaluate different alternatives to address callback hell issues [Gallaba et al., 2015, Kambona et al., 2013].

For Web application development, JavaScript developers can use libraries to address callback dependency issues [Lindesay, 2012, McKenzie, 2014, McMahon, 2010, RxJS, 2018]. Most of them are fairly basic in that they, at best, attempt to hide nested callbacks through the creation of artificial and dependent functions. Other alternatives are specification ECMAScript 7 [ECMA, 2017] or Babel [McKenzie, 2014], which allow developers to write pieces of code in a synchronous manner using specialized language constructs such as `async/await` [Benton et al., 2004]. However, these alternatives scatter these constructs through the code and require the insertion of ad-hoc state machines to control callback executions. In addition, these constructs semantics are fixed for developers.

To address issues that arise from callback hell, this paper presents Sync/cc, a JavaScript package that works on browsers like Mozilla Firefox [Foundation, 2018]. Sync/cc uses continuations [Friedman and Wand, 1984, Haynes et al., 1986] and Aspect-Oriented Programming (AOP) [Kiczales et al., 1996] to allow developers to write asynchronous pieces of code in a synchronous style, preventing callback dependencies and, therefore, modularizing concerns. On the one hand, continuations suspend the asynchronous handler execution—but not the whole program—improving application responsiveness to

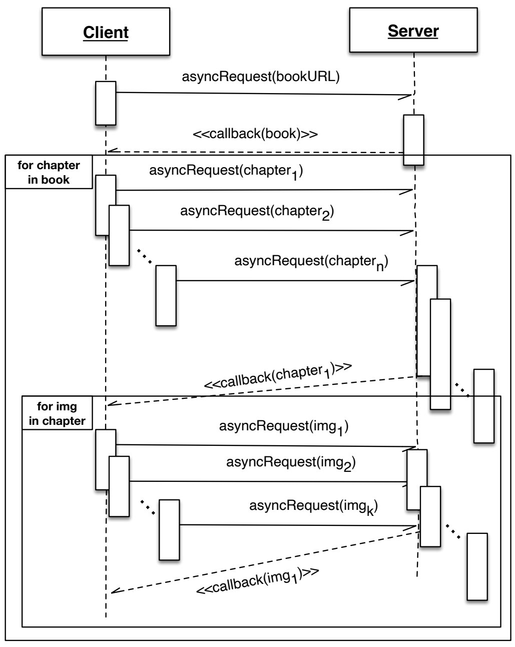


Figure 1: Sequence diagram that shows the control flow of nested callback executions of Listing 3.

users. On the other hand, *aspects*, through the *oblivious interception* of asynchronous operations, allow developers to write clean, sequential-looking code that is executed in the proper asynchronous way. As a result, our work is a practical and applied combination of continuations and aspects, which is a novel combination in the state-of-the-art. Unlike other proposals, Sync/cc:

1. prevents callback hell, that is callback dependencies, tangled and scattered concerns, unexpected order of multiple callback executions, and an unclear control flow of an application execution.
2. provides a modular solution because our proposal is completely supported by continuations and aspects, which are general-purpose features of a programing language. As a consequence, Sync/cc does not require specialized language constructs that are scattered around methods using asynchronous operations.
3. is succinct because the Sync/cc implementation only requires 20 lines of indented code if general or multi purpose abstractions like aspects and continuations are natively supported.
4. is customizable because its implementation utilizes only two user-defined aspects. The complexity of these aspects is hidden through the Sync/cc package.

Although this work have not presented on a conference, the notion of Sync/cc was first presented in two smaller events to receive feedback [Leger and Fukuda, 2016, Leger and Fukuda, 2017]. This current work is the result of a mature idea that allows us to know its *pros* and *cons*; to develop a deep comparison between existing proposals from the state-of-the-art/practice; a concrete implementation of a Sync/cc that now works on both the client and server sides of a Web application; a discussion about implementation details; and a case study in a real Web application. Sync/cc currently uses AspectScript [Toledo et al., 2010], an aspect language for JavaScript, and Unwinder [Long, 2018], a third-party package that supports continuations.

The rest of this paper is structured as follows. The next section reviews current proposals and potential programming language mechanisms that may be used in JavaScript. Section 3 then introduces Sync/cc through a new implementation of the `bookButtonHandler` function. After presenting our proposal, Section 4 describes an application of this proposal in a university management information system. In Section 5, we conclude and describe the major challenges.

Availability. Sync/cc, with the example presented here, is a proof-of-concept that is online and testable at http://pleger.cl/synccc, whereas its source code is available on GitHub: http://github.com/pragmaticslaboratory/synnccc. Our proposal has been tested in Mozilla Firefox (v80.0), Safari (v13.1), and Google Chrome (v85.0) browsers without the need for an extension.

2 Related Work

Issues related to callback hell have already been identified (even in other areas [Zamora-Gómez et al., 2015]), resulting in a number of proposals for the JavaScript language, classified into libraries and specialized language constructs. Taking into account the state-of-the-art/practice, we review these proposals and advances in programming language mechanisms, highlighting their advantages and drawbacks. Finally, we compare and briefly discuss these proposals.

2.1 Libraries

A number of JavaScript lightweight libraries are available on the Web [Lindesay, 2012, McMahon, 2010, ONeal, 2007, Tato, 2010, Farias, 2009], which take advantage of first-class and higher-order functions to mitigate callback hell issues. We illustrate these kinds of solution with an implementation of `bookButtonHandler` using Async.Js [McMahon, 2010]:

```
function bookButtonHandler_ASYNCJS() {
  var URL = //... as above
  var idBook = //... as above
  function downloadAndDisplayBook( continuation ) {
    asyncRequest(URL + "id=" + idBook,function(book) {
      //display description about the book on the HTML
      continuation(book) ;
    }) }

  function downloadAndDisplayChapters(book, continuation ) {
    Async.eachSeries(book.chaptersURL, function(chapter) {
        //display content of the chapter on the HTML
        continuation(chapter) ;
        }),
    ); }

  function downloadAndDisplayImages(chapter, continuation ) {
    chapter.imagesURL.forEach(function(imgURL) {
      asyncRequest(imgURL,function(img) {
        //display images of the chapter on the HTML
        // No more continuations
  });});}); }

  Async.waterfall([downloadBookAndDisplay,
                   downloadAndDisplayChapters,
                   downloadAndDisplayImages]);
}
```

Listing 4: The book handler implementation using Async.Js.

In Listing 4, the Async.waterfall method allows developers to use a *continuation-passing style* pattern [Appel and Jim, 1989] to *visually* hide callback dependencies. However, similar to the original implementation of this handler (Listing 3), developers must be aware of the dependency chain of callbacks through the use of an artificial function (continuation) and use specialized functions (eachSeries) to enforce an order of callback executions. In addition, this package enforces the boilerplate code with a non-common programming style for developers, affecting understandability [Ogden, 2019].

2.2 Programming Language Mechanisms

Programming languages provide several mechanisms that may contribute to the solution of asynchronous issues. We overview the following three widely known mechanisms: promises, observables, and lazy evaluation.

Promises. The key idea is the creation of objects that encapsulate the eventual result of an asynchronous computation. In contrast to standard callbacks, promises can be passed around as first-class values, and can be freely composed using certain operators (*e.g.,* then and catch). Albeit promises were initially supplied by externals libraries, such as PromiseJS [Lindesay, 2012], they are nowadays a standard mechanism of JavaScript [ECMA, 2017]. Listing 5 shows the implementation of the book handler and asynchronous functions using promises. By using the then method, developers can specify *blueprints* of what operations are to be performed after the asynchronous operation, held by a promise, is completed. In the listing, we can see that a first promise, bookPromise, retrieves the book description from the corresponding URL. When this promise finishes, the argument function is executed. Eventually, all chapters and images are retrieved, and, finally, all information is displayed in HTML. Note that this implementation has several issues, similar to those of plain asynchronous operations. First, even though the complexity of callback hell is alleviated by using promises, we find that this kind of code has several nested and/or sequential uses of the `then` method. Indeed, the sole purpose of `then` is to ensure sequential-like behavior in terms of the eventual results of an asynchronous operation. Second, we see that promise composition is not straightforward, in particular, if we need to obtain as much concurrent executions as possible. Third, an unexpected order of callbacks executions can appear in two `forEach` executions. Finally, this piece of code still tangles the downloading and display concerns.

```
function bookButtonHandler_PROMISES_VERSION_1() {
  var URL = // ... as above
  var idBook = // ... as above
  var bookPromise = asyncRequest(URL + "id=" + idBook);

  //an 'empty' promise to start a sequence
  var chapSeqPromise = Promise.resolve();

  bookPromise.then(function(book) {
    //display description about the book on the HTML

    book.chaptersURL.forEach(function(chapterURL) {
      chapSeqPromise = chapSeqPromise.then(function() {
               return asyncRequest(chapterURL);
             }).then(function(chapter) {
                //display content of the chapter on the HTML

                var imgSeqPromise = Promise.resolve();
                chapter.imagesURL.forEach(function(imgURL) {
                  imgSeqPromise.then(function() {
                      return asyncRequest(imgURL);
                  }).then(function(image) {
                    //display images of the chapter on the HTML
                  });
                });
             });
      });
  });
}
```

Listing 5: The book handler implementation using promises.

Using the `Promise.all` method to simplify the use of many consecutive promises and enforce an order of callback executions, Listing 6 shows an alternative implementation with promises of the book handler. This implementation first downloads all the required data (the book description, chapters, and images) into a single data structure, and then displays these pieces of information. Here, we can see that concerns are no longer tangled, but we still get heavily indented and nested use promises similar to callback hell.

```
function bookButtonHandler_PROMISES_VERSION_2() {
  var URL = // ... as above
  var idBook = // ... as above
  var bookPromise = asyncRequest(URL + "id=" + idBook);

  bookPromise.then(function(book) {
    return Promise.all(book.chaptersURL.map(function(chapterURL) {
      return asyncRequest(chapterURL).then(function(chapter) {
        return Promise.all(chapter.imagesURL.map(function(imgURL) {
          return asyncRequest(imgURL);
        }));
      });
    }));
  }).then(function(data) {
    var bookDescription = data[0];
    //display description about the book on the HTML

    var chapters = data[1];
    //display content of the chapter on the HTML

    var images = data[2];
    //display images of the chapter on the HTML
  });
}
```

Listing 6: The book handler implementation using promises to first construct the data.

Reactive Programming. In recent years, there has been an increasing trend in the use of *reactive programming* [Elliott and Hudak, 1997], a programming paradigm to deal with data streams and their change propagation, in the context of interactive JavaScript applications. One of the most widely used libraries is RxJS [RxJS, 2018], which features the concept of *observables*, a novel abstraction for asynchronous programing that uses reactive programming. Observables are first-class objects that emit a data stream from a source such as a server, which may be finite or not, upon which other entities in the application can (un)subscribe to receive and react accordingly. In contrast to promises, observables can emit multiple asynchronous values.

```
function bookButtonHandler_OBSERVABLES() {
  var URL = //... as above
  var idBook = //... as above
  var httpClient = //Observable-based http client
  httpClient.get(URL + "id=" + idBook).subscribe(function(book) {
     //display description about the book on the HTML

     book.chaptersURL.forEach(function(chapURL) {
       httpClient.get(chapURL).subscribe(function(chapter) {
         //display content of the chapter on the HTML

         chapter.imagesURL.forEach(function(imgURL) {
           httpClient.get(imgURL, function(img) {
             //display images of the chapter on the HTML
           });
         });
       });
     });
   });
}
```

Listing 7: The book handler implementation with observables using RxJS.

Listing 7 shows an implementation of bookButtonHandler with observables. In addition to URL and idBook, we create an observable object (httpClient) to make client-side HTTP operations. In the piece of code, we observe that calls to httpClient.get yield observable objects, which support subscriptions. Upon several nested subscriptions, we arrive again at the callback hell problem. In observables, several operators from functional programming (*e.g.,* `map` and `filter`) allow developers to create complex customized data streams and enforce an order of callback executions, but developers are still at risk of falling into callback hell.

Regarding reacting programming approaches in research, we can find the proposals Flapjax [Meyerovich et al., 2009] and Coherent Reaction [Edwards, 2009]. Flapjax is a programming language defined on top of JavaScript. This variant of JavaScript introduces two concepts: *behavior* and *stream event*, which allow developers to create compositions of expressions. A behavior is a *living/reactive* variable, which is automatically updated by an event stream that will give a stream of (infinite) discrete events whose new events trigger additional computation. Once a developer adds a behavior to the event stream, the behavior is updated by the event stream, hiding callbacks. Thus, event handling is managed by the reactive behavior of the language itself (*i.e.,* Flapjax), which updates values that are propagated immediately based on developers-defined dependencies. Although Flapjax hides callbacks from pieces of code, developers have to learn how to use Flapjax and its libraries additionally to a programming style that combines synchronous and asynchronous operations. The second one, Coherent Reaction, introduces the concept of *coherent* in which an ordering of all events are dynamically decided to prevent side effects. Coherent reactions are embedded in a new language. However, this language does not have any concerns about (a)synchronicity but concentrates on correct reactions based on the change of a value.

Lazy evaluation. A programming language uses a strategy that determines when an expression is evaluated, particularly the arguments of a function call. We briefly explain *lazy evaluation*, which may potentially address asynchronous issues by comparing it with *eager evaluation* that many programming languages use, including JavaScript.

```
1 function addOne(num) { return num + 1; }
2 function display(v) { printOnTheScreen(v); }
3
4 display(addOne(10)); // display 11
```

Listing 8: Piece of code used to compare evaluation strategies.

Listing 8 defines the following functions: `addOne`, which returns its argument increased by one, and `display`, which just displays the argument. In Line 4, `display` is invoked with an argument, a value returned from `addOne` with `10`, displaying `11`. In the *eager evaluation* strategy, the argument is aggressively evaluated, meaning that the invocation of `addOne` is done before invoking `display`. As a result, the number `11` is directly passed to `addOne`. In this strategy, the next computation (*e.g.,* `display`) is always blocked until the current computation (*e.g.,* `addOne`) finishes.

In contrast, in the *lazy evaluation* strategy, the evaluation of arguments is deferred when the value is truly required (named *strictness points*). For example, the invocation of `addOne` is not done until the program execution reaches Line 2 where `printOnTheScreen` requires the value of `v`. In this strategy, the next computation (*e.g.,* `display`) may not be blocked even though the current computation (*e.g.,* `addOne`) may have not finished.

```
1 function downloadImage(url) { /* download image */}
2 function init() {
3   var image = downloadImage(url);
4   window.onClick = function(e) { /* click handler on Window */};
5   display(image);
6   }
```

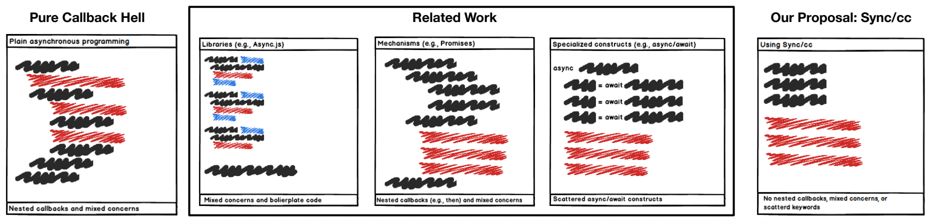
Listing 9: Applying lazy evaluation to a Web application.

Lazy evaluation may help address blocking issues as follows. Listing 9 defines two functions: downloadImage downloads an image and init initializes an application; we assume that downloadImage takes a certain time for downloading. The init function first *a*) downloads an image (Line 3), *b*) adds a handler when there is a "click" on the window (Line 4), and init finally *c*) shows the downloaded image on the screen (Line 5). Given that JavaScript uses eager evaluation, the init execution is blocked until downloadImage finishes, resulting in the click handler not being able to be added. Therefore, applying lazy evaluation enables adding the click handler because image is truly required in Line 6, meaning that the execution of downloadImage is deferred. Although a mechanism that emulates lazy evaluation seems to solve the blocking problem, this kind of mechanism would not work because an asynchronous operation (*e.g.,* downloadImage) does not start until the data is really needed (*e.g.,* display). This implies that the server response can differ from the expected one; for example, when a retrieved image depends on the time.

2.3 Specialized Language Constructs

Babel [McKenzie, 2014] is a JavaScript compiler that allows developers to use non-native features like async/await of C# [Benton et al., 2004]. In addition, these language constructs, with their associated behavior, are supported in the specification ECMAScript 7 [ECMA, 2017], which is already supported by some browsers. In async/await, the tagged async functions can use the await construct, which suspends the current executing function and yields control to the caller of the async method until the awaited asynchronous operation is resolved. In practice, both constructs serve as syntactic sugar to ease programming with promises: async denotes a function that returns a promise, and await waits for the resolution of a promise or async operation (only inside a function that is already async).

Listing 10 shows an implementation of the book button handler using async/await. This implementation shows that the use of async/await permits avoiding callback dependencies and modularizes concerns; for example, the display concern is not inside callbacks. In Line 5, the handler execution is suspended until the value of book is available. In all calls of asyncRequest, the last parameter is now a function with an empty body and can be omitted because this callback is no longer necessary. This is because the handler execution cannot continue until the value is available or an exception is triggered.


Pure Callback Hell
Plain asynchronous programming
Nested callbacks and mixed concerns
Related Work
Libraries (e.g., Async.js)
Mixed concerns and bolierplate code
Mechanisms (e.g., Promises)
Nested callbacks (e.g., then) and mixed concerns
Specialized constructs (e.g., async/await)
async
= await
= await
= await
Scattered async/await constructs
Our Proposal: Sync/cc
Using Sync/cc
No nested callbacks, mixed concerns, or scatterd keywords

(a) Code shape using a native asynchronous mechanism like Ajax.

(b) Code shape using JavaScript libraries like Async.js.

(c) Code shape using mechanisms like Promises.

(d) Code shape using ad-hoc keywords like async/await.

(e) Code shape using Sync/cc.

Figure 2: From a) to d), general code shapes for existing asynchronous programming proposals with their issues: nested callbacks (nest code) result in tangled concerns (color). (a) Typical callback hell, (b) Tangled concerns due to function composition, (c) Heavily nested then *and* subscribe *operations, and (d) Specialized language constructs like async/await results in scattered keywords. Finally, (e) Our proposal, Sync/cc.*

```
async function bookButtonHandler_ASYNC_AWAIT() {
  var URL = //... as above
  var idBook = //... as above
  var book = await asyncRequest(URL + "id=" + idBook, function(){});

  var chapters = book.chaptersURL.map(async function(chapURL) {
        return await asyncRequest(chapURL, function(){});
      });

  var images = chapters.map(function(chapter) {
        return chapter.imagesURL.map(async function(imgURL) {
          return await asyncRequest(imgURL, function(){});
      });});

  //display book description, chapters, and images on the HTML
}
```

Listing 10: The book button handler implementation using async/await.

The async/await constructs unfortunately present an issue: Constructs are scattered around the function body and the callers of this function, especially if asynchronous and synchronous pieces of code are mixed. For example, these two constructs appear six times in the implementation.

From ECMAScript 6 [ECMA, 2016], JavaScript officially introduces the language constructs generator/yield as a core of the language. Inside a generator function, a developer can use the construct yield to keep the stack frame, which can be considered as a continuation. Then, the developer can restart the continuation by invoking the method next() of a generator object created from the execution of the generator function. Constructs generator/yield can solve callback hells; however, the solution presents the same two issues as async/await: scattered specialized constructs over a piece of code and fixed semantics that should be added to the programming language.

Summary. Fig. 2 presents a rough sketch of the shape of the source code in the case of each of the solutions presented here: plain asynchronous programming, JavaScript libraries, mechanisms, and specialized language constructs. The first shape illustrates the

Proposals / Benefits	Plain asynchronous programming	Libraries (e.g., Async)	Mechanisms (e.g., Promises)	Specialized constructs (e.g., async/await)
Natively supported	✓	✓	✓	✓
No nested callbacks		✓		✓
No nested tangled concerns			✓	✓
No require advanced knowledge	✓	✓		
No unexpected callback order (or the order can be forced)		✓	✓	✓
No artificial function required	✓		✓	✓
No scattered (new) constructs	✓	✓	✓	
Synchronous style				✓

Table 1: Comparison summary among existing proposals.

pure callback hell using plain asynchronous programming. The following two shapes present mixed concerns and nested callbacks. A more advanced proposal using specialized constructs for callback hell, such as async/await, does not present nested callbacks but a mix of concerns through scattered ad-hoc keywords. Following the same taxonomy as the figure, Table 1 summarizes different benefits of existing proposals, where we can easily figure out that specialized constructs offer more benefits than the rest of existing proposals. Although existing proposals present a similar behavior in terms of network requests and code execution, we argue that the difficulties for writing, maintaining, and modularizing a piece of code with callback hell provides a solid motivation for our work, especially considering that code indentation and shape have been identified as a good indicator for measuring complexity in terms of software development [Hindle et al., 2009].

3 Sync/cc

Showing a new version of bookButtonHandler, this section presents Sync/cc, which is distributed as a JavaScript package. Unlike previous proposals, Sync/cc does not require nested callbacks, artificial functions, and specialized mechanisms/constructs for callback hell. In addition, our proposal allows developers to customize Sync/cc's semantics because its implementation only needs two aspects, which programmers do not need to understand if Sync/cc's semantics is not modified. Finally, our proposal currently works on both the client and server sides of a JavaScript Web application.

Listing 11 presents the bookButtonHandler implementation with Sync/cc. Every time asyncRequest is called, Sync/cc returns the control to the application avoiding blocking the user experience in the Web application until the response of this request is available. Like async/await, our proposal then resumes the handler execution from the left hand assignment of each asyncRequest call (*e.g.,* var book = ...). As a consequence of the suspension and resumption of a handler execution in each asynchronous request, the order of callback executions is not potentially modified in an unexpected manner if there are more asynchronous requests into the handler. Similar to async/await, the last parameter in the asynchronous operation is now a function with an empty body that can be omitted because this callback is no longer necessary.

```
function bookButtonHandler_SYNC_CC() {
  var URL = //... as above
  var idBook = //... as above
  var book = asyncRequest(URL+"id="+idBook,function(){});

  var chapters = book.chaptersURL.map(function(chapURL) {
        return asyncRequest(chapURL,function(){});
      });

  var images = chapters.map(function(chapter) {
        return chapter.imagesURL.map(function(imgURL) {
          return asyncRequest(imgURL,function(){});
      });});

  //display book description,chapters, and images on the HTML
}
```

Listing 11: The book handler implementation using Sync/cc.

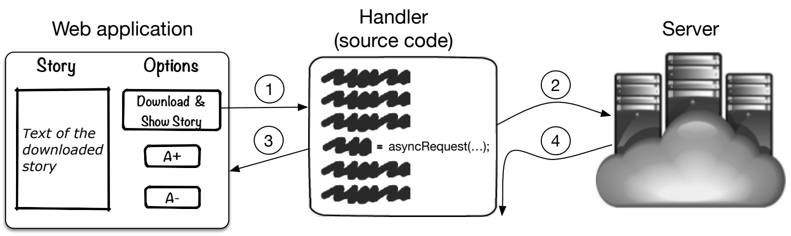


Figure 3: An overview of Sync/cc.

Fig. 3 shows in four steps how Sync/cc works on a Web application. First, the Web application receives a request to execute a handler. Second, the handler invokes an asynchronous operation. Third, Sync/cc suspends the handler execution when this operation is executed and yields control to the Web application. Finally, our proposal resumes the handler execution when the server response is resolved. Like other proposals like async/await, Sync/cc the handler execution is suspended until the server response is available.

3.1 Use of Continuations in Sync/cc

This section briefly introduces continuations and then explains how Sync/cc uses them. Programming languages like Scheme provide continuations [Friedman and Wand, 1984, Haynes et al., 1986], which capture and store the current program control state as a first-class value. If this value is a function, it can be called and the current continuation will be replaced with the stored continuation. Unwinder [Long, 2018] is a JavaScript package, which is currently discounted but is still useful for our purpose, that captures the current continuation with a built-in function, named callCC. We illustrate continuations in Unwinder through a piece of code that captures the execution of a function creating a nickname from a name and lastname (Listing 12):

```
1  var kont;
2
3  function createNick(name,lastname) {
4    return  (function() {
5              kont = callCC(cont => cont);
6              return typeof(kont) == "string"? kont: name + ".";})() +
7              lastname;
8  }
9
10 show(createNick("alan","turing"));                    //shows alan.turing
11 if (typeof(kont) == "function") kont("a");           //shows aturing
```

Listing 12: Use of continuations in the Unwinder package.

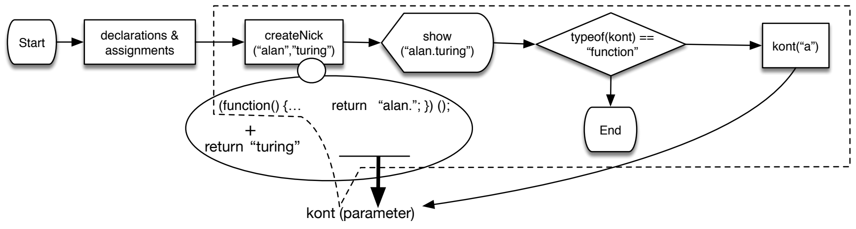

Start
declarations & assignments
createNick ("alan","turing")
show ("alan.turing")
typeof(kont) == "function"
kont("a")
End
(function() {... return "alan."; }) ();
+
return "turing"
kont (parameter)

Figure 4: The Process Flow Diagram (PFD) of Listing 12, which uses the continuation kont*. The invocation of* kont *uses a* parameter *to replace the return value of the anonymous function inside of* createNick *(i.e., "*return "alan"*").*

Fig. 4 shows a flowchart of the piece of code above. In Line 5, the piece of code captures and stores the continuation in kont, bound from cont, before concatenating name with ".". This capture takes place in Line 10 when createNick is called. The result of createNick is passed to show, and then the string "alan.turing" is displayed. Line 11 executes the continuation bound to kont with "a" as parameter, which replaces the return of the anonymous function of Line 4 (*i.e.,* return "alan." → return "a"). As a result, "aturing" is displayed ("aturing" = (name = "a") + (lastname = "turing")). Note that the *if expression* (Line 6) and *if statement* (Line 11) are used to differentiate when a continuation is created from being called. A continuation is a function (Line 9) if this continuation has been created but not called, otherwise the continuation is bound to the value passed by parameter when it is called (*i.e.,* "a" in this piece of code).

If developers need to suspend a handler execution, a continuation should be created at the end of the top-level script because after this execution there is no statements to be executed. For example, Listing 13 shows that the handler execution of showMessageHandler suspends due to the invocation of the continuation suspend:

```
function showMessageHandler() {
  show("This message will be shown");
  suspend();
  show("This message will never be shown");
}

var suspend = callCC(cont => cont);
```

Listing 13: Suspending a JavaScript script with continuations.

Sync/cc uses two continuations: the first one is used to suspend a handler execution just after an asynchronous operation invocation, and the second one is used to resume the handler execution when the asynchronous operation response is available.

3.1.1 Continuations in Programming Languages

Although continuations can be hard to be implemented efficiently [Appel, 1992], some programming languages like C#, Python, and Scala already include these abstractions[1]. In particular, whereas the JavaScript specification [ECMA, 2017] does not include continuations, some interpreters of this language like Rhino [Mozilla, 1997] already include this abstraction in a native way. Additionally, it is possible to find researches oriented to properly implement continuations on top of plain JavaScript [Thivierge and Feeley, 2012].

Unwinder follows a transformation approach to implement continuations. Unsurprisingly, in the JavaScript software ecosystem, including mainstream projects such as TypeScript [TypeScript, 2012], Angular [Angular, 2012], and other packages such as Browserify [Browserling, 2018], Babel [McKenzie, 2014], the usage of (whole-program) transformation pipelines is used in practice as the *de-facto mechanism* for implementation of novel features on top of Vanilla Javascript. Fortunately, as Listing 11 shows, continuations and their transformations are hidden for developers.

As the next section shows, continuations on themselves are not sufficient to tame callback hell. This is because programmers have to be aware of suspending/resuming a program execution, which is similar to the use of async/await in C#.

3.2 Use of Aspects in Sync/cc

This section briefly explains Aspect-Oriented Programming (AOP) [Kiczales et al., 1996] using AspectScript [Toledo et al., 2010]. We then explain how this aspect language is used in Sync/cc. As mentioned previously, developers only need to understand the mechanism behind AOP, when customizations of Sync/cc's semantics are required.

In the pointcut-advice model of AOP, crosscutting behavior is defined by means of *pointcuts* and *advices*, which are encapsulated by an abstraction named *aspect*. Execution points at which an advice may be executed are called *(dynamic) join points*. A pointcut matches a set of join points, and an advice is the action to be taken *before*, *around*, or *after* the matched join point. An around advice can invoke the original computation of the matched join point, known as the *proceed* invocation. Examples of crosscutting behaviors that can be modularized using aspects are security [Toledo and Tanter, 2013], logging [Miles, 2004], and event handling [Leger et al., 2013].

In AspectScript, an aspect is a JavaScript object with three properties: a pointcut, an advice, and an advice kind. The two first properties are functions that receive a join point object as parameter. The advice kind property is a constant that indicates when the advice is executed (*i.e.,* before, around, or after). For example, the notificationOfBookHandler aspect of Listing 14 logs a message after each execution of the bookButtonHandler function. In this aspect, the pointcut matches join points that represent executions of bookButtonHandler and the advice, which is executed after each bookButtonHandler execution, logs a message.

[1] https://en.wikipedia.org/wiki/Continuation

```
var notificationOfBookHandler = {
  pointcut: function(jp) {
    //if jp is an execution of the bookButtonHandler function
    return jp.isExec() && jp.fun == bookButtonHandler
  },

  advice: function(jp) {
    //function executed when jp is matched
    log("bookButtonHandler was executed");
  },

  kind: AFTER
};

//deployment of the aspect
AspectScript.deploy(notificationOfBookHandler);
```

Listing 14: An aspect that shows a message when a handler is executed.

Listing 15 shows the complete implementation of Sync/cc, which uses two aspects. The first aspect executes its advice around an asynchronous operation call. The second aspect triggers its advice after the execution of the associated callback to the asynchronous operation; in the book handler example, Listing 11, the callback is the second and last parameter of `asyncRequest`.

```
var Synccc = function() {
  this.kont; //to resume an handler execution
  this.suspend; //to suspend an handler execution

  this.asyncOperation = {
      pointcut: function(jp) {
        return jp.isCall() && jp.fun == asyncRequest;},
      advice: function (jp) {
        var response = ① jp.proceed(); //executing asyncRequest

        ② kont = callCC(cont => cont);

        if (typeof(kont) == "function") {
          ③ suspend();
        }
        else {
          return kont;
        }
      },
      kind: AROUND
  };

  this.asyncCallback = {
      pointcut: function(jp) {
        return jp.isExec() && jp.fun == callback
      },
      advice: function (jp) {
        var response = jp.args[0]; //1st arg to callback
        kont(response); //resuming the asyncOperation advice
      },
      kind: AFTER
    };
  }
}
```

Listing 15: Implementation of Sync/cc.

The `asyncOperation` aspect matches the call of `asyncRequest` and its around advice carries out three tasks. The task ① executes the original computation. The task ② creates a continuation (kont) that represents the execution of this advice and the rest of the handler execution. Finally, the advice suspends the handler execution using the suspend continuation (task ③). The asyncCallback aspect matches the callback execution, and its advice resumes the kont continuation with the response of the asynchronous operation. As a consequence, the execution of the asyncOperation advice, which replaces the asyncRequest execution, is resumed to return the already available response of the operation (Fig. 5). Thereby, the implementation of bookButtonHandler with Sync/cc (Listing 11) can now use the Array.map method instead of Array.forEach, which is used in the original implementation of this handler (Listing 3) because it only iterates the array without returning a new array as a result.

3.2.1 Aspects in Programming Languages

In the literature, we can find various aspect languages, for example, AspectJ [Kiczales et al., 2001] for Java, AspectS [Hirschfeld, 2002] for Squeak, AspectScheme [Dutchyn et al., 2006] for Scheme, and AspectScript for JavaScript. Similar to Unwinder and other relevant projects mentioned in Section 3.1.1, AspectScript also follows a transformation of the source code of a program.

As the previous section shows, aspects on themselves are not sufficient to tame callback hell. This is because programmers have to suspend/resume a handler execution when an asynchronous request is called, and its result is used.

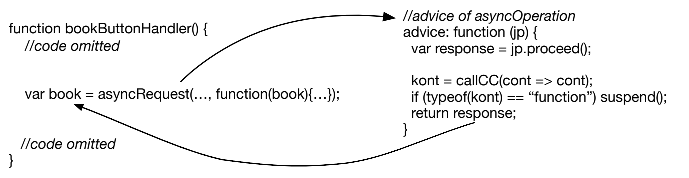
3.3 Semantics Customization of Sync/cc

Developers can use the Sync/cc package without knowing how AOP works, especially these two aspects. Nevertheless, if developers require customize Sync/cc's semantics, some AOP knowledge is required. Next, we explain what customizations can be done when developers modify these aspects.

Although aspects of Listing 11 show that Sync/cc only works with asyncRequest, our proposal allows developers to customize what asynchronous operations intercept. For this customization, developers only need to change the functions that must be matched by pointcuts. Indeed, an application interface as a function SyncCC.setAsyncOperation(..) would enable Sync/cc works on different kinds of asynchronous operations available on frameworks such as JQuery, Prototype, YUI, MooTools, etc. In addition, our proposal can be adapted to different and unforeseen semantics of asynchronous operations. We briefly explain two examples where Sync/cc customizations may be useful. First, handling exceptions inside callbacks is challenging because of its unclear propagation [Ploski and Hasselbring, 2005]. Using Sync/cc, we may customize the asyncCallback aspect (Listing 15) to trigger/propagate an exception when, for example, the server response is *timeout*. Second, consider an asyncRequestFromMultipleServers method which sends multiple requests to different servers and only uses the first server response that is received (*e.g.,* a geo-location service). By modifying both previous aspects used to implement Sync/cc, our proposal can discard all server responses after the first response.

3.4 Implementation Details

As aspects and continuations are not natively supported in JavaScript, it is necessary to use some libraries available on NPM [NPM, 2018], a large repository of JavaScript



```
function bookButtonHandler() {
  //code omitted

  var book = asyncRequest(..., function(book){...});

  //code omitted
}
```

```
//advice of asyncOperation
advice: function (jp) {
  var response = jp.proceed();

  kont = callCC(cont => cont);
  if (typeof(kont) == "function") suspend();
  return response;
}
```

Figure 5: Behavior of the asyncOperation *advice to return a value to the assignment of an asynchronous operation.*

Overhead/Example	Story	Book
(Server-side) Performance with only CPU usage	300%	
(Client-side) Performance with network latency	6.6%	13.3%

Table 2: Sync/cc performance overload with story and book examples. First, overload considering network latency, and then only CPU usage.

libraries available as packages. This section describes the use of these libraries to support Sync/cc in Web applications.

Fig. 6 shows the workflow used to support in Sync/cc. This workflow depicts the use of AspectScript and Unwinder. To use our proposal, a JavaScript programmer instruments to *instrument* a script that contains asynchronous requests to add aspects and continuations support. Then, the workflow utilizes the Browserify package [Browserling, 2018] to bundle in only one JavaScript file that is compatible with existing browsers (*e.g.,* Mozilla Firefox).

Table 2 shows a performance evaluation of Sync/cc in the server and client side. For both evaluations, we used Mozilla Firefox (v70.x) [Foundation, 2018] on a Macbook Pro (2017), 3.1 GHz Dual-Core Intel Core i5 with 8GB of RAM running macOS Catalina. The table shows that the runtime of aspect and continuations affects performance, that is the server-side. Although performance results are important, they do not really affect handlers of interactive JavaScript applications because of asynchronous operation latency. In the client-side, we have tested the two examples presented here (story and book handlers) on the Sync/cc Website without any noticeable difference (between 6.6% and 13.3%) over the Web application without Sync/cc. In addition, as aspects and continuations are general purpose abstractions, Sync/cc may not need any kind of instrumentation or additional libraries to work if these abstractions are natively supported. Finally, note these values are for reference only because this is not an optimized production-ready implementation, meaning that it is useful to see the orders of magnitude rather than specific performance numbers.

4 Case Study: A Management Information System to Administer University Courses

The Catholic University of the North (http://www.ucn.cl) is located in two cities of Chile. As of 2017, this university is home to 11,164 students, and has 38 undergraduate program studies that are lectured in 35 academic units. Courses administration is carried

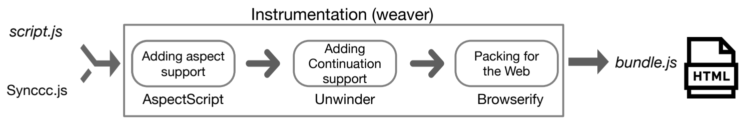


Figure 6: Workflow of libraries used to support aspects and continuations in Sync/cc.

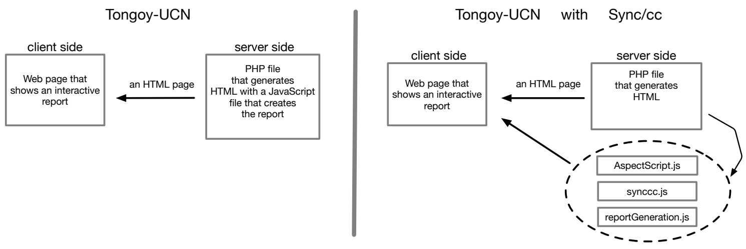

Tongoy-UCN
client side
server side
Web page that shows an interactive report
an HTML page
PHP file that generates HTML with a JavaScript file that creates the report
Tongoy-UCN with Sync/cc
client side
server side
Web page that shows an interactive report
an HTML page
PHP file that generates HTML
AspectScript.js
synccc.js
reportGeneration.js

Figure 7: The current Tongoy-UCN implementation versus a version that applies Sync/cc.

out by a Management Information System (MIS) [Laudon and Laudon, 2016] called Tongoy-UCN [Ross, 2018]. This system uses Web technologies to allow *a*) students to register their attendance, access their grades, and comment lectures, and *b*) professors to take attendance, publish grades, and define a course plan. In addition, Tongoy-UCN is able to create different kinds of reports. This Web system approximately administers 2,500 courses per semester that are lectured in 33 academics units (94.2%) at the university.

One of Tongoy-UCN's features is showing a report about a running course in a Web page. This report includes the grades and attendance of students, as well as the state of a specific course. The creation of a course report requires information from four different sources in Tongoy-UCN (*i.e.,* tables of a database). This report is generated by a script that is within an HTML Web page (client-side). To generate the report, this script executes four nested asynchronous operations, where each one receives information of a source (Listing 16). These asynchronous operations illustrate *callback hell*. Note that although these four information requests come from the same database, creating only one request that receives all information would require an ad-hoc modification of the Tongoy-UCN backend to only implement this feature in the front-end, where this script is executed.

```
function reportGeneration(parameters) {
  //Variables and functions are translated to English
  //For clarification, some piece of code are subtly modified

  asyncRequest("professor/pbd.php?op=ea",function(sAttendance){
    asyncRequest("professor/pbd.php?op=en",function(sDegree){
      asyncRequest("professor/pbd.php?op=es",function(cState){
        asyncRequest("professor/pbd.php?op=list",function(cInstance){

          //process retrieved information
          displayReport(/* processed information */);
        });
      });
    });
  });
}
```

Listing 16: Callback hell in the generation of a course report in Tongoy-UCN.

Overhead/Example	Tongoy's script
Script length	3,081%
Performance with network latency	15.3%

Table 3: Sync/cc overload in its application in Tongoy.

We applied Sync/cc in a development version of Tongoy-UCN to verify if our proposal can address issues related to callback hell. Listing 17 illustrates this application in the generation of a course report in Tongoy-UCN. In this listing, we can easily see that the four nested asynchronous operations disappear, and empty callbacks could be removed. In addition, we can see that each asynchronous operation returns and binds the required information to a variable. Unlike existing proposals discussed in related work (Section 2), we can see this solution does not require:

– additional (and artificial) functions,

– any nested function declarations,

– to enforce an order of callback executions, and

– to scatter specialized (and ad-hoc) language constructs in the piece of code.

```
function reportGeneration_SYNCCC(parameters) {
  //Variables and functions are translated to English
  //For clarification, some piece of code are subtly modified

  var sAttendance, sDegree, cState, cInstance;

  sAttendance = asyncRequest("professor/pbd.php?op=ea");
  sDegree = asyncRequest("professor/pbd.php?op=en");
  cState = asyncRequest("professor/pbd.php?op=es");
  cInstance = asyncRequest("professor/pbd.php?op=list");

  //process retrieved information
  displayReport(/* processed information */);
}
```

Listing 17: The course report implementation using Sync/cc in Tongoy-UCN.

Because of the current Tongoy-UCN implementation, the integration between Sync/cc and Tongoy-UCN required a small refactoring. Fig. 7 shows the refactoring in this system component to use Sync/cc, which adds the AspectScript runtime and Sync/cc libraries. Because our current proposal requires code instrumentation to work, we separated the instrumented code in a single JavaScript file (generationReport.js) to prevent confusion in a PHP programmer because the current Tongoy-UCN implementation mixes client and serve code. As Table 3 shows, its increment in terms of script length and performance considering network latency remain similar to the book example shown in Table 2; this is because both scripts have a potentially equivalent number of nested callbacks. Finally, although someone may suggest that this refactoring illustrates a sign of weakness of Sync/cc, we argue that this refactoring is even recommended in the current Tongoy-UCN implementation because the server-side (PHP) and client-side (JavaScript) concerns are now mixed.

5 Conclusions

Because there is a strong push to build complex, interactive, and highly responsive JavaScript applications, appropriate abstractions for asynchronous programming are crucial. In this kind of programming, callbacks are widely used nowadays. Unfortunately, callback dependencies (*a.k.a.* callback hell) are likely to appear and make it difficult to eventually understand and maintain pieces of code that crosscut concerns. Current proposals do not fully address callback dependencies. Using general and multi-purpose concepts in programming languages, this paper proposes Sync/cc, a JavaScript package to prevent callback hell using continuations and aspects. Continuations prevent the need to execute statements in callback bodies and aspects apply continuations without the need to modify the code.

Regarding future work, we plan to conduct usability tests to claim this proposal is a good option for developers that must deal with asynchronous programming. For example, developers may make errors when are maintaining a module that uses an asynchronous operation because the absence of some construct (*e.g.,* async/await) or framework that indicates this kind of operation. Similar to the user-study applied in [Rossbach et al., 2010], we plan to carry out a usability evaluation where a set of developers will modify a module of a Web application. This module will use asynchronous operations and developers will modify it using existing proposals and Sync/cc in order to compare them.

Acknowledgments

We are grateful to Dominique Leger for her help in clarifying our initial intuition. In addition, we thank Felipe Ruiz (felipe.ruiz@alumnos.ucn.cl) and Erik Astorga (eab008@alumnos.ucn.cl) for their efforts to develop a working and usable version of Sync/cc in modern browsers.

References

[Alimadadi et al., 2016] Alimadadi, S., Mesbah, A., and Pattabiraman, K. (2016). Understanding asynchronous interactions in full-stack JavaScript. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1169–1180, Austin, USA.

[Angular, 2012] Angular (2012). Angular: A typed JavaScript. https://angular.io/. Accessed: 2021-01-20.

[Appel, 1992] Appel, A. W. (1992). *Compiling with Continuations*. Cambridge University Press.

[Appel and Jim, 1989] Appel, A. W. and Jim, T. (1989). Continuation-passing, closure-passing style. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 89)*, pages 293–302, Austin, USA.

[Benton et al., 2004] Benton, N., Cardelli, L., and Fournet, C. (2004). Modern concurrency abstractions for C$^\sharp$. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804.

[Browserling, 2018] Browserling (2018). Browserify: A tool for compiling node-flavored commonjs modules for the browser. http://browserify.org. Accessed: 2020-01-20.

[Dutchyn et al., 2006] Dutchyn, C., Tucker, D. B., and Krishnamurthi, S. (2006). Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239.

[ECMA, 2016] ECMA (2016). ECMAScript 6: A scripting-language specification for JavaScript - https://www.ecma-international.org/ecma-262/6.0. Accessed: 2020-01-20.

[ECMA, 2017] ECMA (2017). ECMAScript 7: A scripting-language specification for JavaScript - https://www.ecma-international.org/ecma-262/7.0. Accessed: 2020-01-20.

[Edwards, 2009] Edwards, J. (2009). Coherent reaction. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, pages 925–932, Orlando, Florida, USA.

[Elliott and Hudak, 1997] Elliott, C. and Hudak, P. (1997). Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, pages 263–273, Amsterdam, The Netherlands.

[Farias, 2009] Farias, B. (2009). A utility to hoist nested callbacks. Accessed: 2020-01-20.

[Foundation, 2018] Foundation, T. M. (2018). Firefox: A free and open-source web browser. Accessed: 2020-01-20.

[Friedman and Wand, 1984] Friedman, D. P. and Wand, M. (1984). Reification: Reflection without metaphysics. In *Proceedings of the Annual ACM Symposium on Lisp and Functional Programming*, pages 348–355.

[Gallaba et al., 2015] Gallaba, K., Mesbah, A., and Beschastnikh, I. (2015). Don't call us, we'll call you: Characterizing callbacks in JavaScript. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10, Beijing, China.

[Garrett, 2005] Garrett, J. J. (2005). Ajax: A new approach to Web applications. https://adaptivepath.org/ideas/ajax-new-approach-web-applications/. Accessed: 2020-01-20.

[Haynes et al., 1986] Haynes, C., Friedman, D., and Wand, M. (1986). Obtaining coroutines with continuations. *Computer Languages*, 11(3):143–153.

[Hindle et al., 2009] Hindle, A., Godfrey, M. W., and Holt, R. C. (2009). Reading beside the lines: Using indentation to rank revisions by complexity. *Science of Computer Programming*, 74(7):414–429.

[Hirschfeld, 2002] Hirschfeld, R. (2002). AspectS – aspect-oriented programming with Squeak. In Akşit, M., Mezini, M., and Unland, R., editors, *International Conference NetObjectDays on Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *Lecture Notes in Computer Science*, pages 216–232. Springer-Verlag.

[Kambona et al., 2013] Kambona, K., Boix, E. G., and De Meuter, W. (2013). An evaluation of reactive programming and promises for structuring collaborative web applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications (DYLA)*, pages 1–9, Montpellier, France.

[Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. (2001). An overview of AspectJ. In Knudsen, J. L., editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary. Springer-Verlag.

[Kiczales et al., 1996] Kiczales, G., Irwin, J., Lamping, J., Loingtier, J., Lopes, C., Maeda, C., and Mendhekar, A. (1996). Aspect oriented programming. In *Special Issues in Object-Oriented Programming*. Max Muehlhaeuser (general editor) et al.

[Laudon and Laudon, 2016] Laudon, K. and Laudon, J. (2016). *Management information system*. Pearson Education India.

[Leger and Fukuda, 2016] Leger, P. and Fukuda, H. (2016). Using continuations and aspects to tame asynchronous programming on the web. In *Proceedings of the Workshop on Foundations of Aspect-Oriented Languages (FOAL 2016)*, pages 79–82, Malaga, Spain.

[Leger and Fukuda, 2017] Leger, P. and Fukuda, H. (2017). Sync/cc: Continuations and aspects to tame callback dependencies on JavaScript handlers. In *Proceedings of the 32th Annual ACM Symposium on Applied Computing (SAC 2017)*, pages 1245–1250, Marrakech, Morocco.

[Leger et al., 2013] Leger, P., Tanter, É., and Douence, R. (2013). Modular and flexible causality control on the web. *Science of Computer Programming*, 78(9):1538–1558.

[Lindesay, 2012] Lindesay, F. (2012). Promise: A library for promises in JavaScript. https://www.promisejs.org. Accessed: 2020-01-20.

[Long, 2018] Long, J. (2018). Unwinder: A call/cc library. https://www.npmjs.com/package/unwinder-engine. Accessed: 2021-04-20.

[Loring et al., 2017] Loring, M., Marron, M., and Leijen, D. (2017). Semantics of asynchronous JavaScript. In *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages*, pages 51–62, Vancouver, BC, Canada.

[McKenzie, 2014] McKenzie, S. (2014). Babel: A compiler for writing ES6 and ES7 generation JavaScript. https://babeljs.io. Accessed: 2020-01-20.

[McMahon, 2010] McMahon, C. (2010). Async.Js: A library for working with asynchronous JavaScript. https://github.com/caolan/async. Accessed: 2020-01-20.

[Meyerovich et al., 2009] Meyerovich, L., Guha, A., Baskin, J., Cooper, G., Greenberg, M., Bromfield, A., and Krishnamurthi, S. (2009). Flapjax: A programming language for ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 1–20, Orlando, Florida, USA.

[Miles, 2004] Miles, R. (2004). *AspectJ Cookbook: Aspect Oriented Solutions to Real-World Problems*. O'Reilly Media.

[Mozilla, 1997] Mozilla (1997). Rhino: An open-source implementation of JavaScript written entirely in Java. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino. Accessed: 2021-04-10.

[NPM, 2018] NPM (2018). NPM: A package manager online for JavaScript. https://www.npmjs.com. Accessed: 2020-01-20.

[Ogden, 2019] Ogden, M. (2019). A website about callback hell. http://callbackhell.com. Accessed: 2019-01-03.

[ONeal, 2007] ONeal, A. (2007). FuturesJS: Asynchronous library in JavaScript using futures. https://github.com/FuturesJS. Accessed: 2020-01-20.

[Overflow, 2020a] Overflow, S. (2020a). Callback hell on stackoverflow: How to avoid nested asynchronous operations. https://stackoverflow.com/questions/4234619/how-to-avoid-long-nesting-of-asynchronous-functions-in-node-js. Accessed: 2020-01-17.

[Overflow, 2020b] Overflow, S. (2020b). Callback hell on stackoverflow: Problems with asynchronous operations. https://stackoverflow.com/questions/14220321/how-do-i-return-the-response-from-an-asynchronous-call/14220323. Accessed: 2020-01-17.

[Ploski and Hasselbring, 2005] Ploski, J. and Hasselbring, W. (2005). The callback problem in exception handling. In *Proceedings of ECOOP Workshop on Exception Handling in Object-Oriented Systems*, pages 39–62, Montpellier, France.

[Ross, 2018] Ross, E. (2018). Tongoy-UCN: A management information systems to admin university's courses. https://tongoy.ucn.cl. Accessed: 2020-01-20.

[Rossbach et al., 2010] Rossbach, C., Hofmann, O., and Witchel, E. (2010). Is transactional programming actually easier? In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 47–56, Bangalore, India.

[RxJS, 2018] RxJS (2018). Reactive extensions for JavaScript. Accessed: 2020-01-20.

[StackoverFlow, 2019] StackoverFlow (2019). Developer survey results. https://insights.stackoverflow.com/survey/2019. Accessed: 2020-01-20.

[Tato, 2010] Tato, L. (2010). Wait.for: Sequential programming for NodeJs/JavaScript. https://github.com/luciotato/waitfor. Accessed: 2020-01-20.

[Thivierge and Feeley, 2012] Thivierge, E. and Feeley, M. (2012). Efficient compilation of tail calls and continuations to javascript. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*, pages 47–57, Copenhagen, Denmark. ACM.

[Tilkov and Vinoski, 2010] Tilkov, S. and Vinoski, S. (2010). Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83.

[Toledo et al., 2010] Toledo, R., Leger, P., and Tanter, É. (2010). AspectScript: Expressive aspects for the Web. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, pages 13–24, Rennes and Saint Malo, France. ACM Press.

[Toledo and Tanter, 2013] Toledo, R. and Tanter, É. (2013). Secure and modular access control with aspects. In Kinzle, J., editor, *Proceedings of the 12th International Conference on Aspect-Oriented Software Development (AOSD 2013)*, pages 157–170, Fukuoka, Japan. ACM Press.

[TypeScript, 2012] TypeScript (2012). TypeScript: A typed JavaScript. https://www.typescriptlang.org/. Accessed: 2021-04-10.

[W3 Techs, 2019] W3 Techs (2019). Usage of client-side programming languages. https://w3techs.com/technologies/history_overview/client_side_language/all. Accessed: 2020-01-20.

[Zamora-Gómez et al., 2015] Zamora-Gómez, E., García-López, P., and Mondéjar, R. (2015). Continuation complexity: A callback hell for distributed systems. In *European Conference on Parallel Processing (EURO-PAR)*, pages 286–298, Vienna, Austria.

[Zheng et al., 2011] Zheng, Y., Bao, T., and Zhang, X. (2011). Statically locating web application bugs caused by asynchronous calls. In *Proceedings of the 11th International World Wide Web Conference (WWW 2011)*, pages 805–814, Hyderabad, India.