# Automatically update your callbacks into Promises

Etienne Brodu, Stéphane Frénot
*firstname.lastname*@insa-lyon.fr
Université de Lyon, INRIA,
INSA-Lyon, CITI-INRIA, F-69621, Villeurbanne, France

Frédéric Oblé
frederic.oble@worldline.com
Worldline
53 avenue Paul Krüger - CS 60195
69624 Villeurbanne Cedex

## Categories and Subject Descriptors
D.3.4 [**Software Engineering**]: Processors—*Code generation, Compilers, Run-time environments*

## General Terms
Compilation

## Keywords
Flow programming, Web, Javascript

## Abstract

## 1. INTRODUCTION
The world wide web started as a document sharing platform for academics. It is now a rich application platform, pervasive, and accessible almost everywhere. This transformation began in Netscape 2.0 with the introduction of Javascript, a web scripting language.

Javascript was originally designed for the manipulation of a graphical environment : the Document Object Model (DOM [1] ). Functions are first class-citizens ; it allows to manipulate them like any object, and to link them to react to asynchronous events, *e.g.* user inputs and remote requests. These asynchronously triggered functions are named callbacks, and allow to efficiently cope with the distributed and inherently asynchronous architecture of the Internet. This made Javascript a language of choice to develop both client and, more recently, server applications for the web.

Callbacks are well suited for small interactive scripts. But in a complete application, they are ill-suited to control the larger asynchronous execution flow. Their use leads to intricate imbrications of function calls and callbacks, commonly presented as *callback hell*[2] , or *pyramid of Doom*. This is widely recognized as a bad practice and reflects the unsuitability of callbacks in complete applications. Eventually, developers enhanced callbacks to meet their needs with the concept of Promise[8].

Promises bring a different way to control the asynchronous execution flow, better suited for large applications. They fulfill this task well enough to be part of the next version of the Javascript language. However, because Javascript started as a scripting language, beginners are often not introduced to Promises early enough, and start their code with the classical Javascript callback approach. Moreover, despite its benefits, the concept of Promise is not yet widely acknowledged. Developers may implement their own library for asynchronous flow control before discovering existing ones.There is such a disparity between the needs for and the acknowledgment of Promises, that there is almost 40 different implementations[3] .

With the coming introduction of Promise as a language feature, we expect an increase of interest, and believe that many

1. `http://www.w3.org/DOM/`
2. `http://maxogden.github.io/callback-hell/`
3. `https://github.com/promises-aplus/promises-spec/blob/master/implementations.md`

developers will shift to this better practice. In this paper, we propose a compiler to automate this shift in existing code bases. We present the transformation from an imbrication of callbacks to a sequence of Promise operations, while preserving the semantic.

Promises bring a better way to control the asynchronous execution flow, but they also impose a conditional control over the result of the execution. Callbacks, on the other hand, leave this conditional control to the developer. This paper focuses on the transformation from imbrication of callbacks to chain of Promises. To avoid unnecessary modifications on this conditional control, we introduce an alternative to Promises leaving this conditional control to the developer, like callbacks. We call this alternative specification Dues. This approach enables us to compile legacy Javascript code from repositories and brings a first automated step toward full Promises integration. This simple and pragmatic compiler has been tested over $n$ Github repositories, $k$ of which with success. todo

In section 2 we define callbacks and Promises. We then introduce Dues in section 2.4. In section 3, we explain the transformation from callback imbrications to dues sequences. In section 4, we present an implementation of this transformation. In section 5, we evaluate this compiler. We present related works in section 6, and finally conclude in section 7.

## 2. DEFINITIONS

### 2.1 Callback

A callback is a function passed as a parameter to a function call. It is invoked by the callee to continue the execution with arguments not available in the caller context. We distinguish three kinds of callbacks.

**Iterators** are functions called for each item in a set, often synchronously.

**Listeners** are functions called asynchronously for each message in a stream.

**Continuations** are functions called asynchronously once a result is available.

As we will see later, Promises are designed as placeholders for a unique outcome. Iterators and Listeners are invoked multiple times resulting in multiple outcomes. Only continuations are equivalent to Promises. Therefore, we focus on continuations in this paper.

Callbacks are often mistaken for continuations ; the former are not inherently asynchronous while the latter are. In a synchronous paradigm, the sequentiality of the execution flow is trivial. An operation needs to complete before executing the next one. On the other hand, in an asynchronous paradigm, parallelism is trivial, operations are executed in parallel. The sequentiality of operations needs to be explicit. Continuations are the functional way of providing this control over **the sequentiality of the asynchronous execution flow**.

A **continuation** is a function passed as an argument to allow the callee not to block the caller until its completion. The continuation is invoked later, at the termination of the callee to continue the execution as soon as possible and process the result ; hence the name continuation.

When using continuation, the convention on how to hand back the result must be common for both the callee and the caller. For example, in *Node.js*, the signature of a continuation uses the *error-first* convention. The first argument contains an error or `null` if no error occurred ; then follows the result. Listing 1 is a pattern of such a continuation. However, continuations don't impose any conventions ; indeed, other conventions are used in the browser.

```
my_fn(input, function continuation(error, result) {
  if (!error) {
    console.log(result);
  } else {
    throw error;
  }
});
```

**Listing 1: Example of a continuation**

The callback hell occurs when many asynchronous calls are arranged to be executed sequentially. Each consecutive operation adds an indentation level, because it is nested inside the continuation of the previous operation. It produces an imbrication of calls and function definitions, like in listing 2. We say that continuations lack the **chained composition** of multiple asynchronous operations. Promises allow to arrange such a sequence of asynchronous operations in a more concise and readable way.

```
my_fn_1(input, function cont(error, result) {
  if (!error) {
    my_fn_2(result, function cont(error, result) {
      if (!error) {
        my_fn_3(result, function cont(error, result) {
          if (!error) {
            console.log(result);
          } else {
            throw error;
          }
        });
      } else {
        throw error;
      }
    });
  } else {
    throw error;
  }
});
```

**Listing 2: Example of a sequence of continuations**

### 2.2 Promise

In a synchronous paradigm, the sequentiality of the execution flow is trivial. While in an asynchronous paradigm, this control is provided by continuations. The specification[4] defines a **Promise** as an object that is used as a placeholder for the eventual outcome of a deferred (and possibly asynchronous) operation. Promises provide a unified **control over the execution flow** for both paradigms. They expose a `then` method which expects a continuation to continue with the result ; this result being synchronously or asynchronously available.

4. https://people.mozilla.org/~jorendorff/es6-draft.html#sec-promise-objects

Promises force another control over the execution flow. According to the outcome of the operation, they call one function to continue the execution with the result, or another to handle errors. This **conditional execution** is indivisible from the Promise structure. As a result, Promises impose a convention on how to hand back the outcome of the deferred computation, while classic continuations leave this conditional execution to the developer.

```javascript
var promise = my_fn(input)

promise.then(function onSuccess(result) {
  console.log(result);
}, function onErrors(error) {
  throw error;
});
```

**Listing 3: Example of a promise**

Promises are designed to fill the lack of chained composition from continuations. They allow to arrange successions of asynchronous operations as a **chain of continuations**, by opposition to the imbrication of continuations illustrated in listing 2. That is to arrange them, one operation after the other, in the same indentation level.

The listing 4 illustrates this chained composition. The functions `my_fn_2` and `my_fn_3` return promises when they are executed, asynchronously. Because these promises are not available synchronously, the method `then` synchronously returns intermediary Promises. The latter resolve only when the former resolve. This behavior allows to arrange the continuations as a flat chain of calls, instead of an imbrication of continuations.

```javascript
my_fn_1(input)
.then(my_fn_2, onError)
.then(my_fn_3, onError)
.then(console.log, onError);

function onError(error) {
  throw error;
}
```

**Listing 4: A chain of Promises is more concise than an imbrication of continuations**

The Promises syntax is more concise, and also more readable because it is closer to the familiar synchronous paradigm. Indeed, Promises allow to arrange both the synchronous and asynchronous execution flow with the same syntax. It allows to easily arrange the execution flow in parallel or in sequence according to the required causality. Programmers are encouraged to arrange the computation as series of coarse-grained steps to carry over inputs. In this sense, Promises are comparable to the data-flow paradigm.

## 2.3 From Continuation to Promise

As detailed in the previous sections, continuations provide the control over **the sequentiality of the asynchronous execution flow**. Promises improve this control to allow **chained compositions**, and unify the syntax for the synchronous and asynchronous paradigm. This chained composition brings a greater clarity and expressiveness to source codes. At the light of these insights, it makes sense for a developer to switch from continuations to Promises. However, the refactoring of existing code bases might be an operation impossible to carry manually within reasonable time. We want to automatically transform this sequentiality from an imbrication to a chained composition.

To develop further this transformation, we identify two steps. The first is to provide an equivalence between a continuation and a Promise. The second is the composition of this equivalence. Both steps are required to transform imbrications of continuations into chains of Promises.

to be able to compose this equivalence for imbrications of continuations to obtain chains of Promises.

Because Promises bring composition, the first step might seem trivial as it does not imply any imbrication to compose. However, as explained in section 2.2, Promises impose a convention on how to hand back the outcome, while continuations don't.

In the Javascript landscape, there is no dominant convention for handing back outcomes to continuations. In the browser, many conventions coexist. For example, *jQuery*'s `ajax` [5] method expects an object with different continuations for success and errors. *Q* [6] , a popular library to control the asynchronous flow, exposes two methods to define continuations : `then` for successes, and `catch` for errors. These two examples uses different conventions than the Promise specification detailed in section 2.2. Conventions for continuations are very heterogeneous in the browser. On the other hand, *Node.js* API always used the *error-first* convention, encouraging developers to provide libraries using the same convention. In this large ecosystem the *error-first* convention is predominant. We chose to focus on the *error-first* convention because it is likely to represent a large code base.

The Promise specification does not adopt the *error-first* convention. Promises include the conditional execution over the outcome, while the *error-first* convention let developers provide it. To translate one into the other, the compiler would need to identify and extract this conditional execution to prevent duplication with Promises. Such an identification is possible with a static analysis of the control flow. todo references In this paper, however, we focus on the transformation from imbrications to chains, not from one convention to another. For this reason, we propose an alternative specification to Promise following the *error-first* convention.

In the next section we present this specification, called Due. In section 3, we explain the two steps of the transformation from continuations to Dues.

## 2.4 Dues

In this section, we present *Dues*, a simplification of the Promise specification. A Due is an object used as placeholder for the eventual outcome of a deferred operation. Dues are essentially similar to Promises, except for the convention to hand back outcomes. They use the *error-first* convention, like *Node.js*, as illustrated in listing 5. The implementation of Dues and its tests are in appendix A. A more in-depth

5. `http://api.jquery.com/jquery.ajax/`
6. `http://documentup.com/kriskowal/q/`

description of Dues and their creation follows in the next paragraph.

```
var due = my_fn(input)

due.then(function continuation(error, result) {
  if (!error) {
    console.log(result);
  } else {
    throw error;
  }
});
```

**Listing 5: Example of a due**

A due is typically created inside the function which returns it, as illustrated in listing 6. At its creation, the due expects a callback containing the deferred operation. This callback is executed synchronously with the function `settle` as argument to settle the Due, synchronously or asynchronously. Indeed, the operation might be synchronous, or asynchronous. At the end of the operation, its continuation `cont` calls `settle` to settle the Due and save the outcome. The `then` method allows to define a continuation to retrieve this outcome, and continue the execution after the settlement of its Due. If the deferred operation is synchronous, the Due settles during its creation and the `then` method immediately calls this continuation. If the deferred operation is asynchronous, this continuation is called during the Due settlement.

```
function my_fn(input) {
  return new Due(function(settle) {
    deferred(input, function cont(err, result) {
      settle(err, result);
    })
  })
}
```

**Listing 6: Creation of a due**

The composition of Dues is essentially the same than for Promises. This composition is explained in details in section 2.2, and illustrated specifically for Dues in listing 7. Through this chained composition, Dues arrange the execution flow as a sequence of actions to carry on inputs.

```
my_fn_1(input)
.then(screenError(my_fn_2))
.then(screenError(my_fn_3))
.then(screenError(console.log));

function screenError(fn) {
  return function(error, result) {
    if (!error) {
      return fn(result);
    } else {
      throw error;
    }
  };
}
```

**Listing 7: Dues are chained like Promises**

This simplified specification adopts the same convention than *Node.js* continuations to hand back outcomes. Therefore, the equivalence between a continuation and a Due is trivial. Dues are admittedly tailored for this paper, hence, they are not designed to be written by developers, like Promises are. They are an intermediary steps between classical continuation and Promises. We highlight in the section 3 the equivalence between the two latter. But our goal is also to highlight the similitudes between the chained composition, and data-flow paradigms. Indeed, both arrange the computation as series of chained operations to carry, in parallel and in sequence.

## 3. COMPILATION

The previous section defined continuations, the *error-first* convention, and Dues, a simpler Promises specification bringing chained composition. From this definitions, we present the semantic equivalence between a continuation and a Due, and then the composition of this equivalence to develop further the transformation from imbrications into chains.

To illustrate the transformations, we use the function `my_fn` in listing 8. This function is tailored for this transformation, it both expects a callback and returns a Due. The transformation modifies the required signature of the callee. In the source of the transformation, the callee expects a continuation, while in the result of the transformation, the callee returns a Due. The modification of the signature of the callee is addressed in section 4.

```
var D = require('due');

module.exports = function my_fn(arg, continuation) {
  return new D(function(settle) {
    setImmediate(function() {
      var result = arg,
          err = null;

      if (continuation)
        continuation(err, result);

      settle(err, result);
    })
  })
}
```

**Listing 8: `my_fn` expects a callback, and returns a Due**

### 3.1 Simple equivalence

Continuations provide a control over the sequentiality of the asynchronous execution flow, and Dues brings the chained composition. The equivalence between a single continuation and a Due does not involve composition, so the manipulation to transform listing 9 into 10 is trivial. It consist of calling the method `then` of the returned Due, and moving `continuation` to the arguments of this new call.

```
var my_fn = require('./my-fn');

var arg = '1';

my_fn(arg, function continuation(err, res) {
  console.log(res);
});
```

**Listing 9: A simple continuation**

```
var my_fn = require('./my_fn').async;

var arg = '1';

my_fn(arg)
.then(function continuation(err, res) {
  console.log(res);
});
```

**Listing 10: The syntax of a continuation and its Due equivalence are very similar**

By moving the evaluation of `continuation` after the call to `my_fn`, we might alter the execution order, hence the semantic. But the definition of a *Function Expression* is free of side-effects. Indeed, the identifier `continuation` is available only inside itself, for recursion. The manipulation conserves the semantic, it is *sound*.

We focus only on continuations expressed as *Function Expression*. For other types of continuations, the manipulation is either irrelevant, or unsound. When using *Identifiers*, it is impossible to nest continuations, the manipulation is irrelevant. When using *Immediately-Invoked Function Expression*, the manipulation is unsound. Indeed, before the manipulation, the evaluation of this expressionwould occur **before** the call to `my_fn`. While, after the manipulation, it would occur **after** the call to `my_fn`. If this evaluation induce side-effects, the manipulation would prevent them from happening before the call to `my_fn`. The manipulation is *sound* and relevant only when manipulating *FunctionExpression*.

## 3.2 Composition of nested continuations

The equivalence previously presented is incomplete, it leaves sequential operations nested one in the other. To transform an imbrication of continuations into a chain of Dues, we need to assure the composition of this simple equivalence. An example of nested continuations is illustrated in listing 11. The Due equivalence is illustrated in listing 12. The composition of this equivalence requires two additional transformations.

- The nested continuation `cont2` is chained in the same indentation level as the first one, by a second call to the `then` method, line 12. This second call refers to the intermediary Due returned by the first call to the `then` method.
- For this chain to be possible, the Due returned by the nested call to `my_fn` must be linked with this intermediary Due which the second call refers to. The Due from the nested asynchronous function `cont1` is retrieved, line 12, and returned line 10 to be linked internally to the intermediary Due.

The composition of the equivalence leads to some semantical differences between listing 11 and 12. It is unsound in some corner cases. In the next paragraph, we explore the three limits of the composition to ensure soundness.

```
1  var my_fn = require('./my-fn');
2
3  var arg1 = 'A',
4      arg2 = 'B';
5
6  my_fn(arg1, function cont1(err, res) {
7    var shared_identifier = '>> ' + res;
8    my_fn(arg2, function cont2(err, res) {
9      console.log(shared_identifier + ', ' + res);
10   });
11 });
```

**Listing 11: Overlapping continuations definitions**

```
1  var my_fn = require('./my-fn');
2
3  var arg1 = 'A',
4      arg2 = 'B',
5      shared_identifier;
6
7  my_fn(arg1)
8  .then(function cont1(err, res) {
9    shared_identifier = '>> ' + res;
10   return my_fn(arg2);
11 })
12 .then(function cont2(err, res) {
13   console.log(shared_identifier + ', ' + res);
14 });
```

**Listing 12: Sequential continuations definitions using Dues**

### 3.2.1 *Scope disjunction*

In listing 11, because the definitions of `cont1` and `cont2` are overlapping, their scopes [7] are also overlapping. The function `cont1` shares its identifiers with `cont2`. While in listing 12, the definitions of `cont1` and `cont2` are siblings, they have disjoint scopes. The manipulation modifies the semantic, it change the hierarchy of scopes. To keep the semantic intact, we need to assure their disjunction before the manipulation.

A subset of Javascript is statically scoped at the function level, it is possible to infer the scope of identifiers before run time. Javascript exposes two built-in functions that dynamically modify scopes : `eval` and `with`. To avoid dynamical modifications, we consider the subset of Javascript excluding these two built-in functions. This subset is statically scoped at the function level.

The scope of an identifier is limited to its defining function and all its descendants. We define two scopes as disjoints if none use an identifier defined inside the other. They can use common identifiers if they are defined in a common parent scope. Any identifier preventing disjunction is replaceable by an identifier declared in the parent outer scope to be accessible by both the parent and the child.

In listing 11, the identifier `shared_identifier` is accessible both from its defining function `cont1` and the descendant `cont2`. However, in listing 12, the scope of `cont1` and `cont2` are disjoints. For the identifier `shared_identifier` to be accessible by both function, it needs to be registered in a common parent scope. That is the global scope.

### 3.2.2 *Chain of Due*

For the chained composition to be possible, each continuation containing a nested asynchronous function call must return the Due returned by this asynchronous function. The modifications to retrieve and return this Due is sound only if it take place in the common situations. In *Node.js*, continuations are triggered by the engine itself. There is no caller, therefore, there is no context waiting for a result to be returned to. In this situation, a *Return Statement* has purpose only to control the execution flow, and not to hand a result back. However if the continuation was not called directly by Node.js, but wrapped in another continuation, this situation may change.

For the same reason, most asynchronous functions does not return any significant value, because the result of the operation is available only asynchronously. Therefore, most asynchronous functions can modified to return a Due instead.

7. `https://people.mozilla.org/~jorendorff/es6-draft.html#sec-lexical-environments`

However, some asynchronous functions make use of this return value for different purposes. `SetTimeout`, for example, return an identifier pointing to the timer. This identifier is used to cancel the timer before it happened.

### 3.2.3 *Asynchronous function*

In listing 11, if `my_fn` calls `cont2` synchronously, its execution occurs before ②, line ??. While in listing 12, whether the Due returned by `my_fn` settles synchronously or not, the execution of `cont2` occurs after ②, line ??. To keep the semantic intact, only continuations of asynchronous functions can be turned into Dues. We need to assure the asynchronism of `my_fn`.

The deferred computation is asynchronous, and the execution flow is not modified by the manipulation. The function `cont2` is executed after the function `cont1`, and they share the same environment record. So all type of accesses are equivalents : writing or reading. The type of access required by `cont1` and `cont2` is insignificant for this manipulation.

## 4. COMPILER

TODO insert that

We expect developers to have a limited control over the implementation of the callee. In *Node.js*, asynchronous functions are either part of the *Node.js* API like `fs`, or wrapper from libraries like `express`. Instead of modifying the implementation of the callee, we propose to wrap it inside a function which transform its signature, leaving the semantic intact. The `due` library provides such a wrapper : `mock`.

We explain in this section the compilation process. The compiler transform asynchronous call with continuation to make them compatible with due. This process flatten a continuation pyramid into a cascading sequence of call to `then`. There is roughly two steps in this process. The first, described in section 4.1, is to build the chain of continuation from the continuations pyramids. The second, described in section 4.1, is to extract the shared identifiers to move them in a parent scope.

As stated earlier, the compiler doesn't detect rupture points. It expects a list of previously detected rupture points. In the prototype, we spot the rupture point by hand. In section 4.3, we present some thoughts about automation solutions.

### 4.1 Build continuation chains

The first step is to build arrange the rupture points in chain. These chains are branches of trees of rupture points.

A tree of rupture points represent the hierarchy of the rupture points in the source code. To form this tree, there is only one constraint : a child rupture point cannot be separated from its parent by a function. This is because this middle function is not assured to be executed only once, or synchronously. If this middle function is used as an iterator or a listener, there would be multiple child Dues to return, while only one is expected by the parent callback. If this middle function is used as a continuation, the due returned by the child rupture point would net be available synchronously to be returned by the parent callback. For example this middle function might be defined in the parent, but used in a different part of the program.

At the end of this first process, we have multiple trees containing the hierarchy of all the rupture points in the application. Because a function can only return one Due, it is not possible to flatten a tree of rupture points, only a chain. As a callback cannot return more than one Due, it is not possible to build a sequence of Due from a tree. The next step of the compilation is to trim the trees to obtain chains of callbacks transformable into sequence of Due.

Each tree is walked to find rupture point with more than one child. If there is more than one child, we try to find a legitimate child to continue the chain. A legitimate child is a child with at least one child. If there is more than one legitimate child, all are discarded, they all start new chains. The non legitimate child start a new tree to walk the same way.

The result is a list of chains of rupture points. Each chain is assured to be transformable into a sequence of `then` calls. However, as stated earlier, this transformation modifies the scopes organization. To keep the semantic intact, we need to modify the source code in some way that allow the flattening modification to keep the semantic intact.

### 4.2 Identifier extraction

To keep the semantic intact after the flattening of rupture points, no identifier must be shared between two callbacks. Every declaration of shared identifiers is extracted in a parent scope.

We iterate over the rupture point in a chain. If there is any reference to a variable in the children rupture points, then this variable is marked as shared. If the rupture point is not a parent, the descendants scope are not modified by the flattening process.

All shared variables are extracted from their current scope, and placed in the scope at the root of the chain so to be shared by all callbacks in the chain. If there is a conflict with another variable in this root scope, it is necessary to rename one of these variables.

### 4.3 Crowd sourced compilation

Spotting rupture points is equivalent to spotting continuation from other callbacks. A continuation is defined only by its invocation. Spotting a continuation means identifying the function called with the continuation as argument. Function, in Javascript, are first-class citizen, they can take many forms. Statically identifying a function expecting a continuation implies the compiler to have a very deep understanding of the program. This understanding comes from certain static analyses which don't guarantee a good enough result.

If it is not possible to automate the screening process at an individual scale, it might be possible to automate it at a global scale. Most rupture point calls are expected to have distinct names, *e.g.* `fs.readFile`. In future works, we would like to study the possibility to harvest the result of every compilation to build a list of common rupture points. With this list, it would be possible to approximate this automation to ease the compilation interaction.

## 5. EVALUATION

## 6. RELATED WORKS

To our knowledge, our work is the first to present a transformation from continuation to Promise in Javascript. This section relates the various work related with ours. Our work is obviously based on the previous work on Promises and Futures [8], and their specifications in Javascript [8] [9].

Because of its dominant position in the web, Javascript is recently subject to a growing interest in the field of static analysis. We identified currently two teams working on static analysis for Javascript.

In the Department of Computing, Imperial College London, S. Maffeis, P. Gardner and G. Smith realised a large body of work around the static analysis of Javascript. Their work is based around an operational semantic[9] to bring program understanding[12, 4, 3]. Their goal seems to revolve around Security applications of this analysis[11, 10]. In the industry, there already exist some security tools based on static analysis, we can cite for example, the company Shape Security [10]. In a collaboration between the programming language research groups at Aarhus University and Universität Freiburg, P. Thiemann, S. Jensen and A. Møller are working on the static analysis of Javascript.

They presented a tool providing type inference using abstract interpretation[13, 7, 6]. Their goal is to improve the tools available to the Javascript developer[2]. The industry seems to follow the same trend. Facebook released on October 26 2014, a static type checker for Javascript : flow [11]. Another example is the adaptation of the points-to analysis from L. Andersen's thesis work[1] to Javascript[5].

Our compiler aggregates user preferences to transparently improve the service for every user. To our knowledge, we are the first to use principle for software compilation. A good example of similair work is Aviate [12], an android homescreen which automatically organize smartphone applications into existing categories. The use of crowd feedbacks is now a very common practice for many web services. The first example that comes in mind is the search engine suggestions, like Google Autocomplete [13]. Similarly, many services propose a recommendation feature centered on such feedback loops *e.g.* TripAdvisor [14], Yelp [15]. But there exist many other examples making use of this network effect *e.g.* AirBnB [16], Hotel Tonight [17], Uber [18], Lyft [19], Home Joy [20], TaskRabbit [21], handy [22], Shyp [23].

8. `https://promisesaplus.com/`
9. `https://people.mozilla.org/~jorendorff/es6-draft.html#sec-promise-objects`
10. `https://shapesecurity.com/`
11. `http://flowtype.org/`
12. `http://aviate.yahoo.com/`
13. `https://support.google.com/websearch/answer/106230?hl=en`
14. `http://www.tripadvisor.com/`
15. `http://www.yelp.com`
16. `https://www.airbnb.com`
17. `https://www.hoteltonight.com/`
18. `https://www.uber.com/`
19. `https://www.lyft.com/`
20. `https://www.homejoy.com/`
21. `https://www.taskrabbit.com`
22. `https://www.handy.com/`
23. `http://www.shyp.com/`

## 7. CONCLUSION

In this paper, we introduced a compiler to automatically transform an imbrication of continuations into a sequence. Firstly, we defined callbacks and Promises as the base for this work. We then introduced Dues, a new specification similar to Promises, to carry the demonstration of this transformation. We presented the equivalence between a continuation and a Due, and the composition of this equivalence for imbricated continuations. And finally, we presented a compiler to automate this transformation on code bases.

A continuation share its scope with its descendance, *i.e.* the following imbricated continuations. Imbricated continuations can share identifiers. While a due callback can not share identifiers with the following dues. Their scopes are disjoints, still, sequence of dues can share global identifiers and object references. This difference of accessibility implies, after compilation, the segmentation of the asynchronous control flow into indepenent steps. This segmentation is soft : their stacks are independent, but they share the heap.

Dues allow to be arranged in cascade. The result of an asynchronous operation is passed from one Due to the next. A serie of asynchronous operations organized with Dues is very suggestive of a data flow process. It is a chain of operations feeding the next with the result of the previous.

We aim at pushing further this analogy. We want to impose the compiler to bring complete independance to asynchronous operations. We think it is possible to arrange an application as a chain of independent asynchronous operations communicatiing by flow of messages. Such a compiler would be able to transform a monolithic program into a chain of independent asynchronous operations linked by a flow of data. We expect the possibility for new execution models to take advantage of this independence to bring performance scalability. While developers would continue using the monolithic model for its evolution scalability.

## Références

[1] LO Andersen. “Program analysis and specialization for the C programming language”. In : (1994).

[2] E Andreasen, A Feldthaus et SH Jensen. “Improving Tools for JavaScript Programmers”. In : *users-cs.au.dk* ().

[3] P Gardner et G Smith. “JuS : Squeezing the sense out of javascript programs”. In : *JSTools@ ECOOP* (2013).

[4] PA Gardner, S Maffeis et GD Smith. “Towards a program logic for JavaScript”. In : *ACM SIGPLAN Notices* (2012).

[5] D Jang et KM Choe. “Points-to analysis for JavaScript”. In : *Proceedings of the 2009 ACM symposium on Applied ...* (2009).

[6] SH Jensen, PA Jonsson et A Møller. “Remedying the eval that men do”. In : *Proceedings of the 2012 ...* (2012).

[7] SH Jensen, A Møller et P Thiemann. “Type analysis for JavaScript”. In : *Static Analysis* (2009).

[8] B Liskov et L Shrira. *Promises : linguistic support for efficient asynchronous procedure calls in distributed systems*. 1988.

[9] S Maffeis, JC Mitchell et A Taly. "An operational semantics for JavaScript". In : *Programming languages and systems* (2008).

[10] S Maffeis, JC Mitchell et A Taly. "Isolating JavaScript with filters, rewriting, and wrappers". In : *Computer Security–ESORICS 2009* (2009).

[11] S Maffeis et A Taly. "Language-based isolation of untrusted Javascript". In : *. . . Symposium, 2009. CSF'09. 22nd IEEE* (2009).

[12] GD Smith. "Local reasoning about web programs". In : (2011).

[13] P Thiemann. "Towards a type system for analyzing javascript programs". In : *Programming Languages and Systems* (2005).

## APPENDIX

## A. DUE IMPLEMENTATION

We present the implementation of Due in listing 13, with a small set of test cases in listing 14.

```
function Due(callback) {

  var self = this;

  this.id = Math.floor(Math.random() * 100000);

  this.value = undefined;
  this.status = 'pending';
  this.deferral = [];
  this.followers = [];
  this.futures = [];

  this.defer = function(onSettlement) {
    /*  Defer the execution of the settlement handler
     */
    self.deferral.push(onSettlement);
  }

  this.link = function(follower) {
    /*  Link the status of follower to the status of the
          future due
     */
    if (this.status !== 'pending')
      follower.apply(null, this.value)
    else
      this.defer(follower);
  }

  this.resolve = function() {
    /*  ++ Resolve Due ++
     *  If the current due is settled (1)
     *  Execute every settlement handler, and store the (
           future) results (2).
     *  Link every follower (4) added by a returned due (
           returned.9) to every future returned by the
           current resolution (3)
     */

    if (self.status !== 'pending') { // (1)
      self.futures = self.deferral.map(function(deferred)
            { // (2)
        return deferred.apply(null, self.value);
      })

      self.futures.forEach(function(future) {
        if (future && future.isDue) { // (3)
          self.followers.forEach(function(follower) {
            future.link(follower); // (4)
          })
        }
      });
    }
  }

  // Call the deferred computation with the settlement
        function as argument
  callback(function() {
    self.value = arguments;
    self.status = 'settled';
    self.resolve();
  });
}

Due.prototype.isDue = true;

Due.prototype.then = function(onSettlement) {
  this.defer(onSettlement);
  this.resolve();

  var self = this;
  return new Due(function(settle) {
    /*  ++ Returned Due ++
     *  If the current due is settled (1), then the
          future is available.
     *  If this future value is a due, link the returned
          due to it (2)
     *  If this future value is not a due, settle the
        returned due with the future value (3).
     *  If the current due is pending (4), add the
        settlement handler to the followers (5), to be
        deferred to the future dues of the current due.
         (resolve.4)
     */

    if (self.status !== 'pending') { // (1)
      self.futures.forEach(function(future) {
        if (future && future.isDue)
          future.link(settle); // (2)
        else
          settle.apply(null, future); // (3)
      })
    } else { // (4)
      self.followers.push(settle); // (5)
    }
  });
}

// Transform a function expecting callback into a
     function returning due.
Due.mock = function(fn) {
  return function() {
    var _args = Array.prototype.slice.call(arguments),
        _this = this;
    return new Due(function(settle) {
      _args.push(settle);
      fn.apply(_this, _args);
    })
  }
}

module.exports = Due;
```

Listing 13: Implementation of Due

```
var D = require('../src');

describe('Due', function(){
  it('should settle synchronously', function(done){
    var d = new D(function(settle) {
        settle("result");
    })

    d.then(function(result) {
      if (result === "result")
        done();
    })
  })

  it('should settle asynchronously', function(done){
    var d = new D(function(settle) {
        setImmediate(function() {
          settle(null, "result")
        });
    })

    d.then(function(error, result) {
      if (result === "result")
        done();
    })
  })

  it('should cascade synchronously', function(done){
    new D(function(settle) {
      settle(null, "result");
    })
    .then(function(error, result) {
      return new D(function(settle) {
        settle(null, "result2");
      });
    })
    .then(function(error, result) {
      if (result === "result2") {
        done();
      }
    })
  })

  it('should cascade asynchronously', function(done){
    new D(function(settle) {
      setImmediate(function() {
          settle(null, "result")
      });
    })
    .then(function(error, result) {
      return new D(function(settle) {
        setImmediate(function() {
          settle(null, "result2")
        });
      });
    })
    .then(function(error, result) {
      if (result === "result2")
        done();
    })
  })

  it('should cascade synchronously then asynchronously',
       function(done){
    new D(function(settle) {
      settle(null, "result");
    })
    .then(function(error, result) {
      return new D(function(settle) {
        setImmediate(function() {
          settle(null, "result2");
        });
      });
    })
    .then(function(error, result) {
      if (result === "result2")
        done();
    })
  })

  it('should cascade asynchronously then synchronously',
       function(done){
    new D(function(settle) {
      setImmediate(function() {
          settle(null, "result");
      });
    })
    .then(function(error, result) {
      return new D(function(settle) {
        settle(null, "result2");
      });
    })
    .then(function(error, result) {
      if (result === "result2")
        done();
    })
  })

  it('should allow multiple then to same synchronous due'
       , function(done){
    var d = new D(function(settle) {
      settle(null, "result")
    })

    var count = 0;

    var then = function(error, result) {
      if (result === 'result' && ++count === 2)
        done()
    }

    d.then(then);
    d.then(then);
  })

  it('should allow multiple then to same asynchronous due
       ', function(done){
    var d = new D(function(settle) {
      setImmediate(function() {
          settle(null, "result")
      });
    })

    var count = 0;

    var then = function(error, result) {
      if (result === 'result' && ++count === 2)
```

```
        done()
    }

    d.then(then);
    d.then(then);
  })


  it('should expose the mock function', function(){
    if (D.mock === undefined)
      throw 'mock not available'
  })

  // returned value should either be a vow, or a value
})
```

**Listing 14: Tests for the implementation of Due**