From callback hell to promises sequence

Etienne Brodu
etienne.brodu@insa-lyon.fr
IXXI – ENS Lyon
15 parvis René Descartes – BP 7000
69342 Lyon Cedex 07 FRANCE

Stéphane Frénot
stephane.frenot@insa-lyon.fr
IXXI – ENS Lyon
15 parvis René Descartes – BP 7000
69342 Lyon Cedex 07 FRANCE

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
Callbacks and promises are two different tools to arrange the flow of deferred operations, possibly asynchronously. Callbacks implies the inversion of the execution flow. It often result in an intricate imbrication of functions and calls, called the callback hell [1], and largely considered a bad practice. Promises are an alternative on top of callbacks to avoid this imbrication. It allows to replace the overlapping callbacks by a cascading [2] sequence of call. This paper presents an equivalence to transform callback into Promise. To do so, we define an simpler alternative to Promise, called Vow. We present an equivalence to transform callbacks to Vows, and then an equivalence to transform Vows into Promises. We intend to transform the callback hell into a flatten sequence of promises.

2. DEFINITIONS

2.1 Callbacks
A callback is a callable object, *e.g.* a function, passed as an argument to defer its execution, possibly asynchronously. In *Node.js*, the signature of a callback uses the convention *error-first* [3] [4]. The first argument contains an error or `null` if no error occurred ; then follows the result. Listing 1 is an example of callback. The `my_fn` function is defined in listing 2.

```
my_fn(<arg>, function callback(error, result) {
  if (!error) {
    // do something with result ...
  }
})r
```

Listing 1: Example of a callback

2.2 Vows
We present a light alternative to promises called *Vow*. A Vow is identical to a promise, except for two points. *a)* It follows the *error-first* convention, instead of two callbacks, and *b)* it only provides the method `then`. A vow is an object that is used as a placeholder for the eventual results of a deferred (and possibly asynchronous) computation. Any Vow

1. http ://maxogden.github.io/callback-hell/
2. http://stackoverflow.com/questions/758486/how-to-implement-javascript-cascades
3. https://docs.nodejitsu.com/articles/errors/what-are-the-error-conventions
4. http://programmers.stackexchange.com/questions/144089/different-callbacks-for-error-or-error-as-first-argume

object is in one of two mutually exclusive states : settled and pending.

At its creation, the vow expects a callback containing the deferred computation. This callback is called with the function `settle` as argument, to settle the vow. After its creation, the vow exposes a `then` method expecting a callback to continue the execution after the deferred computation.

A vow `v` is settled when the function `settle` is called. A call to `p.then(onSettlement)` immediately call the function `onSettlement`. A vow is pending if it is not settled. A vow is resolved if it is settled or if it has been locked in to match the state of another vow. Attempting to settle a resolved vow has no effect. A vow is unresolved if it is not resolved. An unresolved vow is always in the pending state. A resolved vow may be pending or settled.

The `Vow` object exposes these methods :

`Vow.prototype.then(onSettlement)`
Appends settlement handlers to the vow, and returns a new vow resolving to the return value of the called handler. If the value is a *thenable*, *i.e.* has a method `then`, the returned vow will follow that *thenable*, adopting its eventual state ; otherwise the returned vow will be fulfilled with the value.

We present in section **??** a simple implementation of Vow in Javascript. We only implement `then`, `resolve` and `reject` to keep the implementation concise. The method `catch` is redundant with the method `then`. The implementation for the methods `all` and `race` are out of scope in this paper. However, we present equivalences for both in section **??**.

3. EQUIVALENCES

We present two examples of syntax manipulation to transform callbacks into Vows. The first manipulation transforms a unique callback into a Vow. The second manipulation transforms multiple callbacks with overlapping definitions into a sequence of Vows. The manipulation of the callback definition break the semantic. We present a static lexical analysis to avoid fix the semantic after modification.

The result of the manipulation must use libraries compatible with Vows. So the functions using callback before the manipulation, must returns a Vow after manipulation. `my_fn` in listing 2 is a function both expecting a callback and returning a Vow. There is no known libraries compatible with both callback and Vow, like `my_fn`. We don't focus neither on the detection of these libraries, nor on the detection of their methods. We expect the method using callbacks to be already pointed out, either by hand, or by another automated tool.

```
1  var V = require('./Vow/src');
2
3  module.exports = {
4    sync: function(arg, callback) {
5      return new V(function(settle) {
6        var result = arg,
7            err = null;
8
9        if (callback)
10         callback(err, result);
11
12       settle(err, result);
13     })
14   },
15
16   async: function(arg, callback) {
17     return new V(function(settle) {
18       setImmediate(function() {
19         var result = arg,
20             err = null;
21
22         if (callback)
23           callback(err, result);
24
25         settle(err, result);
26       })
27     })
28   }
29 }
```

Listing 2: Example of two function expecting a callback, and returning a promise, one synchronous the other asynchronous.

3.1 Simple equivalence

A callback is a function passed as argument to defer its execution, like in listing 3. A Vow is on object exposing the method `then` accepting a function passed as argument to defer its execution, like in listing 4. The difference is mainly syntactical. The transformation is immediate, and trivial. As `my_fn` both accept a callback and return a Vow. The manipulation consist of appending a call to the method `then`, referring to the Vow returned by `my_fn`, and moving `callback` to this new call. For *FunctionExpression* like `callback`, this manipulation conserve the semantic. The manipulation is *sound*. For other types of callbacks, *e.g.* a call returning a function, this manipulation is not *sound*. *Soundness* and *Completeness* of the manipulation are addressed in section 3.4.

```
1 var my_fn = require('./my-fn');
2
3 var arg = '1';
4
5 my_fn(arg, function callback(err, res) {
6   console.log(res);
7 });
```

Listing 3: A simple callback

```
1 var my_fn = require('./my_fn').async;
2
3 var arg = '1';
4
5 my_fn(arg)
6 .then(function callback(err, res) {
7   console.log(res);
8 });
```

Listing 4: A simple Vow is very similar to a simple callback

3.2 Overlapping callbacks

One of the intention using Vows, is to flatten the overlapping definitions of callbacks. In listing 5, the two callbacks definition, `cb1` line 6 and `cb2` line 10, are overlapping. While, in listing 6, they are not overlapping, they are defined sequentially, one after the other. It is the expected result using Vows, and Promises. The transformation between 5 and 6 is the same than in the previous example, only two more transformation are required. To link the sequence of execution, the `cb1` must retrieves the Vow returned by the second call to `my_fn`, line **??**, and return it, line **??**.

However, there is two semantical differences between listing 5 and 6. Moving the definition of `cb2` is not *sound*.

In listing 5, because the definitions of `cb1` and `cb2` are overlapping, their environment record, commonly called scope, are also overlapping. The function `cb1` shares its scope with `cb2`. While in listing **??**, the definitions of `cb1` and `cb2` are siblings, so `cb1` and `cb2` have their environment records disjoints. To keep the semantic intact, we need to analyze the environment records to assure their disjunction before the manipulation. We address this issue in section 3.3.

In listing 5, if `my_fn` calls `cb2` synchronously, its execution occurs before ②, line 12. While in listing 6, whether the Vow returned by `my_fn` settle synchronously or not, the execution of `cb2` occurs after ②, line **??** To keep the semantic intact, we need to assure the asynchronism of `my_fn`. To address this issue, we impose the manipulation to be applied only on asynchronous functions.

```
1  var my_fn = require('./my-fn');
2
3  var arg1 = 'a 1',
4      arg2 = 'a 2';
5
6  my_fn(arg1, function cb1(err, res) {
7    // ① ...
8    console.log(res);
9
10   my_fn(arg2, function cb2(err, res) {
11     console.log(res);
12   });
13   // ② ...
14 });
```

Listing 5: Overlapping callbacks definitions

```
1  var my_fn = require('./my-fn');
2
3  var arg1 = 'b 1',
4      arg2 = 'b 2';
5
6  my_fn(arg1)
7  .then(function cb1(err, res) {
8    // ① ...
9    console.log(res);
10
11   var v = my_fn(arg2);
12   // ② ...
13   return v; // return the promise from my_fn
14 })
15 .then(function cb2(err, res) {
16   console.log(res);
17 });
```

Listing 6: Sequential callbacks definitions using Vows

3.3 Assure environment record disjunction

A subset of Javascript is lexically scoped at the function level. A function defines a Lexical Environment [5]. A lexical environment consists of an environment record and a possibly null reference to an outer environment. An Environment Record records the identifier bindings that are created within the scope of its associated Lexical Environment.

A Lexical Environment is static, it is immutable during run time. So it is possible to infer the identifiers and their scopes before run time. The scope of an identifier is limited to the defining function and its children. Javascript exposes two built-in functions that dynamically modify lexical environment : `eval` and `with`. To assure the disjunction of two Environment records, we exclude programs using these functions, to avoid dynamical modifications.

In listing 5, the environment records of `cb1` and `cb2` are overlapping. An identifier `example_identifier` declared in place of ① line 8, would be accessible from `cb2`. However, in listing 6, the Environment Records of `cb1` and `cb2` are siblings. The identifier `example_identifier` is no longer accessible from `cb2`. We want to assure the disjunction between a parent record environment and its child to move the latter while keeping the semantic. Two environment records are disjoints if they don't share any bindings. Two environment records are joints if they share at least one binding. A shared binding is replaceable by a binding declared in the parent outer environment record to be accessible by both the parent and the child. In listings 5 and 6 this outer environment is the global environment records. The execution flow is not modified by the translation into vows. So all type of accesses, writing or reading, to a binding are equivalents.

3.4 Soundness and Completeness

TODO TODO prove soundness and completeness with the following The call to `my_fn` is a *CallExpression* [6]. The arguments of a CallExpression are only AssignementExpression.

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

6. `https://people.mozilla.org/~jorendorff/es6-draft.html#sec-expression-rules`