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. DEFINITIONS

1.1 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* [1] [2]. 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

1.2 Vows

We present a simpler alternative to promises in Javascript called *Vow*. A vow is an object that is used as a placeholder for the eventual results of a deferred (and possibly asynchronous) computation. Any Vow object is in one of two mutually exclusive states : settled or pending. A Vow is identical to a promise, except for two points. *a)* It follows the *error-first* convention, like *Node.js* callbacks, and *b)* it provides only one method `then` to continue the execution after the deferred computation.

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. If `v` is settled, a call to `v.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 only exposes the `then` method. **`Vow.prototype.then(onSettlement)`** Appends settlement handlers to the vow, and returns a new

1. `https://docs.nodejitsu.com/articles/errors/what-are-the-error-conventions`
2. `http://programmers.stackexchange.com/questions/144089/different-callbacks-for-error-or-error-as-first-argumen`

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.

2. EQUIVALENCES

We present two examples of source code manipulation to transform callbacks into Vows. The first manipulation is the simplest one. It transforms a unique callback into a Vow. The second manipulation is the composition of the first manipulation. It transforms multiple callbacks with overlapping definitions into a sequence of Vows. This second manipulation requires to move a callback definition. This modifies the semantic. We finally present a static lexical analysis to refactor the source code before the manipulation to avoid the semantic modification.

The main advantage for developers using Vows, is to flatten the overlapping callbacks into a more readable sequence of functions. The pyramid of callbacks only occurs when these callbacks are defined by *FunctionExpressions*[3]. If the callback is not declared *in situ*, it will most likely not lead to a pyramid of function declarations and calls. This equivalence would not improve readability. Moreover, it would require heavier manipulation of the source code, as explained in section **??**.

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 libraries compatible with both callback and Vow, like `my_fn`. We don't focus neither on the replacement of these libraries, nor on the detection of their methods in the source code. We expect the method using callbacks to be already pointed out, either by a developer, or by another automated tool.

```
var V = require('./Vow/src');

module.exports = {
  sync: function(arg, callback) {
    return new V(function(settle) {
      var result = arg,
          err = null;

      if (callback)
        callback(err, result);

      settle(err, result);
    })
  },

  async: function(arg, callback) {
    return new V(function(settle) {
      setImmediate(function() {
        var result = arg,
            err = null;

        if (callback)
          callback(err, result);

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

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

2.1 Simple equivalence

As explained in section 1.1, a callback is a function passed as argument to defer its execution, like in listing 3. As explained in section **??**, a Vow is an object to defer a computation, and exposes a method then to continue the execution after the deffered computation, like in listing 4. The difference between the listings 3 and 4, is mainly syntactical. The transformation is immediate, and trivial. As illustrated in listing 2, `my_fn` both accepts a callback and returns a Vow. The manipulation consist of calling the method `then` of the Vow returned by `my_fn`, and moving `callback` to this new call For some types of callbacks, *e.g.* a function call returning a function, this manipulation is not *sound* because it modifies the execution order. Before the manipulation, the callback evaluation would occur **before** the call to `my_fn`. After the manipulation, the callback evaluation would occur **after** the call to `my_fn`. For *FunctionExpression* like `callback`, this manipulation conserves the semantic because of the *hoisting* features of Javascript. Inside a function, Javascript process variable and function declarations before executing any code. Declaring an identifier anywhare in a function is equivalent to declaring it at the top. The identifier `callback`, is declared before the call to `my_fn` in both listings. This behavior is called *hoisting*. It makes this manipulation *sound*. It is the reason why it is *sound* only when manipulating *FunctionExpression*, as explained in the beginning of section **??**.

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

var arg = '1';

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

Listing 3: A simple callback

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

var arg = '1';

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

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

2.2 Overlapping callbacks

The previous manipulation allows the modification of only one callback. To transform an overlapping pyramid of callback into a sequence of Vows, we need to assure the composition of this simple transformation. In listing 5, the two callbacks definition, `cb1` line 6 and `cb2` line 11, are overlapping. While, in listing 6, they are not overlapping, they are defined sequentially, one after the other. It is the expected result for the composition of Vows. The transformation between 5 and 6 is the same than in the previous example, only

3. http://www.ecma-international.org/ecma-262/5.1/#sec-11.2.5

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 13, and return it, line 15.

The composition of the simpler manipulation leads to two semantical differences between listing 5 and 6. Moving the definition of `cb2` is not *sound*.

- In listing 5, if `my_fn` calls `cb2` synchronously, its execution occurs before 2, line 14. While in listing 6, whether the Vow returned by `my_fn` settle synchronously or not, the execution of `cb2` occurs after 2, line 14 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.
- 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 identifiers with `cb2`. While in listing 6, the definitions of `cb1` and `cb2` are siblings, so `cb1` and `cb2` have their environment records disjoints. If `cb2` uses identifiers defined in `cb1`, the manipulation makes them inaccessible. 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 2.3.

```
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    var shared_identifier = res + '>>';
9    console.log(res);
10
11   my_fn(arg2, function cb2(err, res) {
12     console.log(shared_identifier + res);
13   });
14   // 2 ...
15 });
```

Listing 5: Overlapping callbacks definitions

```
1  var my_fn = require('./my-fn');
2
3  var arg1 = 'b 1',
4      arg2 = 'b 2',
5      shared_identifier;
6
7  my_fn(arg1)
8  .then(function cb1(err, res) {
9    // 1 ...
10   shared_identifier = res + '>>';
11   console.log(res);
12
13   var v = my_fn(arg2);
14   // 2 ...
15   return v; // return the promise from my_fn
16 })
17 .then(function cb2(err, res) {
18   console.log(shared_identifier + res);
19 });
```

Listing 6: Sequential callbacks definitions using Vows

2.3 Assure environment record disjunction

We consider a subset of Javascript. This subset is lexically scoped at the function level. A function defines a Lexical Environment [4]. 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 avoid dynamical modifications by excluding programs using these functions.

In listing 5, the environment records of `cb1` and `cb2` are overlapping. The identifier `shared_identifier` declared line 8, is accessible from `cb2`. However, in listing 6, the Environment Records of `cb1` and `cb2` are siblings. The identifiers declared in `cb1` are 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 identifiers. Two environment records are joints if they share at least one identifier. A shared identifier is replaceable by a identifier declared in the parent outer environment record to be accessible by both the parent and the child. The identifier `shared_identifier` is moved to the outer environment, shared by both `cb1` and `cb2`. In listings 5 and 6 this outer environment is the global environment records.

As assured in section **??**, the deferred computation is asynchronous. And the execution flow is not modified by the manipulation. The function `cb2` is executed after the function `cb2`, and they share the same environment record. So all type of accesses are equivalents : writing or reading. The type of access required by `cb1` and `cb2` is insignificant for this manipulation.

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