Automatic transformation from callback imbrication to sequence

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

Fabien Cellier, Frédéric Oblé
firstname.lastname@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 with the introduction in Netscape 2.0 of Javascript, a web scripting language.

Javascript was originally designed as a script language for the manipulation of a graphical environment : the Document Object Model (DOM [1]). Functions are first class-citizen. It allows to manipulate functions like any object, and link them to react to asynchronous events, *e.g.* user inputs, remote requests. These asynchronously triggered functions are named callback, 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 web applications.

Callbacks were 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 to control large asynchronous execution flow. Eventually, developers enhanced callbacks to meet their needs with the concept of Promise[1].

Promises bring the missing features to control the asynchronous execution flow. 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. Despite its benefits, the concept of Promise is not yet widely acknowledged, developers usually wrote their own version of Promise. There is such a disparity between the needs for and the acknowledgment of Promises, there is almost 40 different *known* implementations [3].

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

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

callbacks, the *callback hell*, to a sequence of Promise operations, We focus on the consequences of this transformation on the source of a web application.

In section 2 we define callback and Promise. We then introduce in section 3 a new specification, called Due, essentially similar to Promise. The interface exposed by Dues is close to the interface of Callbacks. While Promises use a different convention. In section 4, we explain the transformation from imbrication to sequence. In section 5, we present an implementation of this transformation. We present related works in section 6, and finally conclude in section 7.

2. DEFINITIONS

2.1 Callbacks

A callback is a function passed as an argument to another function to defer its execution. It allows to process arguments not available in the current context. We distinguish three kinds of callbacks.

- An **Iterator** is a function called for each item in a set. An iterator is often called synchronously.
- A **Listener** is a function called for each message in a stream. A listener is called asynchronously, for each new available message.
- A **Continuation** is a function called asynchronously once a unique result is available. Callbacks are often mistaken for continuations. We only focus on continuation in this paper, because promises - or dues - can replace only continuations.

When continuation are used, the convention on how to handle the result must be common for both the caller and the caller. In *Node.js*, the signature of a continuation uses the convention *error-first*[4] [5]. The first argument contains an error or `null` if no error occurred ; then follows the result. Listing 1 is an example of continuation. The `my_fn` function is defined in listing 6. Another convention is to provide two distinct functions to handle results and errors.

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

Listing 1: Example of a continuation

The callback hell occurs when many asynchronous calls are cascaded, like in listing 2. Each caller must wait for the result before calling the next function. Promise allows to arrange a sequence of deferred execution in a more readable way.

```
my_fn_1(arg, function continuation(error, result) {
  if (!error) {
    // do something with result to get arg
    my_fn_2(arg, function continuation(error, result) {
      if (!error) {
        // do something with result to get arg
        my_fn_3(arg, function continuation(error, result)
          {
          if (!error) {
            // do something with result ...
          } else {
            // do something with error ...
          }
        });
      } else {
        // do something with error ...
      }
    });
  } else {
    // do something with error ...
  }
});
```

Listing 2: Example of a cascade of continuations

2.2 Promise

This section is based on the Promises section of the specification in ECMAScript 6 Harmony[6] and the Promises page on the Mozilla Developer Network[7]. The specification defines a promise as *an object that is used as a placeholder for the eventual results of a deferred (and possibly asynchronous) computation.* (ECMAScript 6 Harmony Specification, section 25.4 Promise Objects)

A promise is an object returned by a function to represent its result. Because it is possibly unavailable synchronously, it still requires a callback to defer the execution when the result is made available. A promise also requires another callback to defer the execution in case of errors. This two callbacks are passed to the method `then` of the promise, like illustrated in listing 5.

```
var promise = my_fn(<arg>)

promise.then(function onSuccess(result) {
  // do something with result ...
}, function onErrors(reason) {
  // do something with the reason of the error ...
});
```

Listing 3: Example of a promise

A sequence of promises is arranged as a cascade of call, by opposition to the imbrication of callbacks. To allow cascading, the method `then` returns a promise which resolves when the promise returned by its callbacks resolve. This is illustrated in listing 4. The two first `onSuccess` callbacks call `my_fn_2` and `my_fn_3`, return the promises `p2` and `p4`. The promises `p3` and `p5`, returned by the `then` calls to `p1` and `p3`, resolve respectively when `p2` and `p4` resolve. This behavior allow to arrange the callback in a flat cascade of calls, instead of an imbrication of calls and callback.

```
var p1 = my_fn_1(arg)

var p3 = p1.then(function onSuccess(result) {
  // do something with result to get arg
  var p2 = my_fn_2(arg);
  return p2;
}, function onErrors(reason) {
  // do something with the reason of the error ...
})

var p5 = p2.then(function onSuccess(result) {
  // do something with result to get arg
  var p4 = my_fn_3(arg);
```

4. `https://docs.nodejitsu.com/articles/errors/what-are-the-error-conventions`
5. `http://programmers.stackexchange.com/questions/144089/different-callbacks-for-error-or-error-as-first-argument`
6. `https://people.mozilla.org/~jorendorff/es6-draft.html#sec-promise-objects`
7. `https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise`

```
  return p4;
}, function onErrors(reason) {
  // do something with the reason of the error ...
})

p5.then(function onSuccess(result) {
  // do something with result ...
}, function onErrors(reason) {
  // do something with the reason of the error ...
});
```

Listing 4: Example of a sequence of promise

The convention for callbacks used by Promises differs from the convention used in *Node.js*. The former uses two callbacks, while the latter uses the only one, with the *error-first* convention. This difference is mainly syntactic, and is irrelevant to the transformation from imbrication to sequence. To remove the syntactic difference from the transformation presented in this paper, we present a new specification, Due, in section 3. Dues are essentially similar to Promises, and use the *error-first* convention, predominant in *Node.js*.

3. DUES

We present a simpler alternative to promises in Javascript called *Due*. A due is an object that is used as a placeholder for the eventual results of a deferred computation. The method `then` of a due expects only one callback, following the convention *error-first*, like in *Node.js*.While the method `then` of a promise expects two callbacks, `onSuccess` and `onErrors`.

Any Due object is in one of two mutually exclusive states : settled or pending. At its creation, the due expects a callback containing the deferred computation. This callback is called with the function `settle` as argument, to settle the due. After its creation, the due exposes a `then` method expecting a callback to continue the execution after the deferred computation. Similarly to Promise, to allow cascading, the method `then` returns a Due which resolve when the due returned by its callbacks resolve.

```
var due = my_fn(arg)

due.then(function callback(error, result) {
  if (!error) {
    // do something with result ...
  } else {
    // do something with error ...
  }
});
```

Listing 5: Example of a due

In listing 5, `due` is settled when the function `settle` is called. If `due` is settled, a call to `due.then(onSettlement)` immediately call the function `onSettlement`. A due is pending if it is not settled. A due is resolved if it is settled or if it has been locked in to match the state of another due. Attempting to settle a resolved due has no effect. A resolved due may be pending or settled, while an unresolved due is always in the pending state. The `Due` object only exposes the `then` method. We present in section **??** a simple implementation of Due in Javascript.

4. EQUIVALENCES

We present two examples of source code manipulation to transform continuation into Dues. The first manipulation is the simplest one. It transforms a unique continuation into a Due. The second manipulation is the composition of the first manipulation. It transforms multiple continuations with overlapping definitions into a sequence of Dues. This second manipulation requires to move the continuation definitions. This modifies the semantic. We finally present a static lexical analysis to modify the source code before the manipulation to avoid the semantic modification.

The main advantage for developers using Dues, is to flatten the overlapping continuations into a more readable sequence of functions. The pyramid of continuations only occurs when they are defined by *FunctionExpressions*[8]. When the continuation is not declared *in situ*, it avoids the imbrication of function declarations and calls. So, we focus only on the modification of continuation declared *in situ*.

This transformation modifies the syntax of the call. The function called needs to be modified to fit this new syntax, it must return a Due. `my_fn` in listing 6 is a function both expecting a callback and returning a Due. There is no libraries compatible with both callback and Due, like `my_fn`. However, the Due library provide a function `mock` to transform a function expecting continuation into a function returning a Due. 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 continuation to be already screened out from other callbacks, either by a developer, or by another automated tool. We address this problem in section .

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

4.1 Simple equivalence

As explained in section 2.1, a continuation is a function passed as argument to defer its execution, like in listing 7. As

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

explained in section **??**, a Due is an object to defer a computation, and exposes a method `then` to continue the execution after the deferred computation, like in listing 8. The difference between the listings 7 and 8, is mainly syntactical. The transformation is immediate, and trivial. As illustrated in listing 6, `my_fn` both accepts a callback and returns a Due. The manipulation consist of calling the method `then` of the Due returned by `my_fn`, and moving `continuation` to the arguments of this new call. In Javascript, when entering a scope, declaration of variables and functions are processed before any execution. Declaring an identifier anywhere in a scope is equivalent to declaring it at the top. The identifier `continuation`, is declared before the call to `my_fn` in both listings 7 and 8. This behavior is called *hoisting*. It makes this manipulation *sound*. The manipulation conserves the semantic for *FunctionExpression* like `continuation`.

For other types of continuations, *e.g.* an expression returning a function, this manipulation modifies the execution order. Before the manipulation, the expression evaluation would occur **before** the call to `my_fn`. While, after the manipulation, the expression evaluation would occur **after** the call to `my_fn`. If the expression evaluation produces expected side-effects, the manipulation would prevent them from happening before the call to `my_fn`. The manipulation is *sound* only when manipulating *FunctionExpression*.

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

var arg = '1';

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

Listing 7: 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 8: A simple Due is very similar to a simple continuation

4.2 Overlapping continuations

The previous manipulation allows the modification of only one continuation. To transform an overlapping pyramid of continuation into a sequence of Dues, we need to assure the composition of this simple transformation. An example of overlapping pyramid of continuation is illustrated in listing 9. The expected composition manipulation is illustrated in listing 10. In listing 9, the two continuations definition, `ct1` line **??** and `ct2` line 11, are overlapping. While, in listing 10, they are not overlapping, they are defined sequentially, one after the other. The transformation between 9 and 10 is similar to the previous example, only two more transformations are required. To link the sequence of execution, the `cb1` must *a)* retrieves the Due returned by the second call to `my_fn`, line **??**, and *b)* return it, line 15.

The composition of the simpler manipulation leads to two semantical differences between listing 9 and 10. Moving the definition of `ct2` is not *sound*.

— In listing 9, if `my_fn` calls `ct2` synchronously, its execution occurs before ②, line 14. While in listing 10, whether the Due returned by `my_fn` settles synchronously or not, the execution of `ct2` occurs after ②, line 14 To keep the semantic intact, the manipulation is practicable only on asynchronous functions. We need to assure the asynchronism of `my_fn`.

— In listing 9, because the definitions of `ct1` and `ct2` are overlapping, their environment record, commonly called scope, are also overlapping. The function `ct1` shares its identifiers with `ct2`. While in listing 10, the definitions of `ct1` and `ct2` are siblings, so `ct1` and `ct2` have their environment records disjoints. If `ct2` uses identifiers defined in `ct1`, 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 4.3.

```
1  var my_fn = require('./my-fn');
2
3  var arg1 = 'a 1',
4      arg2 = 'a 2';
5
6  my_fn(arg1, function ct1(err, res) {
7    // ① ...
8    var shared_identifier = res + '>>';
9    console.log(res);
10
11   my_fn(arg2, function ct2(err, res) {
12     console.log(shared_identifier + res);
13   });
14   // ② ...
15 });
```

Listing 9: Overlapping continuations 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 ct1(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 ct2(err, res) {
18   console.log(shared_identifier + res);
19 });
```

Listing 10: Sequential continuations definitions using Dues

4.3 Assure environment record disjunction

In Javascript, a function defines a Lexical Environment [9]. 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.

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

Javascript exposes two built-in functions that dynamically modify lexical environment : `eval` and `with`. We consider a subset of Javascript, excluding `eval` and `with`.

This subset is lexically scoped at the function level. 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. To assure the disjunction of two Environment records, we avoid dynamical modifications by excluding programs using these functions.

In listing 9, the environment records of `ct1` and `ct2` are overlapping. The identifier `shared_identifier` declared line 8, is accessible from `ct2`. However, in listing **??**, the Environment Records of `ct1` and `ct2` are siblings. The identifiers declared in `ct1` are no longer accessible from `ct2`. To move the child Environment Records out of its parent while keeping the semantic, it needs to be disjoint from its parent. 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 an 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 `ct1` and `ct2`. In listings 9 and **??** 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 `ct2` is executed after the function `ct2`, and they share the same environment record. So all type of accesses are equivalents : writing or reading. The type of access required by `ct1` and `ct2` is insignificant for this manipulation.

5. COMPILER

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 5.1, is to build the chain of continuation from the continuations pyramids. The second, described in section 5.1, 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 5.3, we present some thoughts about automation solutions.

5.1 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.

5.2 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.

5.3 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.

6. RELATED WORKS

7. CONCLUSION

In this paper, we introduced a compiler to automatically transform an imbrication of continuations into a sequence. Firstly, we defined callbacks and Promise as the base for this work. We then introduced Due, a new specification similar to Promise, 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 actual code bases.

A continuation share its scope with its descendance, *i.e.* the following imbricated continuations. While A callback due can not share its own identifiers with its descendance, *i.e.* the following dues. Their scope are disjoints. However, it can share global identifiers, and object references. This difference of accessibility imposes the compiler to segment the asynchronous control flow. This segmentation is soft : the stack is independent, but the heap is shared.

The callback of a Due returns another due, for the asynchronous operation completions to trigger the next. The result of an asynchronous operation is passed to the next through the Due - or Promise - structure. A serie of asynchronous operations operated by Dues - or Promises - 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. So that the only communication is of their results along the flow. 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 continue using the monolithic model for its evolution scalability.

Références

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