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, 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 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, like Promises. There is such a disparity between the needs for and the acknowledgment of Promises, that there is almost 40 different *known* 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 hands, leave this conditional control to the developer. This paper focuses on the transformation of the control of the asynchronous execution flow from callbacks to Promises. We introduce a new specification, called Dues, essentially similar to Promises, but leaving unchanged the conditional control. This approach enables us to compile legacy Javascript code from code repository and brings a first automated step toward full Promises integration. This simple and pragmatic compiler has been tested over n Github repository, k with success.

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

2. DEFINITIONS

2.1 Callbacks

A callback is a function passed a 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 placeholder for a unique outcome. Iterators and Listeners are invoked multiple times resulting in multiple outcomes. Only continuations are equivalent to Promises, or Dues. So, we focus on continuations in this paper.

Callbacks are often mistaken for continuations ; callbacks are not inherently asynchronous, while continuations are. A continuation allows to control the sequentiality of asynchronous operations in the flow of execution. It 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 process the result as soon as possible and continue the execution ; hence the name continuation. The continuation approach is the functional way of addressing asynchronous call without external synchronization mechanism such as IPC.

When using continuation, the convention on how to handle the result must be common for both the callee and the caller. In *Node.js*, the signature of a continuation uses the *error-first*[4] [5] convention. The first argument contains an error or `null` if no error occurred ; then follows the result. Listing 1 is a pattern of continuation.

```
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 produce an imbrication of calls and function definitions, like in listing 2. Promise allows to arrange such a sequence of asynchronous operations in a more 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 Promises

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 outcome 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 continuation to defer the execution when the result is made available. A promise also requires another continuation to defer the execution in case of errors. These two continuations are passed to the `then` method of the promise, like illustrated in listing 3.

```
var promise = my_fn(input)
```

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`

```

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

Listing 3: Example of a promise

Promises are specified as to arrange successions of asynchronous operations as a chain of continuations, by opposition to the imbrication of continuations illustrated in listing 2. To allow cascading, the method `then` returns a promise which resolves when the promise returned by its continuation resolves. This is illustrated in listing 4. 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` returns intermediary Promises. The latter resolve only when the former resolve. This behavior allows to arrange the continuations in a flat chain of calls, instead of an imbrication of calls and continuations.

```
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: Example of a chain of promise

2.3 Analysis

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, while this sequentiality needs to be explicit. Promises and continuations provide this control over **the sequentiality of the asynchronous execution flow**. It allows to explicitly arrange the execution of asynchronous operations one after the other, and declare a relation of causality between two operations.

As explained in section 2.1, continuations are invoked to hand back the result and continue the execution at the end of an asynchronous operation. To arrange a sequence of asynchronous operations with continuations, they are nested one in the continuation of the previous, as illustrated in listing 2. When the continuation is a function declared *in situ*, each asynchronous operation adds a nesting level. Because of this nesting, we say that continuations lack the **chained composition** of multiple asynchronous operations.

As illustrated in listing 4, Promises provides this chained composition. As detailed in section 2.2, the `then` method synchronously returns a Promise linked with the Promise asynchronously returned by its continuation. This link allow to compose **chains** of asynchronous operations. That is to arrange them, one operation after the other, in the same indentation level. The Promises syntax is more readable, because it is closer to the familiar synchronous paradigm.

However, Promises include another control over the execution flow. They call a different function according to the outcome of the asynchronous operation, one to continue the execution with the result, or the other to handle errors. This **conditional execution** is indivisible from the Promise structure. On the other hand, classic continuations leave this conditional execution to the developer. As a result of this difference, Promises and continuations use two different conventions to handle errors and results. The two conventions are illustrated in listings 1 and 3.

We focus on the transformation of **the sequentiality of the execution flow**, but not on the extraction of the conditional execution. We introduce in section 3 a new specification, Dues. They bring the same chained composition than Promises, while leaving the conditional execution to the developer, like continuations.

3. DUES

We present an alternative to Promises called *Due*. Like Promises, a Due is an object that is used as a placeholder for the eventual outcome of a deferred computation. Unlike Promises, and like continuations, Dues leave to the developer the control of the conditional execution over the result. While a promise expects two continuations, `onSuccess` and `onErrors`, the method `then` of a due expects only one continuation, following the convention *error-first*.

A 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 synchronously with the function `settle` as argument. The latter is invoked, potentially asynchronously, to settle the due. Dues expose a `then` method expecting a continuation to continue the execution after its settlement. To allow chained composition, the method `then` returns a Due linked with the due returned by its continuations. The definition of `my_fn` function, in listing 7 illustrate the creation of two Dues, with synchronous and asynchronous deferred computation.

```
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

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 linked with 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 appendix A a simple implementation of Due in Javascript.

```
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 6: Dues are chained like Promises

4. EQUIVALENCES

In the previous section, we present the difference between continuation and Dues Both allow control over the sequentiality of the execution flow. When using only continuation, sequence of asynchronous operations are nested, one in the continuation of the next. On the other hand, Dues allow the linear composition of continuations.

Based on this difference, 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 nested continuations into a linear sequence of Dues. This second manipulation requires to move the continuation definitions, which 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 to use Dues, is to flatten the overlapping continuations into a more readable, linear sequence. The nesting 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. We focus only on the modification of continuation declared *in situ*. Moreover, the transformation is *sound* only when manipulating *FunctionExpressions*, as explained in section **??**.

The transformations presented modifies the syntax of the asynchronous call. The asynchronous function needs to be modified to return a Due, instead of expecting a continuation. For the demonstrations, we use the function `my_fn` in listing 7. It both expects a callback and returns a Due. There is no libraries compatible with both callback and Dues, 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 continuations to be already screened out from other callbacks, either by a developer, or by another automated tool. We address this problem in section 5.3.

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

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

      if (continuation)
        continuation(err, result);

      settle(err, result);
    })
  },

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

        if (continuation)
          continuation(err, result);

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

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

Listing 7: Example of two function expecting a callback, and returning a due, one synchronous the other asynchronous.

4.1 Simple equivalence

As explained in section **??**, a continuation is a function passed as argument to defer its execution, like in listing 8. As explained in section 3, 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 9.

Because the difference between continuations and dues is the composition, the difference between the listings 8 and 9 is mainly syntactical. The transformation is immediate, and trivial. 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 8 and 9. This behavior is called *hoisting*. The manipulation is *sound* because it conserves the semantic.

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 8: 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 9: A simple Due is very similar to a simple continuation

4.2 Composition of nested continuations

The previous manipulation allows the modification of only one continuation. To transform a nested pyramid of continuations into a sequence of Dues, we need to assure the composition of this simple transformation. An example of nested pyramid of continuation is illustrated in listing 10. The expected result for the composition is illustrated in listing 11.

In listing 10, the two continuations definition, `ct1` line 6 and `ct2` line 11, are overlapping. While, in listing 11, they are not overlapping, they are defined sequentially, one after the other. The transformation between 10 and 11 is similar to the previous transformation, only two more transformations are required. For the linear composition, `ct1` must *a)* retrieves the Due returned by the second call to `my_fn`, line 13, and *b)* returns it, line 15.

The composition of the simpler manipulation leads to two semantical differences between listing 10 and 11. Moving the definition of `ct2` is not *sound*.

— In listing 10, if `my_fn` calls `ct2` synchronously, its execution occurs before 2, line 14. While in listing 11, 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, only continuations of asynchronous functions can be turned into Dues. We need to assure the asynchronism of `my_fn`.

— In listing 10, 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 11, the definitions of `ct1` and `ct2` are siblings, so `ct1` and `ct2` have disjoint scopes. If `ct2` uses identifiers defined in `ct1`, the manipulation makes them inaccessible. To keep the semantic intact, we need to analyze their scope 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 10: 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 11: Sequential continuations definitions using Dues

4.3 Assure environment record disjunction

In Javascript, a function defines a *Lexical Environment*[9]. A *Lexical Environment* defines the scope of a function. It consists of an *Environment Record* and a - potentially null - reference to an outer *Lexical Environment*. An *Environment Record* records the identifier bindings that are created within the scope of its associated *Lexical Environment*. Javascript exposes two built-in functions that dynamically modify *Lexical Environment* : `eval` and `with`.

To avoid dynamical modifications of Lexical Environment, we consider a subset of Javascript, excluding `eval` and `with`. This subset is statically - or lexically - scoped at the function level. A *Lexical Environment* is static, it is immutable during run time. 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.

In listing 10, the scopes of `ct1` and `ct2` are overlapping. The *Lexical Environment* of `ct1` is the outer environment of the *Lexical Environment* of `ct2`. 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 10 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

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

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

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 [10] [11].

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 [12]. 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

10. `https://promisesaplus.com/`
11. `https://people.mozilla.org/~jorendorff/es6-draft.html#sec-promise-objects`
12. `https://shapesecurity.com/`

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 [13]. 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 [14], 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 [15]. Similarly, many services propose a recommendation feature centered on such feedback loops *e.g.* TripAdvisor [16], Yelp [17]. But there exist many other examples making use of this network effect *e.g.* AirBnB [18], Hotel Tonight [19], Uber [20], Lyft [21], Home Joy [22], TaskRabbit [23], handy [24], Shyp [25].

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.

13. `http://flowtype.org/`
14. `http://aviate.yahoo.com/`
15. `https://support.google.com/websearch/answer/106230?hl=en`
16. `http://www.tripadvisor.com/`
17. `http://www.yelp.com`
18. `https://www.airbnb.com`
19. `https://www.hoteltonight.com/`
20. `https://www.uber.com/`
21. `https://www.lyft.com/`
22. `https://www.homejoy.com/`
23. `https://www.taskrabbit.com`
24. `https://www.handy.com/`
25. `http://www.shyp.com/`

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 12, with a small set of test cases in listing 13.

```javascript
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);
    return new Due(function(settle) {
      args.push(settle);
      fn.apply(null, args);
    })
  }
}

module.exports = Due;
```

Listing 12: Implementation of Due

```javascript
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 13: Tests for the implementation of Due