Toward automatic update from callbacks to 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

Abstract
Javascript is the prevalent scripting language for the web. It let web pages register callbacks to react to user events. A callback is a function to be invoked later with a result currently unavailable. This pattern also proved to respond efficiently to remote requests. Javascript is currently used to implement complete web applications. However, callbacks are ill-suited to arrange a large asynchronous execution flow. *Promises* are a more adapted alternative. They provide a unified control over both the synchronous and asynchronous execution flows.

Promises are about to replace callbacks. This paper brings the first step toward a compiler to help developers prepare this shift. We present an equivalence between callbacks and Dues. The latter are a simpler specification of Promises developed for the purpose of this demonstration. From this equivalence, we implement a compiler to transform an imbrication of callbacks into a chain of Dues. This equivalence is limited to *Node.js*-style in-line asynchronous callbacks. We test our compiler over a small subset of github and npm projects, some of them with success, and show our results.

We consider this shift to be a first step toward the merge of elements from the data-flow programming model into the imperative programming model.

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

General Terms
Compilation

Keywords
Flow programming, Web, Javascript

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

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

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

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

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

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

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

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

In section 2 we define callbacks, Promises and then Dues. In section 3, we explain the transformation from imbrications of callbacks to sequences of Dues. In section 4, we present the different steps to automate the application of this equivalence, and expose its limitations. In section 5, we evaluate the developed compiler. In section 6, we present related works, and finally conclude in section 7.

2. DEFINITIONS

2.1 Callback

A callback is a function passed as a parameter to a function call. It is invoked by the callee to continue the execution with data 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 event in a stream.

Continuations are functions called asynchronously once a result is available.

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

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

A continuation is a function passed as an argument to allow the callee not to block the caller until its completion. The caller is able to continue the execution with other operations in parallel. The continuation is invoked later, at the termination of the callee to continue the execution as soon as possible and process the result ; hence the name continuation. It provides a necessary control over the asynchronous execution flow. It also brings a control over the data flow to replace the return statement at the end of a synchronous function. At its invocation, the continuation retrieves both the the execution flow and the result.

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

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

Listing 1: Example of a continuation

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

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

Listing 2: Example of a sequence of continuations

2.2 Promise

In a synchronous paradigm, the sequentiality of the execution flow is trivial. While in an asynchronous paradigm, this control is provided by continuations. Promises provide a unified control over the execution and data flow for both paradigms. The specification [4] defines a Promise as an ob-

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

ject that is used as a placeholder for the eventual outcome of a deferred (and possibly asynchronous) operation. Promises expose a `then` method which expects a continuation to continue with the result ; this result being synchronously or asynchronously available.

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

```
var promise = my_fn_pr(input)

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

Listing 3: Example of a promise

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

The listing 4 illustrates this chained composition. The functions `my_fn_pr_2` and `my_fn_pr_3` return promises when they are executed, asynchronously. Because these promises are not available synchronously, the method `then` synchronously returns intermediary Promises. The latter resolve only when the former resolve. This behavior allows to arrange the continuations as a flat chain of calls, instead of an imbrication of continuations.

```
my_fn_pr_1(input)
.then(my_fn_pr_2, onError)
.then(my_fn_pr_3, onError)
.then(console.log, onError);

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

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

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

2.3 From Continuation to Promise

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

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

Because Promises bring composition, the first step might seem trivial as it does not imply any imbrication to compose. However, as explained in section 2.2, Promises impose a control over the execution flow that continuations leave free. This control induces a common convention to hand back the outcome to the continuation.

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

To translate the different conventions into the Promises one, the compiler would need to identify them. Such an identification might be possible with a static analysis. todo references However, impracticable because of the heterogeneity. Indeed, in the browser, each library seems to provide its own convention.

In this paper, we are interested in the transformation from imbrications to chains, not from one convention to another. The *error-first* convention, used in *Node.js*, is likely to represent a large, coherent code base to test the equivalence over. For this reason, we focus only on the *error-first* convention. So, The compiler is only able to compile code that follows this convention. The convention used by Promises is incompatible. We propose an alternative specification to Promise following the *error-first* convention. In the next section we present this specification, called Due.

The choice to focus on *Node.js* is also motivated by our

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

intention to later compare the chained sequentiality of Promises with the data-flow paradigm. *Node.js* is a framework for real-time web applications ; it allows to manipulate flow of I/O data.

2.4 Dues

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

```
1  var my_fn_due = require('due').mock(my_fn);
2
3  var due = my_fn_due(input);
4
5  due.then(function continuation(error, result) {
6    if (!error) {
7      console.log(result);
8    } else {
9      throw error;
10   }
11 });
```

Listing 5: Example of a due

A due is typically created inside the function which returns it. In listing 5, the `mock` method create a Due-compatible function from `my_fn`. We illustrate the creation of a Due with the `mock` method, in listing 6.

At its creation, line 6, the due expects a callback containing the deferred operation, `my_fn`. This callback is executed synchronously with the function `settle` as argument to settle the Due, synchronously or asynchronously. The callback invokes the deferred operation line 8. `my_fn` being asynchronous, it expects a callback as last argument : `settle`. At the end of the operation, the deferred operation calls `settle` to settle the Due and save the outcome. Settled or not, the created Due is synchronously returned. Its `then` method allows to define a continuation to retrieve the saved outcome, and continue the execution after its settlement. If the deferred operation is synchronous, the Due settles during its creation and the `then` method immediately calls this continuation. If the deferred operation is asynchronous, this continuation is called during the Due settlement.

```
1  Due.mock = function(my_fn) {
2    return function() {
3      var _args = Array.prototype.slice.call(arguments),
4          _this = this;
5
6      return new Due(function(settle) {
7        _args.push(settle);
8        my_fn.apply(_this, _args);
9      })
10   }
11 }
```

Listing 6: Creation of a due

The composition of Dues is essentially the same than for Promises, detailed in section 2.2. Through this chained composition, Dues arrange the execution flow as a sequence of actions to carry on inputs.

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

3. EQUIVALENCE

We present the semantic equivalence between a continuation and a Due. This equivalence allows to transform imbrications of continuations into chains of Dues.

To illustrate the transformations, we use the function `my_fn` in listing 7. This function is tailored for this transformation, it both expects a callback and returns a Due. The transformation modifies the required signature of the callee. In the source of the transformation, the callee expects a continuation, while in the result of the transformation, the callee returns a Due. The modification of the signature of the callee is addressed in section 4.4.

```
1  var Due = require('due');
2
3  module.exports = function my_fn(arg, continuation) {
4    return new Due(function deferred(settle) {
5      setImmediate(function cont() { // Simulate an
           asynchron operation
6        var result = arg,
7            err = null;
8
9        if (continuation)
10         continuation(err, result);
11
12       settle(err, result);
13     })
14   })
15 }
```

Listing 7: `my_fn` expects a callback, and returns a Due

3.1 Simple equivalence

Continuations provide a control over the sequentiality of the asynchronous execution flow, and Dues brings the chained composition of this control. The equivalence between a single continuation and a Due does not involve composition, so the manipulation to transform listing 8 into 9 is trivial. It consist of calling the method `then` of the returned Due, and moving `continuation` to the arguments of this new call.

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

Listing 8: A simple continuation

```
1  var my_fn = require('./my_fn');
2
3  var arg = '1';
4
```

```
5 my_fn(arg)
6 .then(function continuation(err, res) {
7   console.log(res);
8 });
```

Listing 9: The syntax of a continuation and its Due equivalence are very similar

The relocation of the evaluation of `continuation` might alter the execution order, hence the semantic. But the definition of a *Function Expression* is free of side-effects. Indeed, it leaves intact the scope in which it is defined. The manipulation conserves the semantic, it is *sound*.

To summarize the specification, these *Expressions* possibly return a function.

— a *FunctionExpression*,
— an *ArrowFunction*,
— an *Identifier*, a *MemberExpression* or a *ThisExpression* pointing to a function,
— a *CallExpression*, *YieldExpression* returning a function, and
— a *NewExpression* creating a new function,

We focus only on continuations expressed as *Function Expressions*. For other types of continuations, the manipulation is either irrelevant, or unsound. Respectively, either the continuation prevent nesting, or its evaluation causes side-effects. For example, when using *Identifiers*, it is impossible to nest continuations, the equivalence would be irrelevant. When using *Immediately-Invoked Function Expression* (IIFE), a common pattern to return a closure, the manipulation is unsound. Before the manipulation, the evaluation of this expression would occur before the call to `my_fn`. While, after the manipulation, it would occur after the call to `my_fn`. The manipulation would prevent side-effects from happening before the call to `my_fn`. The manipulation is *sound* and relevant only when manipulating *FunctionExpression*.

3.2 Composition of nested continuations

The equivalence previously presented is incomplete, it leaves sequential operations nested. To transform an imbrication of continuations into a chain of Dues, we need to assure the composition of this simple equivalence. An example of nested continuations is illustrated in listing 10. The semantic equivalent chain of Dues is illustrated in listing 11. The composition of this equivalence requires at least two additional transformations.

— The nested continuation `cont2` is chained in the same indentation level as the first one, by a second call to the `then` method, line 12. This second call refers to the intermediary Due returned by the first call to the `then` method.
— For this chain to be possible, this intermediary Due must be linked with the Due returned by the nested call to `my_fn`. The Due from the nested asynchronous function `cont1` is retrieved, line 12, and returned line 10 to be internally linked to the intermediary Due.
— The definition of `shared_identifier` is relocated in the global scope to be accessible by both `cont1` and `cont2`. The accessibility of identifiers is developed in section 4.3.

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

Listing 10: Overlapping continuations definitions

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

Listing 11: Sequential continuations definitions using Dues

The composition of the equivalence leads to some semantical differences. It is unsound in some corner cases. In the next section, we explore the limits of the equivalence to ensure soundness, and allow automation.

4. AUTOMATION AND LIMITS

For this demonstration to be reasonable, we intentionally restricted it to continuations expressed as *Function Expression*, following the *error-first* convention. The automation of this equivalence exposes some other limitations in corner cases. We explain in the next paragraphs four compilation steps with their limitations and the choices made to address them.

4.1 Continuations

4.1.1 Identification

The equivalence is applicable only on continuations. The first compilation step is to identify the continuations among the callbacks. A continuation is a callback invoked *a)* only once, and *b)* asynchronously.

a). A Due is a placeholder for a single outcome. It is unable to hold the different results returned by the multiple invocations of an iterator or a listener. Dues replace only continuations.

b). Dues are designed to unify the control over both the synchronous, and asynchronous execution flow. The synchronous equivalent of a continuation is uncommon, but not impossible. However the equivalence is unsound when transforming an imbrication of callees invoking their callbacks synchronously. The execution order changes. A synchronous continuation is executed before the instructions following

the callee. In the equivalent chain of Dues, the synchronous continuation is executed after these following instructions.

Spotting a continuation implies to identify that the callee invokes its callback asynchronously, and only once. There is no syntactical difference between a synchronous and an asynchronous callee. And it is impossible to assure a callback to be invoked only once, because the implementation of the callee is often unavailable statically. Therefore, the identification of continuations is necessarily based on semantical differences.

4.1.2 Composition

Moreover, some uncommon continuations are not compatible with the composition of this equivalence. For the chained composition to be possible, the parent continuation must return the Due returned by the nested asynchronous function. This Due is then linked internally to the next continuation. The modifications to retrieve and return this Due is sound only if it takes place in classic situations. In *Node.js*, continuations are triggered as the root of a new call stack. There is no caller waiting for a result to be returned to. The only purpose of a *Return Statement* is to control the execution flow, not to hand a result back like in a regular function. Since it should not return any significant value, we assume it is safe for a continuation to return a Due.

Similarly, classic asynchronous functions do not synchronously return any significant value. The result of an asynchronous operation is available only asynchronously. Therefore, we assume it is safe for those functions to return a Due.

These two assumptions are verified and the equivalence is sound in most common cases. However, the previous assumptions are false in some uncommon cases. For example, the asynchronous function `SetTimeout` returns an identifier pointing to the created timer, so it cannot return the Due expected to continue the chain..

Identifying compatible continuations means differentiating synchronous from asynchronous functions, enumerating the callbacks invocations and detecting return values. The compiler would need to have a deep understanding of the control and data flows of the program. Because of the highly dynamic nature of Javascript, this understanding is either unsound, limited, or complex. For these reasons, we leave to the developer the identification of compatible continuations among the identified callbacks. They are expected to understand the limitations of this compiler, and the semantic of the code to compile.

Because of this interaction, the compilation process is not automatic at an individual scale. However, we are working on an automation at a global scale. We expect to be able to identify of a continuation only based on its callee name, *e.g.* `fs.readFile`. We built a service to gather these names along with their identification. The compiler query this service to present an estimated identification, and send back the final choices to refine it. In future works, we would like to study the possibility of easing the compilation interaction.

4.2 Chains

The second compilation step is to identify imbrications of continuations. A Javascript program is structured as a tree of functions definitions nested one in its parent. An imbrication of continuations is a subtree in this tree. A continuation is a leaf only if its direct parent is also a continuation. Otherwise, it is the root of a tree.

A tree of continuations can not contain a loop, nor a function definition that is not a continuation. Both modify the linearity of the execution flow which is required for the equivalence to keep the semantic. Indeed, a call nested inside a loop might return multiple Dues, while only one is returned to continue the chain. And a call nested inside a function definition is unable to return the Due to continue the chain. On the other hand, conditional branching leaves the execution linearity and the semantic intact. If the nested asynchronous function is not called, no Due needs to be returned, as the chain is expected stop its execution.

The compositions of continuations and Dues are arranged differently. Continuations can nest multiple children, they are arranged as a tree, while Dues are arranged as a chain. The compiler trims each tree of continuations to get chains to translate into Dues. If a continuation has more than one child, the compiler try to find a legitimate child to continue the chain. The legitimate child is the only parent among its siblings. If there is several parents among the children, then none are the legitimate child. The non legitimate children start a new tree.

From this step, results chains of continuations assured to be transformable into sequences of Dues. However, this transformation modifies the scopes organization which is addressed in the next section.

4.3 Scope disjunction

The third compilation step is to assure the availability of identifiers after the manipulation. In listing 10, the definitions of `cont1` and `cont2` are overlapping, so are their scopes [7] . The function `cont1` shares its identifiers with `cont2`. In listing 11, the definitions of `cont1` and `cont2` are siblings, they have disjoint scopes. The manipulation changes the hierarchy of scopes, it modifies the semantic. To keep the semantic intact, the scopes needs to be disjoints before the manipulation.

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

An identifier is in the scope of its defining function and all its descendants. We define two scopes as disjoints if none reference an identifier defined inside the other. For two scopes to be disjoints, all shared identifiers must be define in a common parent scope.

something to write here

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

In listing 10, the identifier `shared_identifier` is accessible both from its defining function `cont1` and the descendant `cont2`. However, in listing 11, the scope of `cont1` and `cont2` are disjoints. For the identifier `shared_identifier` to still be accessible in both function, its definition is relocated in the first common parent scope, which is the global scope.

The compiler iterate over each continuation in a chain, and relocate shared variables in the root of the chain. If there is a conflict with another variable in this root scope, it is necessary to rename one of these variables.

4.4 Asynchronous function mocking

As seen in listing 6 section 2.4, a function returning a Due is simply a wrapping of a function expecting a continuation. The fourth compilation step is to wrap the callee function to return a Due. The `due` library already provides such a wrapper, the `mock` function.

To impact the semantic a minimum, the compiler wrap function calls inline. In Javascript, a function call is invoked in a context accessible through the `this` keyword. Functions are invoked in the global context, while methods are invoked in the context of their owner. In lisitng 12, the object `my_obj` is initialized with the method `my_mth` which wrap the function `my_fn`. Line 11, the method `mock` returns a function wrapping the method `my_obj.my_mth`. This function is then called, line 12, with `my_obj` as context and `input` as argument, to preserve the semantic of the call to `my_obj.my_mth`.

```
1  my_obj = {
2    my_mth: function(input, cont) {
3      return my_fn(this.my_self + input, cont);
4    },
5    my_self: 'self'
6  }
7
8  my_obj.my_mth(input, continuation);
9
10 require("due")
11 .mock(my_obj.my_mth)
12 .call(my_obj, input)
13 .then(continuation);
```

Listing 12: Inline mock of the callee

This transformation is simple and sound for a limited subset of callees. The callee of a call expression can be

— a *MemberExpression* or an *Identifier*, or
— a *CallExpression* returning a function, or a property pointing to a function.

In the second case, the semantic would be modified by the double evaluation of the context, `my_obj`. For example, in the case of this owner being an *Immediately Invoked Function Expression*, this *IIFE*, would be evaluated twice, line 11 and 12.

5. EVALUATION

6. RELATED WORKS

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

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

In the Department of Computing, Imperial College London, S. Maffeis, P. Gardner and G. Smith realised a large body of work around the static analysis of Javascript. Their work is based around an operational semantic[9] to bring program understanding[12, 4, 3]. Their goal seems to revolve around Security applications of this analysis[11, 10]. In the industry, there already exist some security tools based on static analysis, we can cite for example, the company Shape Security [10]. They developed *Esprima*, a Javascript parser, and a serie of tools to help static analysis.

In a collaboration between the programming language research groups at Aarhus University and Universität Freiburg, P. Thiemann, S. Jensen and A. Møller are working on the static analysis of Javascript. They presented a tool providing type inference using abstract interpretation[13, 7, 6]. Their goal is to improve the tools available to the Javascript developer[2]. The industry seems to follow the same trend. Facebook released flow [11] on October 26 2014, a static type checker for Javascript.

Another example is the adaptation of the points-to analysis from L. Andersen's thesis work[1] to Javascript[5].

paper about the combination of execution flow and data flow

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

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

suggestive of a data flow process. It is a chain of operations feeding the next with the result of the previous.

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

Références

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

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

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

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

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

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

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

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

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

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

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

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

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

APPENDIX

A. DUE IMPLEMENTATION

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

```
function Due(callback) {

  var self = this;

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

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

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

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

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

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

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

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

Due.prototype.isDue = true;

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

  var self = this;
  return new Due(function(settle) {
    /*  ++ Returned Due ++
     *  If the current due is settled (1), then the
         future is available.
     *  If this future value is a due, link the returned
         due to it (2)
```

```
     *  If this future value is not a due, settle the
          returned due with the future value (3).
     *  If the current due is pending (4), add the
          settlement handler to the followers (5), to be
          deferred to the future dues of the current due.
          (resolve.4)
     */

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

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

module.exports = Due;
```

Listing 13: Implementation of Due

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

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

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

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

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

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

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

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

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

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

    var count = 0;

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

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

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

    var count = 0;

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

```
124           done()
125       }
126
127       d.then(then);
128       d.then(then);
129     })
130
131
132     it('should expose the mock function', function(){
133       if (D.mock === undefined)
134         throw 'mock not available'
135     })
136
137     // returned value should either be a vow, or a value
138 })
```

Listing 14: Tests for the implementation of Due