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 lets 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.

The next version of Javascript proposes to replace callbacks with Promises. 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 asynchronous callbacks defined *in situ*. We test our compiler over 64 *npm* packages and show our results. 9 of them are compatible and compile successfully.

We consider this shift to be a first step toward the merge of concepts 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 from 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 [13].

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, ECMAScript 6 [3]. 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 adoption of Promises libraries, that there are almost 40 different implementations [4].

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

1. http://www.w3.org/DOM/
2. http://maxogden.github.io/callback-hell/
3. http://people.mozilla.org/~jorendorff/es6-draft.html
4. https://github.com/promises-aplus/promises-spec/blob/master/implementations.md

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 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 simpler 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 64 *npm* packages, 9 of them with success.

In section 2 we define callbacks, Promises and Dues. In section 3, we explain the transformation from imbrications of callbacks to sequences of Dues. In section 4, we present a compiler to automate the application of this equivalence. 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. In an asynchronous paradigm, parallelism is trivial, but 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 while the callee runs in background. 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 which essentially replaces the `return` statement at the end of a synchronous function. At its invocation, the continuation retrieves both the execution flow and the result.

The convention on how to hand back the result must be common for both the callee and the continuation. 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 ECMAScript 6 specification [5] defines a Promise as an object 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 conti-

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

nuation 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 onError(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 leads 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 some coarse-grained data-flow programming paradigms, such as Flow-based programming [17].

2.3 From continuations to Promises

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 an imbrication of continuations into a chained composition of Promises.

We identify two steps in this transformation. 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 chained composition, the first step might seem trivial as it does not imply any imbrication to transform into chain. 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` [6] method expects an object with different continuations for success, errors and various other events during the asynchronous operation. *Q* [7], a popular library to control the asynchronous flow, exposes two methods to define continuations : `then` for successes, and `catch` for errors. On the other hand, the *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 use different conventions than the Promise specification detailed in section 2.2. They present strong semantic differences, despite small syntactic differences.

To translate these different conventions into the Promises one, the compiler would need to identify them. Such an identification might be possible with static analysis methods such as the points-to analysis [20], or a program logic [6, 3]. However, it seems impracticable because of the number and semantical heterogeneity of these conventions. 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. For this reason, we focus only on the *error-first* convention. Thus, our 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 intention to compare later the chained sequentiality of Promises with the data-flow paradigm. *Node.js* allows to ma-

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

nipulate streams of messages. This proved to be efficient for real-time web applications manipulating streams of user requests. Both Promises and data-flow arrange the computation in chains of independent operations.

2.4 Due

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 available online [8]. 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, line 1, the `mock` method wraps `my_fn` in a Due-compatible function. The rest of this code is similar to the Promise example, listing 3.

We illustrate in listing 6 the creation of a Due through the `mock` method. At its creation, line 6, the Due expects a callback containing the deferred operation, which is `my_fn` here. This callback is executed synchronously with the function `settle` as argument to settle the Due, synchronously or asynchronously. The `settle` function is pushed at the end of the list of arguments. The callback invokes the deferred operation with this list of arguments, and the current context, line 8. When finished, the latter calls `settle` to settle the Due and save the outcome. Settled or not, the created Due is always 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 mocked_fn() {
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

8. https://www.npmjs.com/package/due

The composition of Dues is essentially the same than for Promises (see 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* for 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 step between classical continuation and Promises. We present in section 3 the equivalence between continuations and Dues.

3. EQUIVALENCE

We present the transformation from a nested imbrication of continuations into a chain of Dues. We explain the three limitations imposed by our compiler for this transformation to preserve the semantic. They preserve the execution order, the execution linearity and the scopes of the variables used in the operations.

3.1 Execution order

Our compiler spots function calls with a continuation, which are similar to the abstraction in (1). It wraps the function fn into the function $fn_{\mathbf{due}}$ to return a Due. And it relocates the continuation in a call to the method **then**, which references the Due previously returned. The result should be similar to (2). The differences are highlighted in bold font.

$$fn([arguments], continuation) \tag{1}$$

$$fn_{\mathbf{due}}([arguments]).\mathbf{then}(continuation) \tag{2}$$

The execution order is different whether *continuation* is called synchronously, or asynchronously. If fn is synchronous, it calls the *continuation* within its execution. It might execute *statements* after the execution of *continuation*. If fn is asynchronous, the continuation is called after the end of the current execution, after fn. The transformation erases this difference in the execution order. In both cases, the transformation relocates the execution of *continuation* after the execution of fn. For synchronous fn, the execution order changes ; the execution of *statements* at the end of fn and the continuation switch. The latter must be asynchronous to preserve the execution order.

3.2 Execution linearity

Our compiler transforms a nested imbrication of continuations, which is similar to the abstraction in (3) into a flatten chain of calls encapsulating them, like in (4).

$$\begin{aligned} &fn1([arguments], cont1\{ \\ &\quad declare\ variable \leftarrow result \\ &\quad fn2([arguments], cont2\{ \\ &\quad\quad print\ variable \\ &\quad \}) \\ &\}) \end{aligned} \tag{3}$$

```
declare variable
fn1due([arguments])
.then(cont1{
    variable ← result
    fn2due([arguments])
})
.then(cont2{
    print variable
})                                    (4)
```

An imbrication of continuations must not contain any loop, nor 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. A call nested inside a loop returns multiple Dues, while only one is returned to continue the chain. A function definition breaks the execution linearity. It prevent the nested call to return the Due expected 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, the execution of the chain stops as expected.

We demonstrate the equivalence with a sequence of two continuations. The equivalence is sound for a sequence of n continuations.

3.3 Variable scope

In (3), the definitions of *cont*1 and *cont*2 are overlapping. The *variable* declared in *cont*1 is accessible in *cont*2 to be printed. In (4), however, definitions of *cont*1 and *cont*2 are not overlapping, they are siblings. The *variable* is not accessible to *cont*2. It must be relocated in a parent function to be accessible by both *cont*1 and *cont*2. To detect such variables, the compiler must infer their scope statically. Languages with a lexical scope define the scope of a variable statically. Most imperative languages present a lexical scope, like C/C++, Python, Ruby or Java. The subset of Javascript excluding the built-in functions `with` and `eval` is also lexically scoped. To compile Javascript, the compiler must exclude programs using these two statements.

4. COMPILER

We build a compiler to automate the application of this equivalence on existing Javascript projects. The compilation process contains two important steps, the identification of the continuations, and the generation of chains.

4.1 Identification of continuations

The first compilation step is to identify the continuations and their imbrications. The nested imbrication of callbacks only occurs when they are defined *in situ*. The compiler detects a function definition within the arguments of a function call. This detection is based on the syntax, and is trivial.

Not all detected callbacks are continuations, but the equivalence is applicable only on the latter. A continuation is a callback invoked only once, asynchronously. Spotting a continuation implies to identify these two conditions. 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 statically unavailable. Therefore, the identification of continuations is necessarily based on semantical differences. To recognize these differences, 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. Instead, we choose to 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.

We provide a simple interface for developers to interact with the compiler. We built this interface around the compiler in a web page available online [9] to reproduce the tests. The web technologies allow to quickly build an interface for a wide variety of computing devices.

This interaction prevents the complete automation of the individual compilation process. However, we are working on an automation at a global scale. We expect to be able to identify a continuation only based on the name of its callee, *e.g.* `fs.readFile`. We built a service to gather these names along with their identification. The compiler query this service to present to the developer an estimated identification. After the compilation, it sends back the identification corrected by the developer to refine the future estimations. In future works, we would like to study the possibility for such a service to assist, and ease the compilation process.

4.2 Generation of chains

The compositions of continuations and Dues are arranged differently. Continuations structure the execution flow as a tree, while the chain of Dues arrange it sequentially. A parent continuation can have several children, not a Due. The second compilation step is to identify the imbrications of continuations to transform them into chains.

The compiler trims each tree of continuations to get chains that 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. This step yields chains of continuations assured to be transformable into sequences of Dues. The code generation from these chains is straightforward from the equivalence.

5. EVALUATION

To validate our compiler, we compile several Javascript projects likely to contain continuations. We present the results of these tests.

The compilation of a project requires user interaction. To conduct the test in a reasonable time, we limit the test set to a minimum. We search the *Node Package Manager* database to restrict the set to *Node.js* projects. We refine the selection to web applications depending on the web frame-

9. `compiler-due.apps.zone52.org`

work *express*, but not on the most common Promises libraries such as *Q* and *Async*. We refine further the selection to projects using the test frameworks *mocha* in its default configuration. The test set contains 64 projects. This subset is very small, and cannot represent the wide possibilities of Javascript. However, we believe it is sufficient to represent a majority of common cases.

For each project, we verify that the project is correctly tested, and passes the tests. During the compilation, we identify the compatible continuations among the detected callbacks. We apply the unmodified test on the compilation result. The compilation result should pass the tests as well. This is not a strong validation, but it assures the compiler to work as expected in most common cases.

Of the 64 projects tested, almost a half, does not contain any compatible continuations. We reckon that these projects use continuations the compiler is unable to detect. The other projects were rejected by the compiler because they contain `with` or `eval` statements, they use Promises libraries we didn't filter previously. 9 projects compiled successfully. The compiler did not fail to compile any of these projects.

Over the 10 successfully compiled projects, the compiler detected 172 callbacks. We manually identified 56 of them to be compatible continuations. The false positives are mainly the listeners that the web applications register to react to user requests.

One project contains 20 continuations, the others contains between 9 and 1 continuations each. On the 56 continuations, 36 are single. The others 20 continuations belong to imbrications of 2 to 4 continuations. The result of these tests prove the compiler to be able to successfully transform imbrications of continuations. The details of these results are available in Appendix A.

6. RELATED WORKS

To our knowledge, our work is the first to explore the transformation from continuations to Promises in Javascript. And also to state the similarity between Promises and data-flow programming. This section relates the various works related with ours. Our work is based on the previous work on Promises and Futures [13], 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 identify 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 [14] to bring program understanding [18, 7, 6, 3]. Their goal seems to revolve around security applications of this analysis [16, 15]. 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 [19, 10, 9]. Their goal is to improve the tools available for Javascript developers [2]. Another example of interest for Javascript static analysis is the adaptation of the points-to analysis from L. Andersen's thesis [1] to Javascript, by D. Jang *et al.* [8] and S. Wei *et al.* [20].

The industry seems to follow the same trends. There are some security tools based on static analysis. We can cite for example, the company Shape Security [12]. They developed *Esprima*, a Javascript parser, and a serie of tools to help static analysis. Facebook released flow [13] on 26 October 2014, a static type checker for Javascript.

Promises combine controls over the execution and the data flow. It arrange the execution parts sequentialy and assign the result of one into the inputs of the next. This arrangement seems similar to some works on the field of functional and data-flow programming [11, 4, 17, 12]. We consider it a first step in the merge of elements from the data-flow paradigm into the imperative paradigm. The Functional Reactive Programming paradigm pushes the intrication of data and control-flow even further [21, 5].

7. CONCLUSION

In this paper, we introduce a compiler to automatically transform an imbrication of continuations into a sequence of Dues. First, we define callbacks and Promises, and then introduce Dues, a simpler specification to Promises. We explain the transformation from the nested imbrication of continuations to a chain of Dues, and present a compiler to automate this transformation on existing code bases. The compiler is evaluated against a set of *npm* projects.

This transformation flattens a nested imbrication of continuations. The result is a sequence of operations encapsulated in Dues. The latter, like Promises, arrange both the control and data flow. The outcome of an operation is assigned as the input of the next. Such an arrangement is very suggestive of a data flow process, that is a chain of operations feeding the next with the result of the previous.

We aim at pushing further this analogy. A web application manipulates a flow of user requests. We think it is possible to arrange such an application as a chain of independent operations communicating by messages. We want to develop the compiler further to bring complete independence to the asynchronous operations delimited by the Dues. This independence would allow to transform a monolithic program into a chain of independent asynchronous operations linked by a flow of messages. We expect a possibility for new execution models to take advantage of this independence to bring performance scalability. Developers could continue to use the monolithic model for its evolution advantages, and leave the performance burdens to the execution engines.

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

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] M Bodin et A Charguéraud. "A trusted mechanised JavaSript specification". In : *Proceedings of the 41st ...* (2014).

[4] Albert Cohen, Léonard Gérard et Marc Pouzet. "Programming parallelism with futures in lustre". In : *EMSOFT*. 2012, p. 197–206.

[5] C Elliott et Paul Hudak. "Functional reactive animation". In : *ACM SIGPLAN Notices* (1997).

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

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

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

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

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

[11] Wesley M Johnston, J R Hanna et Richard J Millar. "Advances in dataflow programming languages". In : *ACM Computing Surveys (CSUR)* 36 (2004), p. 1–34.

[12] Gilles Kahn. "The semantics of a simple language for parallel programming". In : (1974).

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

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

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

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

[17] JP Morrison. *Flow-Based Programming*. 1994, p. 1–377.

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

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

[20] S Wei et BG Ryder. "State-sensitive points-to analysis for the dynamic behavior of JavaScript objects". In : *ECOOP 2014–Object-Oriented Programming* (2014).

[21] Daniel Winograd-Cort et Paul Hudak. "Wormholes : Introducing Effects to FRP". In : *ACM SIGPLAN Notices* 47.12 (jan. 2013), p. 91. doi : 10.1145/2430532.2364519.

APPENDIX

A. EVALUATION RESULTS

On the 64 projects tested

- **29** (45.3%) do not contain any compatible continuations,
- **10** (15.6%) are not compilable because they contain `with` or `eval` statements,
- **5** (7.8%) use less common asynchronous libraries we didn't filter previously,
- **4** (6.3%) are not syntactically correct,
- **4** (6.3%) fail their tests before the compilation,
- **3** (4.7%) are not tested, and
- **10** (14.0%) compile successfully.

The compiler do not fail to compile any projects.

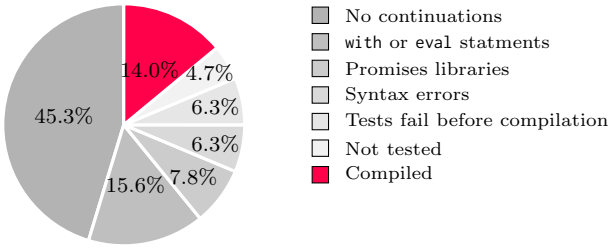

Figure 1: Compilation results distribution

29 projects contains no compatible continuations.

- `app-json-fetcher`
- `express-resource-plus`
- `fizzbuzz-hypermedia-server`
- `flair`
- `generator-wikismith`
- `brokowski`
- `claus`
- `costa`
- `express-device`
- `heroku-proxy`
- `http-test-servers`
- `jellyjs-plugin-httpserver`
- `loopback-angular-cli`
- `loopback-explorer`
- `monami`
- `mongoose-epxress`
- `nodebootstrap-server`
- `oauth-express`
- `public-server`
- `scrapit`
- `sik`
- `sonea`
- `vsoft-explorer`
- `webs-weeia`
- `code-connect-server`
- `csp-endpoint`
- `glsl-transition-minify`
- `moby`

— squirrel-server

10 projects contains eval or with statements.

— arkhaios
— browserman
— infectwit
— ldapp
— levelhud
— manet
— solid
— swac
— swac-odm
— adnoce

5 projects already use asynchronous frameworks.

— boomerang-server (es6-promise)
— hyper.io (when)
— node-lanetix-microservice (bluebird)
— librarian (bluebird)
— webtasks (subtask)

4 projects failed their tests before the compilation.

— express-orm-mvc
— hangout
— derp
— ord

3 Projects do not provide tests.

— tuzi
— inchi-server
— otwo

9 projects successfully compile. The compiler detected 172 callbacks, 52 of them are compatible continuations.

	continuations		chains length			
name	total	compiled	1	2	3	4
express-couchUser	40	20	9	2	1	1
gifsockets-server	3	1	1			
node-heroku-bouncer	7	3	3			
moonridge	37	6	2	2		
redis-key-overview	14	9	6		1	
slack-integrator	6	3	1	1		
timbits	34	8	8			
tingo-rest	12	4	4			
express-endpoint	19	2	2			
total	172	54	36	5	2	1