Dependence-Driven Delimited CPS Transformation for JavaScript

Laure Philips Joeri De Koster Wolfgang De Meuter Coen De Roover
Software Languages Lab, Vrije Universiteit Brussel, Belgium
{lphilips, jdekoste, wdmeuter, cderoove}@vub.ac.be

Abstract

In today's web applications asynchronous requests to remote services using callbacks or futures are omnipresent. The continuation of such a non-blocking task is represented as a callback function that will later be called with the result of the request. This style of programming where the remainder of a computation is captured in a continuation function is called *continuation-passing style* (CPS). This style of programming can quickly lead to a phenomenon called "callback hell", which has a negative impact on the maintainability of applications that employ this style. Several alternatives to callbacks are therefore gaining traction within the web domain. For example, there are a number of frameworks that rely on automatically transforming code, written without callbacks in a direct sequential style, into the continuation-passing style. However, these frameworks often employ a conservative approach in which each function call is transformed into CPS. Even when they do a selective transformation they will often encapsulate the entire remainder of the computation in the continuation. This conservative approach can sequentialise requests that could otherwise be run in parallel. So-called delimited continuations can remedy, but require special marks that have to be manually inserted in the code for marking the beginning and end of the continuation. In this paper we propose an alternative strategy in which we apply a delimited CPS transformation that operates on a Program Dependence Graph instead to find the limits of each continuation. We implement this strategy in JavaScript and demonstrate its applicability to various web programming scenarios.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code Generation

Keywords CPS transformation, Program Dependence Graph, JavaScript

1. Introduction

Transformation of code into continuation-passing style is a well-known strategy for compiling and optimisation [1]. For example, compilers for higher-order languages frequently use this style as an intermediate representation of a program. This is because the compiler can perform more optimisations on this intermediate representation than on the source code [7].

A more recent application of continuations can be found in the web domain. Because of its distributed nature, most web developers are used to programming with continuation functions, often referred to as callbacks. For example, the PLT Scheme Web Server [15] uses continuations to handle requests from clients. The browser environment and NodeJS (which enables running JavaScript on the server) are both single-threaded. This means that developers are forced to write in an asynchronous style with explicit continuations in order for their applications to remain responsive. For example, callbacks are needed in the user interface, e.g., when the user clicks a button, or when accessing a service on another distributed node. However, programming directly in continuation-passing style is laborious and error-prone. When different asynchronous requests rely on the result of other previous requests these callbacks can become deeply nested. This well-known problem is called the *callback hell* or *pyramid of doom* which limits the maintainability of the resulting code. Often every callback is also responsible for its own failure handling (e.g., the service you are trying to reach is not available), further complicating the spaghetti of callbacks. As a result, promises, generators and reactive programming gain traction with web programmers. These approaches enable the programmer to manage asynchronous calls in a more direct way. Next to these approaches, CPS transformation tools are an alternative approach for web programming that is on the rise. By eliminating callback functions completely, they enable programmers to write code in a synchronous style without impacting the responsiveness of the resulting application. The transformation of the synchronous code into the CPS version of the same program is left to the compiler or code transformation tool. Section 5 provides an in depth discussion of a number of these tools.

One caveat of such an automatic transformation is that the resulting CPS code which is outputted by the compiler is not always meant to be human readable. Moreover, the control flow of the direct-style program is lost after the transformation into a CPS. This drastically impacts the development process for the developer, as the transformed code is often the one that is tested and debugged. This raises an important issue, a debugger that works on the transformed code should be tightly integrated with the compiler such that runtime bugs can be reported on the original direct-style program rather than on the transformed code. Some specialised tools already exist that aim to help the programmer with this issue [26], such as tools that support inverse CPS transformations.

Another problem with a number of existing code transformation tools is that they follow a conservative approach while transforming the direct-style program in which the entire remainder of the program is encapsulated in each continuation. This can lead to the sequentialisation of independent requests that could otherwise

run in parallel. For example, when dealing with concurrency the notion of the rest of the computation is not practical [10]. Subcontinuations [10], sometimes also referred to as delimited, partial or compose-able continuations [8], address this issue by capturing only a part of the remainder of the program. The borders of a continuation have to be made explicit through delimited control operators such as `shift` and `reset`. Delimited continuations have proven useful in the context of concurrency, partial evaluation and even mobile computing [12]. In order to improve the performance of the resulting application, a transformation tool that targets delimited continuations might be desirable.

A third problem with some CPS transformation tools is that they convert each and every function into its continuation-passing equivalent. This is often undesirable as only functions that have computational effects should be converted to CPS [21]. Deciding whether a function should be transformed can be based on a strictness analysis [4], effect analysis [21], or on programmer-provided annotations as in Scala [23]. These selective transformations are needed when transformed code needs to be linked with untransformed code from libraries, or in situations where the performance overhead introduced by the transformation [5] should be minimised.

In this paper we propose an alternative CPS transformation tool based on delimited continuations that is guided by a control and data flow analysis of the program. Our approach enables transforming a direct-style program into its continuation-passing form without the need for explicit delimited operators. To this end, we construct a Program Dependence Graph of the program; a directed graph of which the edges correspond to data and control dependences [14, 27]. These graphs have been shown by Kuck et al. [16] to be an appropriate vehicle for developing program optimisations. We improve upon existing transformation tools on the nesting level of independent callbacks and remove for the need for operators at every call site. Moreover, our approach improves the readability of the resulting code by respecting the structure and variable names of the original program as much as possible.

The contributions of this paper are as follow:

- A delimited CPS transformation based on a dependence graph instead of explicit dedicated control operators,
- General approach to selective CPS transformation on top of the delimited CPS transformation,
- Concrete implementation of this approach for JavaScript.

2. Motivation

In this section we illustrate the issues raised in the previous section by means of a small motivating example written in NodeJS. Our example, as shown in Listing 1, accomplishes two separate tasks. It first requests the temperature for the city of Brussels from a weather service and writes the result to a file (lines 5–24). Next, it starts up a server that reads out a welcome text from a file and returns it to every client that connects (lines 33–39).

```
1  var fs      = require('fs'),
2      request = require('request'),
3      http    = require('http');
4  /* Get temperature for given location */
5  function getTemperature (city, cb) {
6    var url = 'http://weatherapi/weather?q=' + city;
7    request(url, function (error, response, body) {
8      if (error)
9        cb(error, null);
10     else if (response.statusCode == 200) {
11       var data = JSON.parse(body),
12           temp = parseFloat(data.temp_C);
13       fs.writeFile('temp.txt',temp,function (err){
14         if (err)
15           cb(err, null);
16         console.log('temperature written');
17           cb(null,  temp);
18       })
19     }
20     else {
21       cb(new Error('Get temperature'), null);
22     }
23   })
24 }
25
26 getTemperature('Brussels', function (err, temp) {
27   if (err)
28     throw err;
29   console.log(temp);
30 }
31
32  /* Create the server */
33 http.createServer(function (req, res) {
34   fs.readFile('welcome.txt', function (err,data){
35     if (err)
36       throw err;
37     res.write(data.toString());
38   })
39 }).listen(8080, 'localhost')
```

Listing 1. Callbacks in Node.js.

The code depicted in Listing 1 has been simplified at certain parts but still captures the essence of the example. Because the temperature file can only be written (line 13) once the result of the temperature request (line 7) is known the callback for `fs.writeFile` needs to be nested inside the callback for the temperature `request`. In more complex examples any of these dependencies between callbacks introduces another level of CPS. This could result in an arbitrarily deep nesting of callback functions, rendering the program difficult to read, debug, and maintain. Every callback is moreover responsible for handling its error argument, further complicating the code.

Note that the callback for `createServer` on line 33 does not encapsulate the call to `getTemperature` or vice versa. This was intentional as those two parts of the application are independent. This becomes important later as more conservative approaches of automatic CPS transformation tools might not detect this.

The most common solution to structuring asynchronous code and avoiding scattered error handling in JavaScript is by using promises. Asynchronous requests in JavaScript return a promise, which can be *consumed* by calling the `then` primitive which accepts as an argument a function that will be called once the promise is fulfilled. Because the `then` primitive itself returns a promise these can effectively be chained together, thus avoiding the nesting of callbacks. Additionally, promises allow error handling code to be grouped into one single catch-clause, instead of being scattered throughout the different nested callbacks. However, promises merely provide a syntactic improvement over manual callback functions. The program still needs to be written in a CPS where the continuation of the program has to be provided as the argument of the `then` primitive.

Alternatively there exist a number of semi-automated CPS transformation tools that generate a CPS version of the program from a direct-style implementation. Libraries such as CONTINUATION.JS and STREAMLINE.JS[1] are examples of such tools. What these two libraries specifically have in common is that all asynchronous calls can be written down as synchronous function calls that take a special construct as additional argument instead of a callback. That construct marks a function call for the library to guide its CPS transformation.

Listing 2 shows how our motivating example could be written down using CONTINUATION.JS. Each of the asynchronous calls of

[1] `https://github.com/BYVoid/continuation` `https://github.com/Sage/streamlinejs`

Listing 1 has been transformed into a synchronous function call marked with an extra `cont` parameter. The `cont` special construct takes the same parameters as the callback function it replaces. The example can be implemented in a synchronous style. The continuation of the program no longer needs to be stored in an extra callback and thus the program no longer needs to be written in a CPS.

```
1  function getTemperature(city, callback) {
2    var url = 'http://weatherapi/weather?q=' + city,
3        body, response;
4    request(url, cont(error, response, body));
5    if (error)
6      throw error;
7    else if (response.statusCode == 200) {
8      var data = JSON.parse(body),
9          temp = parseFloat(data.temp_C);
10     fs.writeFile('temp.txt', temp, cont(error));
11     if (error)
12         throw error;
13     console.log('temperature written');
14   }
15 }
16
17 getTemperature('Brussels', cont(temperature));
18 console.log(temperature);
19
20  /* Create the server */
21 http.createServer(function (req, res) {
22   fs.readFile('welcome.txt', cont(error, data));
23   if (error)
24     throw error;
25   res.write(data.toString());
26 }).listen(8080, 'localhost')
```

Listing 2. Writing synchronous code with CONTINUATION.JS.

Failure handling in CONTINUATION.JS is supported by means of the familiar `try/catch` block. For example, the programmer could add a `try/catch` block around the call to `getTemperature` at line 17. The transformation tool will then make sure that any runtime errors that are thrown in the transformed code are properly propagated to the correct catch block.

The code generated by CONTINUATION.JS is depicted in Listing 3. For saving space, we have only included the transformed code for lines 17–26 of Listing 2. As expected, for every marked function call corresponding callback functions have been generated and their return values are bound to the correct declarations. The details of the transformed code are not important, however we do want to point out an important difference between the control flow of the resulting code and the control flow of our initial version with a manual CPS transformation. As noted earlier, the calls to `getTemperature` and `createServer` are independent of each other and can be run concurrently. However, because the transformation tool does not have access to this information it will conservatively nest the call to `createServer` in the continuation of the call to `getTemperature` (line 4). This sequentialisation of independent requests has several drawbacks with respect to performance, fault-tolerance and responsiveness. Firstly, because of this sequentialisation multiple independent requests cannot be issued in parallel. For example, this decreases the performance for those applications that issue requests to multiple services and then combine the results as those requests will be sequentialised. Secondly, if one request fails, the control flow of the program is broken and other requests further up the chain are not executed, severely limiting the fault-tolerance of the entire application. This is especially problematic when issuing several independent requests. If one request fails this can cause other nested independent requests never to be executed. Lastly, the responsiveness of the application can be impacted as well as one of the callbacks that is responsible for reacting to user input might unintentionally be nested within other long-running requests.

The CONTINUATION.JS library has an advantage over other libraries in terms of maintainability as it preserves the variables names of the original program after the transformation. However, as noted before, ideally the debugger is integrated with the transformation tool in such a way that runtime bugs can be reported on the original code. Section 5 gives a more in depth analysis and comparison of the existing frameworks.

```
1  getTemperature('Brussels', function (arguments,
       _$param4) {
2    temperature = _$param4;
3    console.log(temperature);
4    http.createServer(function (req, res) {
5      var err, data;
6      fs.readFile('temp.txt', function (arguments,
           _$param5, _$param6) {
7        err = _$param5;
8        data = _$param6;
9        if (err) {
10         throw err;
11       }
12       res.write(data.toString());
13     }.bind(this, arguments));
14   }).listen(8080, 'localhost');
15 }.bind(this, arguments));
```

Listing 3. Generated code by CONTINUATION.JS for Listing 2.

In conclusion, with our approach we aim to improve over existing work on three levels:

- **Concurrency**: Existing approaches conservatively wrap each asynchronous request in the continuation of the previous request regardless of whether they are independent or not. This limits the performance, fault-tolerance and responsiveness of the resulting application.
- **Implicit transformation**: A lot of existing approaches require the programmer to not only explicitly mark every asynchronous request in the original code but also explicitly mark every function that can potentially issue an asynchronous request in its control flow. In this paper we advocate for an implicit approach where the burden is put on the transformation tool to identify which calls need to be transformed.
- **Maintainability**: To improve maintainability the transformed code should preserve as much as possible the structure and variable names of the original program. This facilitates understanding, testing and debugging of the transformed code. Ideally this should permit the programmer to understand and modify the transformed CPS code after it has been deployed.

This paper advocates an approach that is aware of the dependencies between the different instructions in the code in order to exploit the available concurrency within the transformed code. Continuations only encapsulate the instructions on which it depends and that are strictly necessary. This results in a transformed code that better respects the behaviour of the original code in terms of performance, fault-tolerance and responsiveness.

In contrast with other approaches we do not add explicit language constructs to mark asynchronous requests that need to be transformed. This means it is up to the transformation tool to identify potential asynchronous function calls and transform the code into a CPS where necessary. However, this also implies the programmer no longer has control over how the transformation tool transforms the code and where a CPS is applied. In order to improve this we have added a number of optional explicit language constructs that allow the programmer to influence the transformation process. These are discussed in more detail in section 3.3.

This implicit transformation has as an advantage that the code is more configurable. This approach can be applied to calls between server and client tier, which are asynchronous by default. Take for instance code with local calls which the transformation converts to remote procedure calls with a callback function. When the developer decides that a certain part of the code should be on the server instead of the client, the code can be copied as is, without the need to remove explicit continuation marks.

Eventually our aim is to integrate a debugger with our transformation tool in such a way that any runtime state can be reported on the original code while the debugger is executing the transformed code. Additionally, our approach also attempts to maintain as much as possible the structure and variable names of the original program in order to improve the maintainability of the transformed code.

3. Overview of the Approach

Our program transformation tool performs its transformation in two steps. Starting from the synchronous, direct-style program code, the transformation tool first constructs a program dependency graph (PDG). This graph is then used together with the abstract syntax tree in a second step as input for the actual CPS transformation. We now describe the accepted input language and the constructed program dependency graph that act as input for the transformation.

3.1 Input A : Synchronous Code

Our approach is targeted towards the web domain. Because of this our framework accepts JavaScript as a target language. However, our approach is general enough to be applied to any higher-order, general-purpose language.

Our framework accepts any valid JavaScript program as input. Additionally, any explicit callback parameters of asynchronous function calls can be omitted. In that case, the asynchronous function call can be used as an expression in a synchronous context where an immediate result is required. For example, Listing 4 depicts a server-side JavaScript program that uses an external API to collect information about boardgames to display on a webpage. In this example two requests are issued to the external API. First the latest and 'hottest' boardgames are fetched. Secondly, several requests are issued to fetch the latest games friends have been playing[2]. Please note that the function arguments on line 9 and 15 are not callback functions that wait for the call to return but are actually event listeners.

```
1  function options(path) {  //e2 f1 in
2    return {                // f1 out
3      hostname: 'bgg-json.azurewebsites.net', //s1
4      path: path,    //s2
5      method: 'GET' //s3
6    }
7  }
8  var latest = https.get(options('/hot')); //s4 c1
9  latest.on('data', function (d) { //s5 c2
10   console.log(d);
11 })
12 var friends = ['lphilips', 'jdekoste', 'wdmeuter',
       'cderoove'];  //s6
13 friends.forEach(function (friend) {
14   var played = https.get(options('/plays/' +
         friend));//s7 c3
15   played.on('data', function (d) { //s8 c4
16     console.log(d);
17   })
18 })
```

Listing 4. Sample program in direct-style JavaScript with mapping to nodes

[2] We collect this information from `boardgamegeek.com` using a Boardgamegeek API `http://bgg-json.azurewebsites.net/`

On Line 1 we first define a function that constructs the request object that needs to be included when using the HTTPS module[3]. The path parameter can either be `'/hot'` to retrieve the most popular games or `'/plays/uname'` to obtain the latest games the user with username `uname` has played.

On line 8 the callback parameter of the request was omitted and the request is used as a synchronous function call of which the result is stored in the `latest` variable. To simplify the example, we have omitted the client-side code and only print the results to the console. Similarly, on lines 13 we iterate over a list of friends and on line 14 request the latest games played by each of those friends. This request can be used as a synchronous expression for which the result is stored in the `played` variable.

Any asynchronous function call can just omit the callback parameter and be used in a synchronous context. This is in contrast to other existing approaches, where the developer is required to manually mark asynchronous function calls or definitions.

3.2 Input B : the Program Dependence Graph

The second input to our transformation is information about the dependencies between the different parts of the program. This information is carried by a program dependence graph that contains all control and data flow dependencies.

More concretely, a PDG is a directed graph of which the nodes correspond to instructions and branch predicates, and of which the edges correspond to data and control dependences [14, 27]. An instruction is data dependent on another if values flow from the latter to the former. An instruction is control dependent on a branch predicate if the outcome of the latter determines whether the former will be executed. For instance, the instructions in the body of a function are control dependent on the *entry node* of that function. A program dependence graph also incorporates parameter binding edges from concrete arguments of a call to the formal parameters of the function definition.

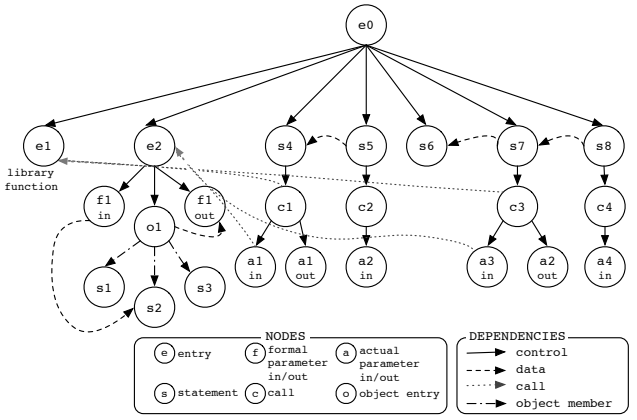

Figure 1. PDG for the code given in Listing 4.

Figure 1 depicts the PDG for the code given in Listing 4. Not all details are present, but all necessary parts to explain our approach are. The root node `e0` of the program has control dependencies to seven PDG nodes: two function definitions and five statements, which are all control dependent on the root node. The library function `https.get` is represented as an entry node (`e1`), without all details (parameters, body). The definition of the function `options` adds a new entry node (`e2`), which has one `return` statement as its body. This return statement creates an object literal, resulting in an object entry node. This object literal has three object members: `hostname`, `path` and `method`. The `option` function has one parameter `path`, represented in the PDG as a formal parameter node

[3] `https://nodejs.org/api/https.html`

$\texttt{f1}_{in}$. This `path` parameter is sent along to the actual request to obtain a list of boardgames, rendering statement `s2` data dependent on the formal parameter node. The function has a concrete return value, the object literal which contains all the information for the request, so a formal out parameter is added to the function ($\texttt{f1}_{out}$).

The two variable declaration statements for `latest` and `played` each contain a call to the `https.get` library function. This means that these statements have a *call node* that is call dependent on the entry node of the function that is called[4]. A call node has an actual parameter for each formal parameter of the invoked function. We omitted the parameter-binding edges for presentation purposes, these would bind the corresponding actual in nodes with the formal in parameters of the called function and likewise the formal and actual out parameters. These parameter-binding edges produce a transitive flow from the call node to the function nodes and back.

In the code example we see how the results from the two requests are being used later on to print them. This results in data dependencies from the declarations of the result variables to the statements where they are being used; from `s5` to `s4` and from `s8` to `s7`. The first one is because the statement on line 9 uses the variable `latest` declared on line 8. Variable `played` gets used on line 13, hence creating a data dependency from that statement node `s8` to the statement node that corresponds to the declaration of `played`, `s7`.

PDGs are primarily used for program slicing [25, 27]. Informally, a program slice is a subset of the program that has a direct or indirect effect on the values computed at a certain location. Accordingly, a slice is computed with respect to a slicing criterion, usually represented by the combination of a line number and a set of variables. Although many variations of the slicing algorithm exist, they can be roughly categorised into *forward* and *backward* slicing algorithms. In short, backward slicing looks for the statements that influence the slicing criterion, while forward slicing looks for all statements that are influenced by the slicing criterion. The slicing algorithm works as a graph traversal algorithm that returns a set of nodes that belong to the program slice. Applications of program slicing are to be found in debugging, software maintenance, etc. We do not slice the program, meaning we don't exclude parts of the code, but we use the dependency information of the PDG to guide our CPS transformation.

3.3 Transformation based on the Program Dependence Graph

The actual transformation rewrites synchronous calls to asynchronous calls with a continuation callback argument.

The process recursively walks the PDG and looks for call nodes and function declarations (thus entry nodes in the PDG). Call nodes need to be transformed to a call with the original arguments and an additional one for the callback that holds the continuation for that call. Functions on the other hand take an extra parameter for the callback and instead of returning a value, the result of that function is given back to the caller by calling the callback function with the return value. The algorithm for this transformation is given in pseudocode below:

```
1  Make a set of each node in the PDG, nodes
2  Traverse PDG, for each node n do the following:
3    If n is a member of nodes:
4      If n is a Call Node:
5        Let p be the parent node of n,
         cps the transformed call to cps-form.
6        If parent has data dependencies:
7          Let slice be the forward slice of p.
8          For each node sn in slice:
9            Add transform(sn) to body of callback of cps
10       If n has blocking annotation
         or called function has side-effects:
11         For each node pn of the remainder of the program:
12           Add transform(pn) to body of callback of cps
13     If n is an Entry Node for a Function Declaration:
14       Add additional callback parameter to n
15       Transform the return statements
16     Else: do nothing
17     Remove n (and its children) from nodes
```

While the algorithm is traversing the PDG, we keep track of a set of nodes, initially set to all the nodes in the PDG. This way we avoid non-termination when cycles are present in the PDG. Because some transformations that are applied on the PDG move nodes, we also use this set to track whether the current PDG node has already been transformed or not. This is in contrast with the general approach for CPS transformations, as given in section 1. Both approaches traverse the PDG, but the original one takes the ordering as determined by the syntax tree (step **??** of the algorithm described in section 1). Using the dependency information we can establish an ordering that does not completely abide by the PDG, but guarantees that continuations that belong together are nested.

The most interesting part of the algorithm is the transformation for call nodes. It is in this transformation that we construct the continuation of that statement. In order to do so, we look at the parent statement of that call node. If the call happens in another statement, e.g., a variable declaration or another call, we examine whether other nodes are dependent on that node. For instance, in the expression `var a = foo(42)`, the parent of the call node `foo(42)` is the declaration node of `a`. If the call node has such a parent (meaning that the result of the call is being used (later on)), we perform a *forward slice* on the corresponding node in the program dependence graph. A forward slice of a program is the collection of statements that may be affected by the value of variables at that program point.

Should it be that the analysis that serves as the basis for the construction of the PDG is imprecise, we could end up with a forward slice that is not minimal. This implies that the continuation is not minimal and worst case, we end up with the remainder of the program at that point as its continuation. We discuss this in more detail in section 3.5.

In the following code example, the forward slice for the program point at line 1, results in the set of the statements on lines 1, 3 and 4.

```
1 var a = foo(42);  // Slicing criterion
2 var b = foo(43);
3 var c = a + 2;  // Part of slice
4 var d = c + 3;  // Part of slice
```

Listing 5. Example code for a forward slice

Based on this forward slice, we transform each instruction that is part of the slice and put it in the continuation callback of that function (step 7-9). These nodes are thus removed from the set of nodes that still need to be transformed, otherwise they would end up twice or more in the transformed program. The call node itself,

[4] Actually, another call is made in those statements, namely to the `options` function. This would mean call nodes would be added beneath the actual in parameters. To limit the size of the graph, we omitted these call nodes and its actual parameters

together with its children (actual parameters) are removed from the *nodes* set (step 17).

However, when a call is made to a function that has side-effects, the forward slice should include all program statements that follow. This implies that no forward slice is computed, but that the entire remainder of the program at that point is to become the continuation of the current call. This also applies to the case where the call is annotated to be `@blocking`. To decide whether a function has side-effects, we use a straightforward, but conservative AST analysis. A full-blown purity analysis for JavaScript such as [20] could be used to improve precision.

The `@blocking` annotation can be used on one single call, meaning that the remainder of the program should wait or it can be attached to a block statement. This allows programmers to explicitly delimit a continuation; the continuation is captured until the end of the annotated block.

The transformation process for a function declaration involves adding an additional parameter for the callback function. On top of that, the return statements are transformed to a call to the callback parameter (step 13-15).

Other nodes, are not transformed (step 16) but they are removed from the set of nodes that still need to be transformed as well (step 17).

Note that in the listed transformation algorithm, each call node and function declaration node is transformed to CPS. However, in practice, programmers want to perform a *selective* CPS transformation [4], where not every call receives a continuation function. The algorithm presented here can be easily extended to incorporate criteria for deciding whether or not a transformation should take place. We elaborate more on this in Section 3.5 and 5.

3.4 Asynchronous Code

All calls and function declarations from the original program have been rewritten by the transformation process. Our transformation results in the following code for the program listed in Section 3.1:

```
1  function options(path) {
2    return {
3      hostname: 'bgg-json.azurewebsites.net',
4      path: path,
5      method: 'GET'
6    };
7  }
8
9  https.get(options('/hot', function (res) {
10   var latest = res;
11   latest.on('data', function (d) {
12     console.log(d);
13   });
14 });
15 var friends = ['lphilips', 'jdekoste', 'wdmeuter',
       'cderoove' ];
16 friends.forEach(function (friend) {
17   var played;
18   https.get(options('/plays/'+friend), function (
         res) {
19     var played = res;
20     played.on('data', function (d) {
21       console.log(d);
22     });
23   });
24 });
```

Listing 6. Result of transformation process for Listing 4

The transformation process has reacted to the data dependencies from the program dependence graph by pulling the statement that uses the result of the first request (line 9) into the callback that declares `latest`. As a result, the correct dependencies are in scope of one another. The callback continuation for the first call to the `https.get` has been limited to the necessary statements only. The other call to `https.get` is not included in the first callback continuation, but is executed concurrently with the first call.

3.5 Implementation

We have implemented the CPS transformation based on dependence graphs for JavaScript. Our implementation is available online as an interactive tool STIP [5] that can be used to construct a program dependence graph, calculate (backward) slices, and transform a direct-style program to a continuation-passing program. Our implementation can easily be extended to support a refactoring to promises instead of callbacks, based on the dependence analysis. Another option would be to use a refactoring tool that is able to transform callback-based JavaScript to promise-based JavaScript, such as PromisesLand[6]. We now give a more detailed overview of the concrete implementation according to the steps described in section 3.

Input A: Synchronous Code Our implementation is tailored to JavaScript programs. The input code is a direct-style JavaScript program, where some calls have possibly been annotated with the `@blocking` annotation. These annotations reside inside comments, such that other tools, like IDE's or refactoring tools, ignore them. They serve however as a seed for our CPS transformation, together with the Program Dependence Graph.

Input B: the Program Dependence Graph To construct a program dependence graph of the direct-style JavaScript code, we perform a recursive walk on the AST of that program. The control flow dependencies can be calculated from the AST. For example, every expression inside the body of a function, results in a control flow dependency from the entry node for that function to the statement node representing that expression.

Data and call dependencies cannot be calculated from the AST. We rely on an abstract interpretation [2] of JavaScript to uncover these data dependencies. Our current implementation uses the JIPDA abstract interpretation [19][7], but can switch to other abstract interpretation frameworks, such as TAJS [11]. We therefore owe our support for a fairly representative subset of JavaScript to JIPDA —including several features that are difficult to analyse statically such as higher-order functions with side-effects and prototype chains. Imprecise results from the abstract interpretation are reflected in the program dependence graph by multiple call or data dependencies from one node to all possible referees. This produces overfitting continuations, that could end up to contain the remainder of the program. In this worst case scenario we obtain the same results as classic CPS transformations without support for delimiters. For example, when the analysis is imprecise about the declaration statement of a reference, it could be that the graph has two data edges from the reference statement to two different nodes, each declaring the same variable. If both declaration statements contain a call that is transformed to its CPS equivalent, the reference statement will end up in the continuation of both calls. This could possibly lead to wrong behavior of the program. It is therefore that we warn the programmer when either no declaration node can be found or more than one declaration node is returned by the analysis. The programmer should act upon these warnings and give the analysis more hints.

Delimited CPS transformation Section 3.3 described how the CPS transformation walks the PDG, while transforming the corresponding AST nodes for call and function expressions.

The transformation uses the dependencies in the PDG for two purposes: control dependencies drive the recursive walk of the

[5] http://bit.ly/stipjs

[6] `http://salt.ece.ubc.ca/software/promisland/`

[7] `https://github.com/jensnicolay/jipda`

PDG, while the data dependencies guide the computation of the limitations of each continuation.

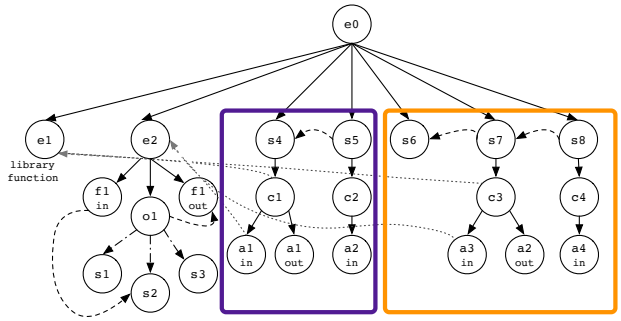

Figure 2. PDG for Listing 4, limits of computations highlighted.

Figure 2 groups the data dependent statements inside rectangles. Statement nodes `s4` and `s5` represent the following code (taken from listing 4):

```
1 var latest = https.get(options('/hot');
2 latest.on('data', function (d) {
3   console.log(d);
4 })
```

Listing 7. Sample program in synchronous JavaScript.

Node `s4` corresponds to the declaration of `latest`, while the installation of the event listener on this variable corresponds to `s5`. Therefore, there is a data dependency from node `s5` to `s4`, because it uses the declared variable of `s4`. As can be seen in the graph, a call to the library function `https.get` is made as well. When transforming this direct-style call into its continuation-passing variant, we need to take into account the data dependencies on node `s4`, because that call is made inside that expression.

The other call that gets transformed into CPS is `c3`, which is a call that is made in the expression that corresponds to node `s7` in the PDG. On its turn, `s8` is data dependent on `s7`, and therefore must be part of the continuation of `c3`.

We compute the borders of a continuation based on the dependencies in the dependence graph, resulting in delimited continuations. These dependencies are ignored when a call or a block statement is annotated to be blocking. Listing 8 shows how the `sleep` function (line 2) is called before executing the request to the board game service. This call expression has been adorned with a `@blocking` annotation, to indicate that the remainder of the program should wait until the `sleep` function has finished its execution. No delimited continuation is computed as a result, but the entire remainder of the program becomes the continuation of that call.

```
1  /* @blocking */
2  sleep(5000);
3  var latest = https.get(options('/hot');
4  // Process latest boardgames
5  var friends = ['lphilips', 'jdekoste', 'cderoove',
       'wdmeuter' ];
6  friends.forEach(function (friend) {
7    var played = https.get(options('/plays/' +
         friend));
8    // Process boardgames friend played
9    })
10 })
```

Listing 8. Sample program with an explicit blocking point.

If there would be no means to express that the call to `sleep` should be blocking, we would end up with the code given in Listing 9. On line 1 the call to `sleep` gets an extra callback parameter, which has an empty body. The call to `https.get` on line 2 is thus executed concurrently with the `sleep` call and not in the continuation of `sleep`.

```
1 sleep(5000, function (err) {});
2 https.get(option('/hot'), function (err, res) {
3   var latest = res;
4   // remainder of computations on latest
5 });
6 // remainder program
```

Listing 9. Faulty transformation for the code in Listing 8

However, the call to sleep has been explicitly annotated to be blocking in listing 8 thus the whole remainder of the program should wait until this call has finished, as demonstrated in the resulting code in Listing 10. We see how both calls to the `https.get` function are now captured inside the continuation of the call to `sleep`, but both calls are still executed concurrently instead of being sequentialised.

```
1  var latest;
2  var friends;
3  sleep(5000, function () {
4    https.get(options('/hot', function (res) {
5      latest = res;
6      // Process latest games
7    });
8    friends = ['lphilips', 'jdekoste', 'wdmeuter', '
         cderoove' ];
9    friends.forEach(function (friend) {
10     var played;
11     https.get(options('/plays/'+friend), function
           (res) {
12       played = res;
13       // Process played games
14     });
15   });
16 });
```

Listing 10. Result of transformation process for Listing 8

Selective CPS transformation The proposed approach transforms every function declaration and corresponding call to its continuation-style equivalent. In practice, however, not every function should be transformed. Current selective CPS transformations use a strictness or effect analysis to determine whether a function should receive an extra continuation parameter. We incorporated part of this selectiveness in the described approach: when a function has side-effects, it should be treated the same as if every call to it is annotated with the blocking annotation. In other scenarios extra criteria could be added to decide whether a call and function should be transformed. For instance, when a library of JavaScript functions always takes a callback parameter by design then every call to such a function should be transformed. The same goes for calls that go from client to server or vice versa, which are non-blocking in nature most of the time. In fact, our selective CPS transformation is an important cornerstone of our approach to tier splitting [22].

4. Evaluation

4.1 Focused Qualitative Evaluation

For the qualitative evaluation we start from an implementation of our motivating example (see Section 2) and convert it to a valid STIP implementation given in Listing 11. For simplicity and space, we omitted the function declaration of `getTemperature(city)`. There are two main advantages of the STIP code over the CONTINUATION.JS code. First of all, in contrast to CONTINUATION.JS, STIP no additional keywords are required because asynchronous function calls are **implicitly identified**. Secondly, STIP enables the use of regular try/catch statements for the handling of exceptions.

```
1  var temp = getTemperature('Brussels');
2  console.log(temp);
3  http.createServer(function (req, res)
4    try {
5      var data = fs.readFile('welcome.txt');
6      res.write(data.toString());
7    }
8    catch (error) {
9      alert(error);
10   }
11 }).listen(8080, 'localhost');
```

Listing 11. Code from Listing 1 as input for the transformation

The code generated by our transformation tool can be seen in Listing 12. There are a number of qualitative advantages of our tool over traditional transformation tools. First of all, our tool correctly identifies that the call to `getTemperature` is independent of the call to `createServer` and does not nest the call to the latter in the continuation of the former. This is in contrast with transformation tools such as CONTINUATION.JS that will conservatively nest every asynchronous call in the continuation of the previous call. Unwrapping independent calls increases the exploited **concurrency** and has advantages in terms of performance, fault-tolerance and responsiveness.

Our transformation tool also preserves the variable names of the original cade, therefore improving the readability and **maintainability** of the transformed code. In this case the `data` variable is used for storing the file contents in both the original cade as well as the transformed code. Please note that our tool also hoists variables, which is a common technique in Javascript engines to separate the declaration (line 6) and assignment (line 8)) of variables.

```
1  getTemperature('Brussels', function (err0, res0) {

2    console.log(res0);
3  });
4  http.createServer(function (req, res) {
5    try {
6      var data;
7      fs.readFile('welcome.txt', function (err1,
           res1) {
8        data = res1;
9        try {
10         if (err1)
11           throw err1;
12         res.write(data.toString());
13       } catch (error) {
14         alert(error);
15       }
16     });
17   }
18   catch (error) {
19     alert(error);
20   }
21 }).listen(8080, 'localhost');
```

Listing 12. Transformed Code from Listing 11

4.2 Larger-scale Quantitative Evaluation

To further validate our transformation tool we have conducted an empirical study to evaluate the effectiveness of our CPS transformation. For this empirical study we chose a number of representative programs from GitHub that exhibit a lot of callbacks. Table 1 gives an overview of the selected GitHub projects. For validating the different applications we based our criteria on an existing empirical study on the use of callbacks in JavaScript programs [9]. Table 2 summarises our results for the selected programs. We list for each program the number of calls with a callback argument, the ratio of these calls to all calls in the program and the number of defined functions that accept a callback parameter over all defined functions in the program. We categorised the calls to library functions that accept a callback into four categories: DOM calls to manipulate an HTML page, network calls like calls to a remote service, calls to timer functions like `sleep` and I/O calls for e.g. file manipulation. The callbacks functions are analysed as well on the number of statements in their bodies and the number of statements that are dependent on the result of the call (thus on the result parameter of the callback function).

In a next step each selected program was rewritten in a synchronous style for both CONTINUATION.JS as well as STIP. This involved removing all callback functions and adding transformation tool specific keywords where necessary. For the CONTINUATION.JS this resulted in replacing every callback function of the form `function (err, res) body` with the `cont(err, res)` expression. For STIP we could simply omit the callback function and insert an `@blocking` annotation where necessary. Afterwards, the code generated from this rewritten programs by both tools was analysed. The results of this analysis can be found in Table 3.

The nesting level is a measure for the minimum and maximum amount of nested callbacks. We confirmed that the original program only contains nested callbacks in the case that these callbacks depend on each other. As we can see from the result, CONTINUATION.JS is unable to correctly identify which calls are independent and often unnecessarily nests the different callbacks in the resulting code after transformation. Conversely, STIP does correctly recognise each of the dependent and independent calls resulting in a nesting level that is similar to the original program. This is off course the best case scenario in which the precision of our analysis is good enough to correctly identify each of the independent calls. While this was the case for each of the analysed programs, it is possible for our analysis to lose precision for some other cases. In such programs our tool will incorrectly assume that two asynchronous calls are dependent and will over synchronise both calls by nesting the resulting callbacks. Thus resulting in a transformation similar to the one of CONTINUATION.JS. In the case where every asynchronous call depends on its predecessor (e.g. as in program 9) both STIP as well as CONTINUATION.JS result in a transformation with the same nesting level.

Not only other asynchronous function calls but any other statements should be considered while transforming the code. If a statement is independent of the result of the callback it should not be included in that callback while transforming the code. As can be seen from the results, CONTINUATION.JS tool locates more statements inside the generated callback functions, as is the case for seven of the examples. This is a consequence of the brute-force transformation that takes the remainder of the program as the continuation.

We also listed the number of continuation specific keywords that are used on the code before the transformation. For the CONTINUATION.JS tool these are `cont` to replace a callback argument of a call. For our approach this is the `@blocking` annotation that is used to indicate a call should be blocking, even when no statements are data dependent on its result. From the table it is clear that our approach does not require the programmer to use dedicated keywords at every asynchronous call, but merely where it should be guaranteed or is unclear. The other approach requires every asynchronous function and call to use these specific keywords.

5. Related Work

Delimited CPS transformations CPS transformation techniques have been around for quite some time now and different flavours thus exist. Delimited CPS transformations focus on the extent of a continuation and often introduce control-operators to mark the continuation's reach.

Table 1. Selected JavaScript programs

Nr.	Program	LoC	URL
1	How Callbacks Work	35	`https://github.com/Runnable/How-Callbacks-Work-Example-App`
2	Callback Example 2	13	`https://github.com/acveer/callback`
3	Callback (Parallel)	28	`https://github.com/dead-horse/callback_hell`
4	Callback Hell	54	`https://github.com/danielrohers/callback-hell-sky`
5	Closures	65	`https://github.com/timestep/javascript-closures-callbacks`
6	Callback Apply	29	`https://github.com/acveer/callback`
7	Games Node	111	`https://github.com/abhidevmumbai/gamesNode`
8	Weather Service	146	`https://github.com/praveen16/weatherApp`
9	Open Marriage (libraries)	519	`https://github.com/ericf/open-marriage`

Table 2. Properties of selected JavaScript programs

Nr.	Nr. of cb	Cb calls	Func with cb	Type cb				Nesting level		Nr. of stms in cb			Data dep. stms
				DOM	network	timer	I/O	min	max	min	max	avg	avg
1	2	22%	0%	0	1	0	1	0	0	2	3	2,5	1
2	2	33%	50%	0	0	0	0	0	0	1	1	1	1
3	4	40%	100%	0	3	0	0	2	2	2	2	2	1,7
4	10	90%	100%	0	0	0	0	9	9	1	1	1	1
5	6	64%	78%	0	0	0	0	1	3	2	4	2,3	1,8
6	2	25%	100%	0	0	0	0	0	0	1	1	1	1
7	14	36%	0%	0	0	0	14	1	1	1	2	1,1	0,9
8	10	23%	50%	0	1	1	0	1	1	1	2	1,1	1,25
9	21	40%	41%	4	4	0	11	0	1	1	2	1,7	1,4

Table 3. Results for the selected JavaScript Programs

Nr.	Nesting level				Nr. of stms in cb						Nr. of keywords	
	min		max		min		max		avg			
	STIP	cont.js	STIP	cont.js	STIP	cont.js	STIP	cont.js	STIP	cont.js	STIP	cont.js
1	0	1	0	1	2	5	3	5	2,5	5	1	2
2	0	1	0	1	2	2	2	3	2	2,5	0	5
3	2	2	2	2	2	4	2	4	2	4	0	3
4	9	9	9	9	1	2	1	2	1	2	0	9
5	1	1	3	5	3	3	5	5	3,3	3,3	0	6
6	0	1	0	1	2	2	2	3	2	2,5	0	2
7	1	2	1	3	1	1	2	3	1,8	2,8	0	14
8	0	1	1	1	1	1	2	4	1,6	2,1	0	10
9	1	1	1	1	1	1	4	6	2,3	4,1	0	20

Scala has a complete package for (delimited) continuations[8]. It offers the control operators `shift` and `reset` to capture and delimit a continuation respectively. Functions should be annotated with the `@cps` annotation to hint the transformation that the function returns a special value, called a `Unit`.

The CPS facilities provided by Scala are exploited in a DSL specialised in rich web applications [13]. The DSL generates JavaScript code and offers an alternative for the callback-driven programming style that is forced upon the programmer when writing asynchronous JavaScript programs. Therefore they exploit the CPS annotations provided by the host language, Scala together with specialised declarations and control abstractions, like `suspendableWhile`.

All of these approaches use explicit control operators to capture and delimit continuations. We do not require such constructs because of our dependency-based analysis. The `@blocking` annotation can be used to tell our transformation that no limits should be calculated, but in general the limits of a continuations are calculated automatically.

Selective CPS transformations We discuss those transformations that are categorised as *selective* CPS transformations; not all functions are transformed to be continuation-passing.

Danvy and Hatcliff perform a selective CPS transformation based on a strictness analysis [4]. It is a selective transformation because certain annotated functions are not transformed. Nielsen defines a selective CPS transformation that preserves part of the program in direct-style [21], based on an effect analysis. Applications in the program are annotated as trivial or non-trivial, based on the control effects of that application. Trivial annotated applications do not get transformed, while non-trivial receive a continuation.

Ley-Wild et al. [17] use annotations to apply an adaptive CPS transformation on self-adjusting programs. It is a selective transformation because certain annotated functions are not transformed.

Our CPS transformation is based on the dependencies between program statements and the programmer can add this selectiveness by implementing a predicate function. This means that our CPS transformation can be used in several domains. Because we already support annotations (`@blocking`), support for selective annotations as in [17] can easily be added to our approach.

Async-based approach Some programming languages take a so-called *async-based* approach that support automatic transformation to asynchronous programs.

C# introduces the `async` and `await` keywords [6]. Methods that are indicated to be `async` can use the `await` keyword to mark the point(s) where an asynchronous operation is performed. When such a point is encountered, the method is suspended and control returns to the caller of that method. The remainder of the method ends up as the continuation of that asynchronous operation. Async methods return a task, a representation of ongoing work. This task contains the state of the asynchronous operation and will contain the result (or exception) when the operation eventually finishes. While the

[8] `http://www.scala-lang.org/api/2.10.1/index.html#scala.util.continuations.package`

Table 4. CPS transformations projects in JavaScript

	Streamline	Continuation	TameJS	StratifiedJS
Replacement cb	_ construct	`cont(params)`	`defer(params)`	`resume`
Cont. granularity	full program	full program	construct	construct
Failure handling	✓	✓	✓	✓
Readable result	×	✓	×	×
Tool support	✓	×	✓	✓

await keyword resembles our `@blocking` annotation, it is always required to be explicitly used.

CLOJURE[9] has the `core.async` library to facilitate asynchronous programming using *channels*. A channel is a queue with consumers and publishers, which take data from and put data in the queue respectively. This can be achieved in a synchronous or an asynchronous way. A `go` block groups asynchronous reads and writes to a channel. CORE.ASYNC transforms the body of each go block in the program to parallel running state machines and transforms the direct-style operations to their asynchronous form.

CPS transformations for JavaScript We already mentioned tools that transform a synchronous JavaScript program to its asynchronous variant. Table 4 depicts four of these tools and evaluates them against a set of essential properties.

STREAMLINE.JS[10] works under the assumption that all callbacks are replaced by an underscore. One inconvenience is that asynchronism is *contagious*; functions that use an underscore to replace a callback somewhere in there body should take an extra underscore parameter as well. The generated code is hardly readable, introducing many intermediate values, function wrappers, etc. Failure handling is carried out by the usual `try/catch` construct. Streamlined functions can be called with classic callbacks as well. To achieve a delimited continuation, the programmer thus has to fall back to callback functions. Debugging streamline code can be achieved through JavaScript Source Maps[11].

We already demonstrated how CONTINUATION.JS performs CPS transformations in section 2. It introduces a virtual function `cont` that replaces the callback functions. Every time `cont` is used, the control flow is suspended until the result of the asynchronous call has returned. Just like STREAMLINE.JS, programmers can still use `try/catch`, but then the `obtain` construct must be used instead of `cont`. The `parallel` construct executes asynchronous functions concurrently, and only after each function call has finished, the consequent code is executed. This construct is thus a way to explicitly mark asynchronous calls in the code that should be executed concurrently. Out of all four transformation tools we discuss here, CONTINUATION.JS is the only one that maintains variable names and produces human readable output.

TAMEJS[24] works on a dialect of JavaScript, introducing several primitives like `await`, that takes a block of code. In such a block, every callback must be replaced by the `defer` construct. Consecutive code must wait for the `await` block to finish, i.e., every `defer` in that block is fulfilled. Every `defer` in the block is executed in parallel. The programmer has thus more control over the execution order of asynchronous tasks. Those that should be executed in parallel must be grouped in an `await` block. The language has tame-aware stack traces, enabling debugging support.

STRATIFIEDJS[12] gets rid of the asynchronous spaghetti in JavaScript by extending it with syntactic additions. Identical to TAMEJS special constructs are introduced to group statements that execute asynchronously and can only be resumed explicitly. The `waitfor` construct and its variants takes a suspending code block that blocks the execution of consecutive code. The `resume` function can explicitly resume consecutive code. It is therefore that `resume` can be passed along instead of a callback, such that the control flow will resume once the callback is called with the result.

Most of these approaches have explicit constructs to express that certain functions call should be executed simultaneously and execution can only progress when all (or one) of these calls has finished. This is a way to prevent that these calls are being sequentialised in the transformed code. However, all of the consecutive code ends up as the continuation of these parallel calls, with no way to delimit the continuation.

6. Conclusion

Writing code in a direct synchronous style has a number of advantages over an asynchronous style using callbacks. For one, it avoids the need for deep nesting of callback handlers colloquially knows as the "callback hell". Today, there already exist a number of transformation tools that enable code written in a synchronous style to be automatically transformed into an asynchronous CPS. The transformation tool presented in this paper improves upon these existing tools on three levels. Firstly, our tool avoids conservative nesting of independent callbacks. This increases the exploited concurrency which in turn improves performance, fault-tolerance and responsiveness of the resulting application. Secondly, our tool does not require explicit annotations in the code to distinguish asynchronous function calls from synchronous function calls. Rather, it employs an analysis based on program dependence graphs as a technology to enable automating selective, delimited CPS transformations. Thirdly, our tool produces a transformation of the code that preserves as much as possible the structure and variable names of the original program. This facilitates understanding, testing and debugging of the transformed code.

To validate our approach we analysed a number of existing JavaScript projects that exhibit a lot of callbacks. We first manually transformed them back into a synchronous style using both CONTINUATION.JS as well as our tool STIP. We then compare the resulting transformations for both tools and show that our approach produces code that improves upon the exploited concurrency and the maintainability compared to other tools. We show that our tool can correctly identify independent callbacks for the investigated programs and only nests callbacks when it is strictly necessary.

Acknowledgments

Laure Philips is supported by a doctoral scholarship granted by the Agency for Innovation by Science and Technology in Flanders, Belgium (IWT). This work has been supported, in part, by the SBO-project Tearless funded by the same agency.

[9] `https://github.com/clojure/core.async`

[10] `https://github.com/Sage/streamlinejs`

[11] `http://www.html5rocks.com/en/tutorials/developertools/sourcemaps/`

[12] `http://onilabs.com/stratifiedjs`

References

[1] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 1992.

[2] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[3] Oliver Danvy. Three steps for the CPS transformation (detailed abstract). Technical Report CIS-92-02, Kansas State University, February 1992.

[4] Olivier Danvy and John Hatcliff. CPS-transformation After Strictness Analysis. *ACM Lett. Program. Lang. Syst.*, 1(3):195–212, September 1992.

[5] Olivier Danvy, Jung-taek Kim, O. Danvy, Jung-taek Kim, Kwangkeun Yi, and Kwangkeun Yi. Assessing the Overhead of ML Exceptions by Selective CPS Transformation. In *In Proceedings of the 1998 ACM SIGPLAN Workshop on ML*, pages 103–114, 1998.

[6] Alex Davies. *Async in Csharp 5.0 - Unleash the Power of Async.* O'Reilly, 2012.

[7] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM.

[8] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding Delimited and Composable Control to a Production Programming Environment. *SIGPLAN Not.*, 42(9):165–176, October 2007.

[9] Keheliya Gallaba, Ali Mesbah, and Ivan Beschastnikh. Don't Call Us, We'll Call You: Characterizing Callbacks in JavaScript. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2015, Beijing, China, October 22-23, 2015*, pages 247–256, 2015.

[10] Robert Hieb, R.Kent Dybvig, and III Anderson, ClaudeW. Subcontinuations. *LISP and Symbolic Computation*, 7(1):83–109, 1994.

[11] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium (SAS)*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

[12] Yukiyoshi Kameyama and Masahito Hasegawa. A Sound and Complete Axiomatization of Delimited Continuations. In *In Proc. of 8th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'03*, pages 177–188. ACM Press, 2003.

[13] Grzegorz Kossakowski, Nada Amin, Tiark Rompf, and Martin Odersky. Javascript as an Embedded DSL. In James Noble, editor, *ECOOP 2012*, volume 7313 of *Lecture Notes in Computer Science*, pages 409–434. Springer Berlin Heidelberg, 2012.

[14] Jens Krinke. Program slicing. In S K Chang, editor, *Handbook of Software Engineering and Knowledge Engineering 3*. World Scientific Publishing, 2004.

[15] Shriram Krishnamurthi, Peter Walton Hopkins, Jay A. McCarthy, Paul T. Graunke, Greg Pettyjohn, and Matthias Felleisen. Implementation and use of the PLT scheme Web server. *Higher-Order and Symbolic Computation*, 20(4):431–460, 2007.

[16] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM.

[17] Ruy Ley-Wild, Matthew Fluet, and Umut A. Acar. Compiling Self-adjusting Programs with Continuations. *SIGPLAN Not.*, 43(9):321–334, September 2008.

[18] James S. Miller. *MULTISCHEME : a parallel processing system based on MIT scheme*. PhD thesis, 1987. PHD.

[19] Jens Nicolay, Carlos Noguera, Coen De Roover, and Wolfgang De Meuter. Determining Coupling In JavaScript Using Object Type Inference. In *SCAM13*, 2013.

[20] Jens Nicolay, Carlos Noguera, Coen De Roover, and Wolfgang De Meuter. Detecting Function Purity in JavaScript. In *Proceedings of the 15th International Working Conference on Source Code Analysis and Manipulation (SCAM15)*, 2015.

[21] Lasse R. Nielsen and BRICS. A Selective {CPS} Transformation. *Electronic Notes in Theoretical Computer Science*, 45(0):311 – 331, 2001. MFPS, Seventeenth Conference on the Mathematical Foundations of Programming Semantics.

[22] Laure Philips, Coen De Roover, Tom Van Cutsem, and Wolfgang De Meuter. Towards Tierless Web Development Without Tierless Languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 69–81, New York, NY, USA, 2014. ACM.

[23] Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing First-class Polymorphic Delimited Continuations by a Type-directed Selective CPS-transform. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 317–328, New York, NY, USA, 2009. ACM.

[24] Maciej Swiech and Peter Dinda. Making JavaScript better by making it even slower. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2013 IEEE 21st International Symposium on*, pages 70–79. IEEE, 2013.

[25] Frank Tip. A Survey of Program Slicing Techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[26] Andrew Tolmach. Debugging Standard ML without Reverse Engineering. In *In Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 1–12. ACM Press, 1990.

[27] Mark Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.