STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LV**, Number 3, 2010

JAVASCRIPT GENERATORS

GHIŢĂ DANIEL-CONSTANTIN

Abstract. The JavaScript language is used for client-side programing in web applications. JavaScript is sent as source code from the web server to the user's browser and executed locally. Because not all current browsers support the latest version of JavaScript, developers are limited to use only the features supported by the lowest common denominator. I propose an approach to solve this problem by using an intermediary compile step which transforms the code written by the user into code supported by all browsers. This allows web developers to use the latest version of JavaScript today, without having to wait until all major browsers implement all the features.

1. Introduction

JavaScript is commonly known as the language of the web. This object-oriented scripting language is implemented in all modern browsers and it is the only cross-platform language available for client-side programming in web application. It is a dialect of the ECMAScript standard [6].

JavaScript is rapidly evolving. Because some implementations are lagging behind, developers cannot take advantage of all the features which have been added to the language. This is further amplified by the presence in the wild of browsers which are not up to date.

We can address this issue using static compiling. Using a compiler we can transform code which is written to use the latest version of JavaScript into code which uses only the basic syntactic constructs and so is (syntactically) compatible with all implementations in the wild. Using a library which will be loaded at run-time the program can add any missing methods to the core objects of the language. We will show how to address each issue individually with examples and offer an implementation of a compiler which integrates all the pieces. The result is a translator which takes as input code written in

Received by the editors: June 10, 2010.

2010 *Mathematics Subject Classification.* 68N20.

1998 *CR Categories and Descriptors.* D.3.4 [**Software**]: PROGRAMMING LANGUAGES – *Processors*.

Key words and phrases. JavaScript, ECMAScript, web applications, compiler, program analysis.

the latest version of JavaScript and outputs code which works on all current JavaScript platforms.

This rest of the paper is structured as follows. Section 2 is structured in two subsections: the first one offers an overview of JavaScript, while the second section takes a look at what benefits compiling has offered other languages. Section 2.3 presents the current status of methods applied for JavaScript source code. Section 3 presents the goal of this paper in more details. Section 4 details each step of the path in achieving said goal, and in the end we give some conclusion that evaluates the results which have been obtained and the roads opened for future work.

2. Background

2.1. **A Brief Overview of JavaScript.**

2.1.1. *Where JavaScript is today.* There are many different implementations of ECMAScript dialects and a few call themselves JavaScript. Every major browser has its own ECMAScript implementation and there are even compilers and interpreters for ECMAScript as a systems language. ECMAScript covers only the core language and many extensions, including "host objects", are left to the implementation:

> ECMAScript as defined here is not intended to be computationally self-sufficient; indeed, there are no provisions in this specification for input of external data or output of computed results. Instead, it is expected that the computational environment of an ECMAScript program will provide not only the objects and other facilities described in this specification but also certain environment-specific *host* objects, whose description and behavior are beyond the scope of this specification[6]

2.1.2. *Basic building blocks.* Often described as "Lisp in C's clothing"[4,5], the JavaScript language is dynamic and weakly typed. It has first-class functions and prototype-based object inheritance. It has closures, which means functions have lexical scope and they can access the variables from the context in which they were created. Functions can be called with any arguments (and any number of arguments). All objects are hash tables and objects cannot be created *ex-nihilo.* Objects and arrays can be written as literals and the language has built-in support for regular expressions. The logical OR (`||`) and logical AND (`&&`) are short-circuiting operators. The ternary operator is similar to the one found in C.

2.2. **Benefits of compiling.** Compiling is the process of transforming code from one computer language into another—called the *target* language. Usually compilers target machine code which can be directly executed by a computer or virtual machine. Historically, compilers have been a major factor in the advance of programming by allowing the same code to be used on different architectures and by performing tedious tasks on the behalf of the programmer, such as checking type correctness, optimizations, relaxing the limitations of the machine (e.g. aligning data at certain memory locations), supporting high-level constructs not available on the underlying hardware, linking (distributing a program across more than one source file) and more. In short, compiling enables easier programming and portability.

2.2.1. *Source-to-source compilers.* While most compilers output machine code or bytecode, *source-to-source compilers* use a high-level language for both the input and the output. Source-to-source compilers can be used, for instance, for adding parallel code annotations (e.g. OpenMP) or for porting code from an old version of the language into a new version.

2.3. **Review of the State of the Art.** In this section we will look at what methods have been previously applied to JavaScript source code before.

2.3.1. *Minification.* Minification [5]is the process in which an input source code is transformed into an output source which, when executed, results in the same operations being performed on the same data. The program which performs this transformation on the code is called a minifier. The goal is usually to obtain a program which works identically to the input code but which has a smaller size. In order to achieve this goal, the minifier employs more than one technique:

- Whitespace removal. Indentation, line feeds and any other whitespace is deleted because it does not affect the semantics of the program.
- Comments are eliminated, for the same reason as whitespace. Usually JavaScript minifiers understand and preserve conditional comments. Also, some minifiers preserve comments which begin with '`!`' or which contain '`@license`' or '`@preserve`'. This is used in order to preserve copyright information.
- Redundant characters with syntactic but not semantic meaning are removed. For instance, many semicolons can be deleted without affecting the instructions because JavaScript allows the programmers to omit semicolons in some places.
- Numbers are written in the shortest form possible. For instance, `1000` could be transformed into `1e3` and `1e1` could be transformed into `10`, each modification saving one character.

- Local variables and function arguments are renamed to use less characters. This can be particularly problematic in dynamic languages which support `eval(...)` such as JavaScript. Because the argument to be evaluated is a string and is usually composed dynamically or received from an external source (e.g. a response from the server), any references to local variables inside the string cannot be updated by the minifier. This means that executing `print(eval("currentIndex + pageSize"));` after minification will result in two undefined references[1].
- Transform conditionals where possible from `if`-statements using short-circuiting operators. A short-circuiting boolean AND operator does not evaluate the second argument if the first is false. Some tools take advantage of JavaScript's boolean operators and transform statements such as `if (a) b();` into `a && b();`.

Minification is usually safe, unless local variables are obfuscated and `eval(...)` is used to reference local variables.

2.3.2. *Packing.* Packing[5] is the process in which the identifiers, keywords and constants are separated from the code and moved into a separate list. Duplicates are discarded and placeholders are used in the code to know where to insert each item. Two strings are obtained: the first contains the code without the removed parts and the second contains all the removed parts with a separator. The final file has the two strings and a few instructions which reassemble the original file (as a string) and feed it to `eval(...)`. Packing is usually used after the source has already been minified.

Tools which implement minification of JavaScript sources include Closure Compiler, JSMin, Microsoft AJAX Minifier, Packer, ShrinkSafe and Yahoo! UI Compressor. Packer is the only one which implements packing.

2.3.3. *Function inlining. Function inlining* or *inline expansion* is the process in which a call to a particular function is replaced with the body of the function. While this increases the size of the code if the function is called from more than one place, it optimizes the execution speed of the program if the function is called often because it eliminates the overhead caused by a function call.

2.3.4. *Aliasing. Aliasing* is the situation in which a data location can be accessed under different names called aliases. In JavaScript aliasing is often used in conjunction with calling the same method on the same object repeatedly.

[1]Local variables and function arguments are not renamed in functions which use a `with` instruction anywhere in the source because declaring variables inside a `with` statement creates properties inside the object instead of local variables.

For instance, creating an alias for the `appendChild` method of the <body> element in an HTML document:

```
function add(newNode) {
    document.body.appendChild(newNode);
}
```

Function inlining is implemented in Closure Compiler.

2.3.5. *Turning Java into JavaScript.* A different approach which has been tried is to give programmers a language with static typing (specifically Java) and then translate that language's syntax into JavaScript constructs. This has the advantage of static type checking and support from the IDEs which target the guest language[2]

The collection of tools which compiles Java into JavaScript is Google Web Toolkit.

2.3.6. *Asynchronous JavaScript.* While JavaScript has features which enable it to work asynchronously (setTimeout, setInterval, asynchronous connections to the server) it is single-threaded and execution itself is synchronous: once a function begins executing, it cannot sleep or pause execution in anyway; it will either terminate or enter an infinite cycle (in which case it will ultimately be killed by the browser). Two techniques have been used to bring this functionality into JavaScript using libraries.

The first method involves using a loader which will scan the JavaScript source before it is loaded into the browser and modify it before it is compiled. This works in all browsers but is very slow.

The second method consists of using the `yield` instruction (which is designed for generators)[7,8]. This takes advantage of a pause-and-resume support built into generator and thus it is more efficient than the first method. The major downside is that it requires JavaScript 1.7 which is currently only available in Mozilla implementations.

2.3.7. *ECMAScript 5 strict subsets.* There has been at least one attempt to implement a strict subset of ES5 strict mode on top of ES3[2]. This has been done in order to limit the flexibility of the language such that modules written using the allowed subset of the language are easier to isolate. This does not actually add anything to language, it just removes some building blocks. Google's Caja project implements two languages: *Cajita* is a strict subset of ES5 strict mode with many features removed (e.g. `eval`, `with`, the `Function` constructor, monkey patching, etc.[3]) and *Valija* is an extension of

[2]JavaScript IDEs are rather poor compared to C++, C# and Java IDEs. For instance, Microsoft Visual Studio 2010 is still unable to understand regular expression literals in JavaScript and tries to auto-complete the text inside and the flags, although the syntax has been in the language since the beginning.

Cajita which adds back some of the elements which have been removed, but with a limited scope (e.g. `eval`).

3. Problem Statement

3.1. **Existing browsers.** JavaScript, as shown in section 1 and section 2, is a fast-evolving language with a huge installed base. Except C and maybe C++, it is a challenge to find one language which can be executed on so many devices. It is more widespread than even Java—every device which can run Java can also execute JavaScript using Rhino (a Java library) and there are many mobile devices which come with a browser but no Java runtime. Not only are there JavaScript shells and cross-platform FastCGI modules(also `#/bin/v8cgi`[1]), but the most popular desktop operating system, Microsoft Windows, comes with a built-in JavaScript environment capable of interacting with the system tools (and no Java runtime unless the user explicitly installs it).

Together with the size of the installed base also comes a disadvantage: fragmentation. Some implementations are lagging behind and, when writing web applications, generic libraries, or other tools, developers are forced to code for the smallest common denominator. This considerably limits the capabilities which can be used and denies the improvements which could be gained: faster development, less error-prone code, better performance and more.

3.2. **Need for compatibility.** While there are many tools which process JavaScript source code, they do not extend it with the latest constructs. The tools either translate other languages to JavaScript constructs[3]. No tool has attempted to implement the new functionality offered by newer versions on top of the common baseline and the topic has not even been formally analyzed before.

Additionally, none of the tools available are written in JavaScript. This is important because JavaScript's `Function` constructor and `eval` function means code will be loaded from external sources, sometimes even on-demand. Unless the tool is implemented in JavaScript, it cannot catch calls to load extra code at runtime and handle the new code.

3.3. **Target and impact.** Today, usually only baseline constructs can be used when developing JavaScript applications[4]. The ones most affected by this are, of course, developers working on JavaScript frameworks. Such frameworks can

[3]If the code must be written in a language other than JavaScript, this usually results in rendering the developer unable to use certain high-level features which are not available in the original language; this may include prototype inheritance, variadic functions, duck typing, iterators, generators, loading code on-demand, etc.

[4]The most notable exception consists of Firefox add-ons. Obviously, they can take advantage of the latest JavaScript implementation which is used by the browser.

also be used outside of the web and into the realm of gadgets, where JavaScript is probably the most popular language[5]. As expected, JavaScript is also used when writing add-ons for browsers[6]. JavaScript is also popular in other circles, as a plugin language for many environments (PDF files, Nokia's Qt framework, OpenOffice, Google Docs, Microsoft Office and more). JavaScript is also used as a general-purpose application programming language (Palm's webOS which is used for phones and other platforms, the GNOME Shell, KDE's kjs, etc).

4. Implementing JavaScript generators in ECMAScript 3

In order to address the issues presented we have built a compiling framework called *Alkaline*[7]. The framework is written in JavaScript in order to provide support for re-entry during runtime when the `Function` constructor or `eval` function are invoked.

4.1. **Architecture overview.**

Alkaline is composed of a few modules and some glue which sends data from one module to the next. Between each stage, an Abstract Syntax Tree (AST) is used as the intermediary representation for the program. Four modules are provided:

- A parsing module which takes JavaScript source code and builds an AST. This module is built using the ANTLR parser generator and includes two distinct parts, a lexer and a parser.
- *Analyzer* annotates the AST (for instance, this resolves variable references).
- *5to3* transforms AST structures from the latest JavaScript specification into structures supported by ECMAScript 3.
- *Printer* transforms the AST back into source code.

4.2. **Generators.**

Generators have been introduced into the JavaScript language from version 1.7[9] and they have been described as "a better way to build Iterators"[10]. There are two main differences between a normal function and a generator:

- generators persist the state of local variables after they *yield* each result
- on each subsequent invocation generators continue execution from the `yield` statement which generated the last item

[5]The following environments use JavaScript: Apple's Dashboard Widgets, Microsoft's Gadgets, Yahoo! Widgets, Google Desktop Gadgets and Serence Klipfolio.

[6]To be more specific, this applies to all the major browsers except Internet Explorer: Firefox, Chrome, Opera, Safari and maybe others.

[7]Alkaline is open source and available at `https://code.google.com/p/alkaline-js/`.

4.2.1. *A simple example.* Listing 1 illustrates a simple generator which yields the next odd number each time it is invoked. The generator is invoked five times in order to return the first five odd numbers.

Listing 1. A simple generator

```
function oddNumber() {
    var i = 0;
    while (true) {
        if (i % 2)
            yield i;
        i++;
    }
}

function printOdd(count) {
    var iter = oddNumber();
    while (count--)
        print(iter.next());
}

printOdd(5);
```

In a way, the generator itself works somewhat similar to a separate thread: once created, it has its own state and even though it never ends, it does not block the main thread.

We can take advantage of the fact that JavaScript offers first-class functions and translate generators into regular functions, and use `yield` statement. In order to demonstrate how the method described affects the code, in Listing 2 we implement the previous simple generator using only ES3 constructs.

The code in Listing 2 takes advantage of the fact that in JavaScript functions are used as constructors and transforms the generator-function `oddNumbers` into both a class and a function which returns a new instance of the class (line 2).

In order to simulate the resume behavior, all blocks of code leading up to the `yield` statement are placed behind `if` statements which test `_continuation` such that when "resuming" execution, they are skipped. All loops and conditionals which gate the execution path toward the `yield` instruction are injected code such that they always take the correct branch when `_continuation` is `true`.

Listing 2. A simple generator modified to use only ECMAScript 3 constructs

```
function oddNumber() {
    if (!(this instanceof oddNumber))
        return new oddNumber();
}

oddNumber.prototype.next = function() {
    if (!this._continuation) {
        this._local_i = 0;
    }
    while (this._continuation || true) {
        if (this._continuation || this._local_i % 2) {
            if (!this._continuation) {
                this._continuation = true;
                return this._local_i;
            } else
                this._continuation = false;
        }
        this._local_i++;
    }
}

function printOdd(count) {
    var iter = oddNumber();
    while (count--)
        print(iter.next());
}

printOdd(5);
```

4.2.2. *Multiple `yield` statements, `StopIteration`.* A natural extension of the simple generator previously presented is the support for multiple `yield` instructions. This increases the flexibility of the method but also complicates the implementation which must support it. Additionally, if after a call to `next()` the generator "terminates" without yielding anything, it automatically throw an error which inherits `StopIteration`. Listing 3 demonstrates all these mechanisms at work. The exception `StopIteration` is easy to solve here by simply adding a `throw` statement at the end of the generator body.

The conditions which gate execution when resuming after a `yield` will need to be updated. Basically, at any point in the source, if we are resuming to a `yield` which is placed after the instruction at the current position we must skip executing the instruction we are considering. If, on the other hand, the `yield` to which we are resuming is inside one of the branches which belongs to the current instruction then we must guide execution toward it. This means that if the instruction gates the third `yield` we must skip it completely when `this._continuation > 3` and we must enter one of the branches when `this._continuation == 3`. Listing 4 applies this approach to the generator from listing 3 mentioned previously. If the resume flag indicates a `yield` which is placed before the current instruction, the code should behave as if the flag is cleared (i.e. normal execution).

Listing 3. A generator with multiple yield statements using only classic constructs (simplified)

```
function multipleGenerator() {
    if (!(this instanceof multipleGenerator))
        return new multipleGenerator();
    this._continuation = 0;
}
multipleGenerator.prototype.next = function() {
    if (this._continuation < 1) {
        this._continuation = 1;
        return "first";
    }
    if (this._continuation < 2) {
        this._continuation = 2;
        return "second";
    }
    if (this._continuation < 3) {
        this._continuation = 3;
        return "third";
    }
    throw new StopIteration();
}

var g = multipleGenerator();
print(g.next()); // prints "first"
print(g.next()); // prints "second"
print(g.next()); // prints "third"
print(g.next()); // throws an error which is an instance of↩
    StopIteration
```

4.2.3. *The `send(...)` method, generator arguments, exceptions.* An important feature of generators is the ability to interact with a generator which has been started. The `yield` statement does more than just pause execution and wait for it to be resumed later. It can return a value and sometimes even throw an exception. By default, it returns undefined.

When `generatorInstance.send(something)` is called, the generator will be resumed (similar to calling `next()`) and the previously paused `yield` instruction will return `something`. This makes it possible to send new information into the generator when it is resuming. By calling `generatorInstance.throw(exception)` the generator will be resumed and the previously suspended `yield` will throw the given exception. Additionally, similar to functions, generators can take arguments. Listing 4 shows a generator for the set of natural numbers. The generator receives an optional starting value—which defaults to 0—and return a new number each time `next()` is invoked. By calling `send(number)` a value can be fed into the generator in order to reset its position to an arbitrary point. This listing demonstrates the use of arguments for generators and the `send(...)` method.

Listing 4. A generator which counts to +Infinity

```
function counter(start) {
    if (!start)
        start = 0;
    while (true) {
        var restart = yield start++;
        if (!isNaN(restart))
            start = restart;
    }
}

var f = counter();
print(f.next()); // prints 0
print(f.next()); // prints 1

f = counter(3);
print(f.next());   // prints 3
print(f.send(8)); // prints 8
print(f.next());   // prints 9
```

It should be noted that both `send(...)` and `throw(...)` wake up the generator. Of course, calling `send(undefined)` is the same as calling `next()`. Calling `send(something)` before the generator yields the first value will throw a TypeError.

Adding support for generator arguments is easy: the names of the arguments are added to the list of local variables and the arguments received in the generator constructor are saved just like local variables. In order to support the `send(...)` method, the generator must take the new value when resuming.

Listing 5 translates the generator `counter` into basic constructs with support for `send(...)`, `throw(...)` and generator arguments.

LISTING 5. A generator which counts to +Infinity using only ES3 constructs

```
function counter(start) {
    if (!(this instanceof counter))
        return new counter(start);
    this._continuation = 0;
    // the unlikely name is used to obtain a reference to the primitive <undefined>
    this._yield_value = this.thisPropertyDoesNotExist;
    this._yield_exception = null;
    this._local_start = start;
}
counter.prototype._next = function() {
    var restart;
    if (!this._local_start)
        this._local_start = 0;
    while (true) {
        return this._local_start++;
        restart = this._yieldReturn();
        this._local_restart = restart;
        if (!isNaN(this._local_restart))
            this._local_start = restart;
    }
}
counter.prototype.next = function() {
    var nextValue = this._next();
    if (this._continuation < 0)
        throw new StopIteration();
    return nextValue;
}
counter.prototype._yieldReturn = function() {
    var exception;
    if (exception = this._yield_exception) {
        this._yield_exception = null;
        throw exception;
    } else {
        var value = this._yield_value;
        this._yield_value = this.thisPropertyDoesNotExist;
        return value;
    }
}
counter.prototype.send = function(yieldValue) {
    if (typeof yieldValue != 'undefined') {
        if (this._continuation == 0)
            throw new TypeError();
        this._yield_value = yieldValue;
    }
    return this.next();
}

var f = counter();
```

```
print(f.next()); // prints 0
print(f.next()); // prints 1

f = counter(3);
print(f.next());  // prints 3
print(f.send(8));   // prints 8
print(f.next());  // prints 9
```

4.2.4. *The `close()` method and `finally` blocks.* To quote from a previous paragraph:

> If, on the other hand, the `yield` to which we are resuming is inside one of the branches which belongs to the current instruction then we must guide execution toward it.

The `close` method is the opposite of this: when the user invokes it, the generator must return to the point in the source where it last left off, but instead of starting to execute instructions it must execute all the `finally` clauses and nothing else. This brings into discussion another aspect which was neglected until now: `finally` clauses. Whenever a JavaScript function returns, any finally clauses which are active are executed. This happens only for `return` and not `yield` because the latter is a pausing mechanism and execution is expected to resume later. By replacing `yield` statements with `return` equivalents we are changing the behavior: the `finally` clauses will be executed for each "yield" and—depending on what code is inside them—this may cause problems. The solution is simple, all we have to do is gate the content of the `finally` clause so it is only executed when the generator is closing and when it is not paused (e.g. `finally { ... }` becomes `finally { if (normal execution or closing) { ... }}`).

4.2.5. *When the generator uses `eval(...)`.* If the body of the generator calls the function `eval` then additional steps must be performed. Because the code compiled and executed by `eval` may reference local variables and we have moved all of them into properties of the generator instance, we must either update the code which eval will compile so all the references are resolved, or make sure the variables exist and are up to date before calling `eval`. The latter option is simpler than parsing, analyzing and changing the code which will be executed. For instance, if a generator uses two variables, `a` and `b`, and the first one is the name of a function, the line `b = eval(a+ "()")` would be changed into `var a = this._local_a, b = this._local_b; this._local_b = eval(a + "()")` which works as expected.

4.2.6. *Putting it all together.* In this section we have built a solution for translating generators for JavaScript into a structure which uses only ECMAScript version 3 constructs and is therefore supported in all ECMAScript environments. No code outside of the generator has to be changed which makes this a local solution. Listing 6 show the template for implementing a generator. The steps are:

(1) If the class `StopIteration` does not exist, create it and make it inherit `Error`.
(2) Create the class for the generator. Inside the constructor, create the start state with the arguments. Save all the arguments in local properties of the generator instance. Move the generator-function to the `_next` method.
(3) Redirect all references to arguments and local variables into properties of the generator instance.
(4) Implement `_continuation` and transform `yield` statements into `return` statements which also push the new state to the list of states. Update `finally` clauses so they don't do anything when the function returns.
(5) If the function `eval` is used, make all the variables reference the current (saved) state before invoking `eval`.
(6) Add the static methods `next()`, `_yieldReturn()`, `send(yieldReturn)`, `"throw"(exception)` and `close()`. It is also possible to place these methods inside a single object and inherit it using the `prototype` chain (which would generate less code when there is more than one generator).

Listing 6. The template used to translate a generator into ECMAScript 3 constructs

```
if (typeof StopIteration == 'undefined') {
    var StopIteration = function(){};
    StopIteration.prototype = new Error();
}

function generatorName(arg1, arg2, ...) {
    if (!(this instanceof generatorName))
        return new generatorName(arg1, arg2, ...);
    this._continuation = 0;
    this._yield_value = this.thisPropertyDoesNotExist;
    this._yield_exception = null;
    this._closing = false;
    this._local_arg1 = arg1;
    this._local_arg2 = arg2;
    ...
}
generatorName.prototype._next = function() {
    ... actual generator code ...
}
generatorName.prototype.next = function() {
    var nextValue = this._next();
    if (this._continuation < 0)
        throw new StopIteration();
    return nextValue;
}
```

```
generatorName.prototype._yieldReturn = function() {
    var exception;
    if (exception = this._yield_exception) {
        this._yield_exception = null;
        throw exception;
    } else {
        var value = this._yield_value;
        // the unlikely name is used to obtain a reference to the primitive <undefined>
        this._yield_value = this.thisPropertyDoesNotExist;
        return value;
    }
}
generatorName.prototype['throw'] = function(exception) {
    this._yield_exception = exception;
    return this.next();
}
generatorName.prototype.send = function(yieldValue) {
    if (typeof yieldValue != 'undefined') {
        if (this._continuation == 0)
            throw new TypeError();
        this._yield_value = yieldValue;
    }
    return this.next();
}
generatorName.prototype.close = function() {
    this._closing = {}; // create a new object
    while (this._closing)
        try {
            this['throw'](this._closing);
        } catch(e) {
            if (e == this.closing || (e instanceof StopIteration))
                this._closing = false;
            else
                // a different error was thrown
                throw e;
        }
}
```

5. Conclusion

JavaScript generators from language 1.7 can be successfully implemented on top of the baseline ECMAScript 3 using a source-to-source compiler.

The proposed method is complete and does not sacrifice any feature which JavaScript brings to the table in order to bring this support to legacy environments.

A compiler toolkit has been implemented which allows compiling new versions of JavaScript and targeting legacy platforms.

5.1. **Future Research.**

(1) The idea presented here can be taken further and support can be implemented for all the features added by recent versions of JavaScript.
(2) A block of code with a very small footprint can be used to decide at runtime whether to load the original JavaScript source or the translated one. This would eliminate any small speed-bumps which the translator may add as long as the user is using an environment which supports the complete set of JavaScript instructions while still supporting everyone with an outdated environment.
(3) Additionally, the Abstract Syntax Tree generated by the parser can be used in order to perform optimizations.
(4) The output module can be replaced with one which generates C++ code or LLVM intructions.

References

[1] Apache configuration for v8cgi. http://code.google.com/p/v8cgi/wiki/ApacheConfiguration, last updated May 12, 2010.
[2] Google Caja, a source-to-source translator for securing Javascript-based web content. http://code.google.com/p/google-caja/, retrieved Jun 9, 2010.
[3] Overview of the Caja system. http://code.google.com/p/google-caja/wiki/CajaOverview#Cajita, last updated Jul 17, 2009.
[4] Douglas Crockford. JavaScript: The world's most misunderstood programming language, 2001. http://www.crockford.com/javascript/javascript.html.
[5] Douglas Crockford. JavaScript: The Good Parts. O'Reilly, May 2008.
[6] Ecma International. ECMA-262, 5 edition, December 2009. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf
[7] Eli Grey. Pausing JavaScript with async.js. Eli Grey's Blog, January 2010. http://eligrey.com/blog/post/pausing-javascript-with-async-js.
[8] Neil Mix. Threading in JavaScript 1.7. Neil Mix's blog, January 2007. http://www.neilmix.com/2007/02/07/threading-in-javascript-17/.
[9] Mozilla Developer Center. New in JavaScript 1.7, 2006. https://developer.mozilla.org/en/New_in_JavaScript_1.7 (revision 144).
[10] Mozilla Developer Center. Iterators and Generators, 2007. https://developer.mozilla.org/en/Core_JavaScript_1.5_Guide/Iterators_and_Generators (revision 13).

Master student, Babeş-Bolyai University, Faculty of Mathematics and Computer Science, 1 M. Kogălniceanu St., 400084 Cluj-Napoca, Romania
E-mail address: `bluepx@gmail.com`