Beyond .*Script

Implementing A Language For The Web

Veit Heller

July 15, 2016

A thesis submitted for the degree of
B.Sc. of Applied Computer Science of
The University of Applied Sciences Berlin

For Meredith, Tobias and all the people who cope with me. Your undying support will not be forgotten.

Except where otherwise indicated, this thesis is my own original work.

Veit Heller
July 15, 2016

Abstract

This thesis presents and evaluates a port of the zepto programming language to the web. It aims to work as seamlessly with existing technologies as possible and is largely influenced by R5RS, a standard of the Lisp derivative Scheme.

It is the result of over a year of independent research on zepto, a new programming language targetting various environments. The prototype described here and implemented for this thesis runs on JavaScript, a language found natively in many web browsers.

The central problem addressed by this thesis is the incoporation of a language runtime into the web ecosystem without the need for extensive code rewrites while supporting most features of the reference implementation of zepto, including large parts of the standard library. This also includes interoperability between the two languages.

It is also discussed inhowfar tooling for both JavaScript and zepto, most specifically their respective package managers, interoperate for building larger scale web applications.

Contents

Abbreviations

API Application Programming Interface. 1, 12–15

AST Abstract Syntax Tree. 9

DOM Document Object Model. 1, 12, 13, 16

FFI Foreign Function Interface. 12, 14–16

FRP Functional Reactive Programming. 6, 7

GHC Glasgow Haskell Compiler. 6, 9, 11, 12

GUI Graphical User Interface. 6

IR Intermediate Representation. 8

REPL Read-Eval-Print Loop. 12

W3C World Wide Web Consortium. 12

YUI Yahoo User Interface Library. 1

1 Introduction

> Controlling complexity is the essence of computer programming.
>
> *(B. Kernighan)*

1.1 Motivation

JavaScript has, since its inception, attracted a lot of controversy. This is rooted in various aspects of its design, from prototypal inheritance and the DOM to its operator precedence. Especially prototypal inheritance has been the root of a lot of discussions in the Computer Science community. It has the reputation of being counter-intuitive, though it is older than JavaScript, the first commonly known programming language that implemented prototypal objects being Self.

This and a few other design choices have prompted many programmers to develop wrapper libraries around almost anything that comprises the language, the arguably most widely used example being jQuery, a library that abstracts over the DOM. But more arcane topics have been covered as well: there is the Yahoo User Interface Library (YUI), a now abandoned project of Yahoo that aims to abstract over web Application Programming Interfaces (APIs) as well but also comes with its own inheritance model, likening JavaScript prototypes to classical classes by introducing special functions and properties to the objects such as `extend` and `superclass`.

All of those libraries have shaped the way JavaScript has developed and in the upcoming revisions of ECMAScript[1], ECMAScript 6[2] and ECMAScript 7[3], a lot of conve-

[1] ECMAscript is the officially trademarked name of what application developers and browser vendors most commonly refer to as JavaScript.

[2] ECMAScript 6 was renamed to ECMAScript 2015 along the way but for the sake of clarity this thesis uses the more commonly know name of the revision. For an up-to-date version of that revision please refer to (ECMA International, 2015).

[3] Now ECMAScript 2017. For an up-to-date version of the current draft please refer to (ECMA International, 2016).

niences from third-party libraries have been adopted by the "vanilla"[4] JavaScript canon. Among these features has been the widely controversial introduction of a class notation for JavaScript that makes prototypes feel more like classes (without breaking existing semantics).

1.1.1 Preprocessors & Transpilers

This and other features have been available to JavaScript developers for much longer than the latest iterations of the JavaScript language and without the need to rely on a possibly massive library - provided they use a preprocessor. One of the earliest exemplars of preprocessing for JavaScript, CoffeeScript, included lambdas, a shorthand for anonymous functions; pattern matching, a technique for destructuring data; and classes, all of which found their way into the latest ECMAScript revisions.

Preprocessing can serve a lot of different purposes apart from providing syntactic commodities to the programmer. Babel, for instance, is a relatively novel library for transpiling JavaScript that adheres to the new standards of ECMAScript back to older revisions, for the sake of backwards-compatibility. It also aims to provide small optimizations before the code reaches the compiler, such as constant hoisting, a compiler optimization technique that eliminates pure functions that always produce the same result in favour of global constants to reduce the overhead of function calls.

Another way to use JavaScript as a platform that gains ever more importance is the transpilation from another language. One could argue that CoffeScript is nothing short of a transpiler, but one could also argue against that notion, considering its only purpose is to compile to JavaScript. Other general-purpose languages such as Clojure and C++, originally developed to run on other platforms, have the option to compile to JavaScript through ClojureScript (Hickey, 2016) and Emscripten (Zakai, 2013) respectively.

Caveats

This thesis was inspired by the work done in the field of transpilation to JavaScript. Its goal is not to present yet another attempt at transpilation, but rather to rethink the way languages are incorporated into the web of today.

At their heart, all of the preprocessors and transpilers that target JavaScript are JavaScript, even if the syntax and semantics differ radically. There needs to be a clean

[4] A common name for JavaScript that refrains from leveraging paradigm-altering libraries.

mapping of the source language to JavaScript. Because of its flexibility, this is often a doable task, albeit not always desirable.

The value of having the same abstraction present at runtime as at compile-time is obvious if the language that is consider places strong emphasis on code mutability, such as in Lisp or Elixir[5].

1.2 Goals of this Thesis

The primary goal of this thesis is to present a novel approach at implementing languages for the Web. This is exemplified by a sample implementation of a non-trivial, pure and largely feature-complete functional programming language that has been tested in production systems.

Many reasons would speak for a procedural or objective-oriented language as a prototype. For one, the language itself could be more easily expressed in terms of JavaScript if it is reasonably close to it. Another point that speaks against using a functional - and especially pure - programming language is that this might make the interoperability harder because JavaScript is intrinsically impure and stateful.

But all of those reasons could also be read as argument for the implementation of a functional language. The difference showcases the ability of JavaScript to express concepts foreign to the language in it. It also serves as a better test bench for more advanced implementation patterns.

This is especially true for those features of the base language that are not present in JavaScript and non-trivial to add to it from a user perspective. The features present in the target language implemented in this thesis but not in JavaScript most notably include Macros[6] and Continuations[7]. Those features were chosen especially because they are extremely foreign to JavaScript programming.

1.3 Structure of this Thesis

Chapter 2 examines related work in the field of cross-compilation into JavaScript and implementation of interpreters that are directly embeddable into larger systems. This includes desktop applications, game scripting engines and creative suites.

[5] As exemplified by Chris McCord in (McCord, 2015).
[6] The ability to rewrite code at parse or compile time, see 3.1.3.
[7] The control state of a program represented as a data structure within the code, see 3.1.4.

Chapter 3 gives an overview of the concept design and how the features are laid out to match the needs of both the goals of this thesis and the prototype itself.

Chapter 4 presents the system design and how the prototype integrates into existing web components.

Chapter 5 discusses the implementation, picking out different fundamental parts of the system and presents how they work.

Chapter 6 evaluates the prototype. This includes problems such as how well the integration of the system worked and how it compares to the reference implementation of zepto.

Chapter 7 gives a brief summary of what was done and gives an outlook to what might happen with zepto, both the desktop and the JavaScript version, in the future.

2 Related Work

This section aims to give a quick overview of work that has already been done in the field of transpilation to and language implementations on top of JavaScript. A few of the most important specimens have already been mentioned briefly in 1.1.1, although their relationship to this thesis have not yet been discussed.

2.1 Existing Projects

In this section, a brief overview over a short, not necessarily exhaustive list of transpilers from existing programming languages to and languages explicitly acting as a layer of abstraction over JavaScript shall be presented.

The aim of this section is not to present the reader with the syntax and semantics of every single language that is discussed, but rather equip them with a general overview of the ecosystem at the time of writing and how it correlates to the work presented in this thesis.

2.1.1 Transpilers

Transpilers are normally defined as compiling one high level language into another. They are often referred to as cross-compilers, although this term is imperfect as it also refers to compilation from one hardware platform or operating system to another.

Due to this ambiguity, the name *transpiler* was chosen to refer to this kind of system in the context of this thesis.

GHCJS

GHCJS - often hyphenated as GHC-JS - is a transpiler from the Haskell language to JavaScript. It is currently maintained by Luite Stegeman. It aims to ”solve the JavaScript problem”[1]. It enables the user to compile any Haskell program to JavaScript

[1] A talk given by Stegeman bore this title, see (Stegeman, 2015).

instead of machine code by the means of inserting a custom compiler backend into the Glasgow Haskell Compiler (GHC) toolchain.

It is the compiler used in this thesis to transpiler the existing zepto codebase to JavaScript.

Emscripten

ClojureScript

2.1.2 Abstraction Languages

The languages mentioned here are languages that purely abstract over JavaScript, choosing it as their primary backend. This differs from the languages discussed in 2.1.1, which are general purpose languages with the option to compile to JavaScript if desired by the developer.

This list is not designed to be exhaustive, but rather aims to inform about recent developments in web programming.

CoffeeScript

As mentioned int 1.1.1, CoffeeScript was one of the first transpilers that targetted JavaScript when it first appeared[2]. As also previously explained, many of the constructs have been adopted by JavaScript by now. CoffeeScript is similar to JavaScript in that it is a procedural, prototypal language. No semantic features have been added or removed from JavaScript, making it relatively dissimilar to the programming system presented in this thesis.

Elm

Elm as it is presented in (Czaplicki, 2012) is a language centered around Functional Reactive Programming (FRP)[3] , with a special regard to browser-based Graphical User Interface (GUI)s. Borrowing its syntax largely from Haskell it is a pure, functional programming language. Notable features include GUI-centric tooling, such as a ”time-

[2]The first git commit dates back to December 13th, 2009. The first public release happened shortly after that.

[3]An in-depth explanation of FRP is out of the scope of this thesis, but it is a programming style originally formulated by Paul Hudak and Conal Elliott in a 1997 paper (**frp**).

traveling" debugger that caches the state of a program at any given time, allowing for rewinding, forwarding and jumping.

The similarities with this thesis go insofar as it is a functional programming language for the Web, but it compiles to JavaScript, HTML and CSS, once again falling short of actually getting the code to the browser. It is also very centered around the aforementioned FRP, whereas the runtime presented in this thesis aims to be a general-purpose programming environment.

PureScript

PureScript is, much like Elm, a functional, pure programming language that borrows a lot of its syntax from Haskell. It aims to compile into human-readable JavaScript and make general purpose programming for the web in a strongly typed, functional style possible.

It is different from GHCJS insofar as it does not provide compatibility with Haskell code, although many libraries and functions are relatively similar. See 2.1.2 for an example.

as It shares a lot of features with Elm, it also largely shares the differences to the system presented in this thesis.

```
1  -- | Extract the first element of a list (as implemented in the language standard library).
2  -- | The Haskell version is unsafe and will throw an error
3  -- | if it encounters an empty list.
4  head :: [a] -> a
5  head (x:_) =  x
6  head [] =  badHead -- an exception function
7
8  -- | the list data type in PureScript is safe by design and
9  -- | the function returns a Maybe monad.
10 head :: forall a. List a -> Maybe a
11 head Nil = Nothing
12 head (Cons x _) = Just x
```

Listing 1: A juxtaposition of a simple function in Haskell and PureScript.

2.2 Existing Standards

3 Concept Design

Practicality beats purity.

(T. Peters—The Zen of Python)

3.1 Construction Design

NOTE: This had to be moved from chapter 2 to match the current design of the thesis. Please excuse the lack of coherence.

3.1.1 Lisp

A common saying among programming language designers is that every programmer has written their own implementation of Lisp. There are a lot of different implementations of Lisp in the wild, even ones that compile to JavaScript[1].

The main reason for that is often cited to be the simplicity of the language on a parsing level. A simple Lisp can be implemented in less than one hundred lines of code, if no intermediate representation is generated. This is made possible by the unique property of Lisp of enclosing every statement in parentheses, where the first element within those parentheses is the statement name and the other elements are the arguments. It can be evaluated straight from a textual level, because things such as operator precedence and statement amiguity do not exist. In regular Lisp as specified in the initial paper by John McCarthy (McCarthy, 1960) only six special forms exist to allow not only for Turing-completeness, but also for expressiveness.

3.1.2 zepto

Zepto is a new Scheme implementation that aims to be as small as possible, to be able to target a lot of different backends. Currently, LLVM and Erlang Core[2] bindings are

[1] such as ClojureScript, a backend of the Clojure compiler that targets JavaScript.

[2] Erlang Core is the Intermediate Representation (IR) of Erlang code before it is complied. Resources and documentation about it are sparse, it mostly seems to exist inside the BEAM's implementation.

under development, the reference implementation is a simple interpreter that interprets code directly from the Abstract Syntax Tree (AST). This is slow but ensures a small interpreter size[3]. The compilers are written directly in zepto itself.

The small code base makes zepto a good target for porting it to the web. Further, because it is written in Haskell the code base was expected to be possibly almost entirely compilable to JavaScript using GHCJS, a backend for the GHC targetting JavaScript instead of native code. It offers many advanced features such as inlining of JavaScript into the code base using a technique called quasi-quoting, where a special character sequence delimits the inlined code, much like regular quotes. This tool set was expected to make the work of porting an existing language to the web as simple as possible.

Of course there are other reasons to use a functional language as an example. With both syntax and semantics differing strongly from JavaScript, this example makes way for languages more closely related to JavaScript also making their eventual way into the browser.

3.1.3 Macros

3.1.4 continuations

3.2 Additional Features

A small paper (**ERL**) that describes its' basics was used to implement the compiler from zepto.
[3]The entire codebase is only about 4000 lines of Haskell code.

4 System Design

Practicality beats purity.

(T. Peters—The Zen of Python)

4.1 Integration into the Web Ecosystem

5 Implementation

> It always takes longer than you expect, even when you take into account Hofstadter's Law.
>
> *(Hofstadter's Law)*

The implementation philosophy philosophy of the port presented in this thesis has always been to reuse as much code from the reference implementation as possible. This guided the flow of design choises down a rather natural path and thus kept the implementation described here fairly short and relatively trivial.

5.1 Description of the Toolchain

The tooling uses GHCJS, which is a backend for the GHC compiler that targets JavaScript rather than native code (TODO: remove this from introduction). This makes cross-compiling the code base to JavaScript a rather simple undertaking.

GHCJS offers many advanced features such as inlining of JavaScript into the code base using a technique called quasi-quoting, where a special character sequence delimits the inlined code, much like regular quotes. This tool set was expected to simplify rewriting the code bits that needed adjustment.

Another important feature was the management of pure JavaScript sources. GHCJS provides management of these sources akin to the management of bits of C code that should interface with Haskell in a regular GHC project. The build file includes a segment called `js-sources` which ensures that the JavaScript in those sources will be included in the final compilation. The informal convention seems to be to put the files that should be included into a directory called `jsbits` (a hat-tip to the GHC convention of putting C sources into cbits).

The code emitted by GHCJS is conservative in feature use. This is an important feature if older clients with possibly even obsolete browsers want to access web pages

supported by zepto code, rare as they might be.

All these conveniences do not spare the programmer from the actual programming process, of course, and while GHCJS seems like a well-maintained project one has to keep in mind that it is not backed by a consortium as GHC is and the current maintainer seems to do this work seemingly completely in his spare time. This leaves us with the notion that the projectm could be abandoned at any point and being too dependent on its' more unusual features could lead to a product that is unnecessarily hard to maintain.

All of this considered, a preliminary analysis revealed that the two main conditions for the success of this thesis were given: the code emitted by GHCJS is fast, solid and reasonably compact and the newest features of both JavaScript and Haskell were supported[1].

5.2 Description of the Implementation

As predicted in 1.1, the code base of zepto could be reused in almost its' entirety. What had to be rewritten was mostly related to the startup of the interpreter, because the regular paths into the code - either via a script being passed into it or launching an interactive Read-Eval-Print Loop (REPL)[2] - were unavailable in the browser context. Instead, a way of passing the sources from within `script` tags needed to be found. Further customizations include a FFI to enable better cross-evaluation of JavaScript and the adaptation of existing APIs, such as the DOM.

5.2.1 The `script` tag

Initially, a DOM node walker was considered, but rejected relatively early because of two reasons: Firstly, it introduced a layer of complexity from within JavaScript code that would have likely made it brittle and hardly portable. Secondly, it would require a walk of the nodes every time a DOM element is inserted or replaced, which is a common occurence in modern interactive web applications.

As of November of 2015, the World Wide Web Consortium (W3C) specifies an API that simplifies this process for the programmer. Within their specification of the DOM4

[1] The newest version of GHC at the time of writing, version 8.0, gained support while this thesis was in the works.

[2] A REPL is an interactive code evaluation environment. Code is typed into a prompt and immediately evaluated. The convenience of such a short feedback loop is often used in the context of scripting languages and shells.

(World Wide Web Consortium, 2015), the fourth specification of APIs for the Web, an object called MutationObserver is included which is able to register for DOM manipulations. Its main function will be triggered whenever a change occurs within the DOM part that it registered for listening to.

This simplifies the implementation of a listener to DOM events a great deal. Only minimal programming is required to configure the listener and to filter out all the nodes that are not script nodes of the type text/zepto[3].

A problem untended to with that method was nodes insert before the listener starts. This was resolved by singling out all the script tags that are present before the listener starts and applying the same filter/evaluation function to all of them. This also ensures that they are executed before any additional code (and possibly dependent) is passed into the zepto object.

The code was then included in the zepto singleton, which is the global interpreter object used for the management and interactivity of the zepto interpreter.

[3]This was chosen in analogy to the existing text/javascript node type.

```
// the initial observer and the function it takes
zepto.observer = new MutationObserver(zepto.handleMutation);

// this function will get a list of mutations and apply handleDom to them
zepto.handleMutation = function(mutations) {
  mutations.forEach(mutation => {
    mutation.addedNodes.map(zepto.handleDom);
  });
}

// evaluate if it is a text/zepto node
zepto.handleDom = function(node) {
  if (node.nodeName != "SCRIPT" || node.type != "text/zepto") {
    return null;
  }
  return zepto.eval(node.innerHTML);
}

// execute this on startup
window.onload = () => {
  let scripts = document.getElementsByTagName("script");
  scripts.map(zepto.handleDom);
}
// the extra arguments signify recursive listening
zepto.observer.observe(document, {childList: true, subtree: true});
```

Listing 2: The final mutation observer code (simplified)

5.2.2 The FFI

The FFI is a central part of the port. If it weren't usable, none of the browser's capabilities could be used from within zepto, thus rendering the effort of bringing zepto into the browser effectively useless. The APIs of the Web are a big part of what it means to program for the browser, after all.

An initial sketch of the programming interface was extremely simplistic: a call to the function `js` could be called with a string as argument, representing the textual representation of the JavaScript program that should be run. It was piped to the JavaScript function `eval` and the function returned an affirmative truth value. Quasi-quoting larger blocks of JavaScript was also possible.

Of course this is unusable. The missing return value makes any effort of talking to an API impossible, as one could never yield any results. A different kind of return value is needed.

The obvious but most challenging to implement solution would be to infer a fitting zepto type for every return value in JavaScript and return a result depending on that. While this could be seen as a rather elegant solution, it comes with its own set of caveats and exceptions, as the mapping between JavaScript and zepto values is not always obvious. A JavaScript object has too many properties that get lost in the process of translating it to zepto as to make it intuitive.

```
1 ; this would return an integer
2 (js "1 + 1")
3
4 ; this would return a hashmap
5 (js "{key: \"val\""})
6
7 ; this is problematic, because it will return an object
8 (js "new Error()")
```

Listing 3: The ideal FFI

Another problematic point is the implementation of JavaScript values in GHCJS. They are opaque datatypes, aliases for addresses and byte vectors. While zepto supports byte vectors and pointers, they are hardly a good representation for semantically rich prototypes as they only offer a glance into the underlying implementation of the JavaScript engine. While it is true that GHCJS itself provides methods for type coercion, they are crude and possibly error-prone.

A simpler method that is still mostly sensible came up: returning the string values of all of the values returned. While this places the burden of coercion into the programmer's hand, it also gives them the power to make their own decisions of how to deserialize values. Functions for deserializing the most common datatypes are included in the standard library of the JavaScript implementation of zepto, to aid the programmer in the process of finding the right methods of getting a value out of the FFI.

This still does not solve the problem of helping manage classes, but it empowers the programmer to find their own ways of serializing on the JavaScript side and deserializing

on the zepto side to preserve the information they need in their specific programming context.

All of this needs an additional layer of abstraction to avoid unnecessary boiler plate, but it is stable enough for most purposes that zepto in JavaScript was used for yet.

```
1 ; the function string->number is a standard zepto function
2 (string->number (js "Math.pow(2, 32)"))
3
4 ; this is an example of how to resolve the earlier problem:
5 ; override the prototype of the object to return the value that is needed
6 (js "Error.prototype.toString = function() { return this.message; }")
7 (error (js "new Error(\"fatal error occured\")"))
```

Listing 4: The final form of the FFI

Implementing the JavaScript to zepto FFI was much simpler, as the interpreter is defined within the JavaScript environment. A call to the `eval` function of the `zepto` object with a string as argument will return in the execution of this piece of code and the return the textual representation of the zepto object so that the entire communication between the languages is string-based.

5.2.3 The DOM

After building the FFI, it was possible to implement the entire communication with the DOM in terms of calls to foreign functions and the parsing of their return values. This allows for a stable library, because it is unintrusive and does not interfer with existing JavaScript constructs.

Existing zepto-js projects often find it convenient to write a hybrid mix of JavaScript and zepto code that calls each other at certain points. This is, however, probably not advisable at the layer of libraries or utilities, because it is at risk of getting in the way of the job.

```
(module "dom"
  (export
    `("create" ,create)
    `("insert" ,insert)
    `("get" ,get))

  (loads "html")

  (update-dom! (lambda (node contents)
    (js (++ "document.getElementById('" node "').innerHTML = '"
          (create contents)
          "';"))))

  (create (lambda (tree)
    ((import "html:build") tree)))

  (insert (lambda (context tree)
     (update-dom! context tree)))

  (get (lambda (node)
    ((import "html:parse")
      (js (++ "document.getElementById('" node "');"))))))
```

Listing 5: A minimal DOM module

6 Evaluation of the Prototype

> When I'm working on a problem, I never think about beauty. I think only how to solve the problem. But when I have finished, if the solution is not beautiful, I know it is wrong.
>
> *(R. Buckminster Fuller)*

6.1 Seamlessness of Integration

6.2 Test Against Standard Implementation of Zepto

* added: ffi
  * removed load statement, REPL functionality
  * inserting libraries?

7 Summary and Outlook

When I'm working on a problem, I never think about beauty. I think only how to solve the problem. But when I have finished, if the solution is not beautiful, I know it is wrong.

(R. Buckminster Fuller)

* porting efforts
  * compiler efforts
  * zeps
  * classes

8 Conclusion

* fixing the web not necessary
  * rethinking it possible

References

Aho, A. V., M. S. Lam, R. Sethi, and J. D. Ullman (2006). *Compilers: Principles, Techniques, and Tools*. Pearson Eudcation, Inc.

Czaplicki, E. (2012). *Elm: Concurrent FRP for Functional GUIs*. Senior Thesis (cit. on p. 6).

ECMA International (2015). *ECMAScript®2015 Language Specification*. URL: http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf (visited on 12/07/2016) (cit. on p. 1).

ECMA International (2016). *ECMAScript®2017 Language Specification*. URL: https://tc39.github.io/ecma262/ (visited on 12/07/2016) (cit. on p. 1).

Elliott, C. and P. Hudak (1997). "Functional Reactive Programming". In: *Proceedings of the second ACM SIGPLAN international conference on Functional programming*.

Fogus, M. (2013). *Functional JavaScript*. O'Reilly Media.

Hickey, R. (2016). *The ClojureScript Wiki*. URL: https://github.com/clojure/clojurescript/wiki (visited on 14/07/2016) (cit. on p. 2).

McCarthy, J. (1960). "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". In: *Communications Of The ACM* (cit. on p. 9).

McCord, C. (2015). *Metaprogramming Elixir: Write Less Code, Have More Fun*. The Pragmatic Programmers, LLC (cit. on p. 3).

Parr, T. (2010). *Language Implementation Patterns*. The Pragmatic Programmers, LLC.

Queinnec, C. (2003). *Lisp in Small Pieces*. Cambridge University Press.

Stegeman, L. (2015). "Solving the JavaScript Problem". CodeNode (cit. on p. 5).

World Wide Web Consortium (2015). *W3C DOM*. URL: https://www.w3.org/TR/dom/ (visited on 12/07/2016) (cit. on p. 14).

Zakai, A. (2013). *Emscripten: An LLVM-to-JavaScript Compiler*. URL: https://github.com/kripken/emscripten/blob/master/docs/paper.pdf (visited on 14/07/2016) (cit. on p. 2).