

Beyond .*Script

Implementing A Language For The Web

Veit Heller

July 4, 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 4, 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. It is largely influenced by R5RS Scheme.

Contents

Abstract	ii
Abbreviations	v
1 Introduction	1
1.1 Motivation	1
1.1.1 Preprocessors	2
1.1.2 Lisp	3
1.1.3 zepto	3
1.2 Goals of this Thesis	4
1.3 Structure of this Thesis	4
2 Related Work	5
2.1 Existing Projects	5
2.2 Existing Standards	5
3 Concept Design	6
3.1 Construction Design	6
3.2 Additional Features	6
4 System Design	7
4.1 Integration into the Web Ecosystem	7
5 Implementation	8
5.1 Description of the Toolchain	8
5.2 Description of the Implementation	9
5.2.1 The <code>script</code> tag	9
5.2.2 The Foreign Function Interface (FFI)	11
5.2.3 The Document Object Model (DOM)	13

Contents

6	Evaluation of the Prototype	15
6.1	Seamlessness of Integration	15
6.2	Test Against Standard Implementation of Zepto	15
7	Summary and Outlook	16
8	Conclusion	17
	References	18

Abbreviations

API Application Programming Interface. 1, 9–12

AST Abstract Syntax Tree. 3

DOM Document Object Model. 1, 9, 10, 13

FFI Foreign Function Interface. 9, 11–13

GHC Glasgow Haskell Compiler. 3, 8, 9

IR Intermediate Representation. 3

REPL Read-Eval-Print Loop. 9

W3C World Wide Web Consortium. 9

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 and its Application Programming Interface (API) to 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 examples being jQuery, a library that abstracts over the DOM. But more arcane topics have been covered as well: there is Yahoo User Interface Library (YUI), a now abandoned project of Yahoo that aims to abstract over web 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 ECMAScript¹, ECMAScript 6² and ECMAScript 7, a lot of conveniences from third-party libraries have been adopted by the "vanilla"³ JavaScript canon.

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

²ECMAScript 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.

³A common name for JavaScript that refrains from leveraging paradigm-altering libraries.

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

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.

Caveats

****STUB: TODO****

In its' heart, all of the preprocessors and transpilers that target JavaScript are JavaScript, even if the syntax and semantics differ radically. This makes for an interesting case, because it might hinder the web itself from evolving: at its' core, there is one technology only.

Obviously, a lot of research is going into the development of JavaScript and it's tools and it might be ignorant to dismiss JavaScript tooling outright, though. Among transpiler writers, JavaScript is hailed as the assembly of the web.

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

1.1.2 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 JavaScript⁴.

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 and the other elements are the arguments. It can be evaluated straight from a textual level, because things such as operator precedence and statement ambiguity 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.

1.1.3 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 Core⁵ bindings are 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 size⁶. 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 GHC-JS, a backend for the Glasgow Haskell Compiler (GHC) targeting 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 wildly from JavaScript, this example makes way for

⁴such as ClojureScript, a backend of the Clojure compiler that targets JavaScript.

⁵Erlang Core is the Intermediate Representation (IR) of Erlang code before it is compiled. Resources and documentation about it are sparse, it mostly seems to exist inside the BEAM's implementation.

⁶The entire codebase is only about 4000 lines of Haskell code.

languages more closely related to JavaScript also making their eventual way into the browser.

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 functional programming language.

- * functional because different
- * fairly different feature set

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.

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 short 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

Practicality beats purity.

(T. Peters—The Zen of Python)

2.1 Existing Projects

2.2 Existing Standards

3 Concept Design

Practicality beats purity.

(T. Peters—The Zen of Python)

3.1 Construction Design

3.2 Additional Features

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 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 choices 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

5 Implementation

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 project 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 supported¹.

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 occurrence 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,

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

²A 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.

5 Implementation

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 unintended 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.

³This was chosen in analogy to the existing `text/javascript` node type

```

1 // the initial observer and the function it takes
2 zepto.observer = new MutationObserver(zepto.handleMutation);
3
4 // this function will get a list of mutations and apply handleDom to them
5 zepto.handleMutation = function(mutations) {
6   mutations.forEach(mutation => {
7     mutation.addedNodes.map(zepto.handleDom);
8   });
9 }
10
11 // evaluate if it is a text/zepto node
12 zepto.handleDom = function(node) {
13   if (node.nodeName !== "SCRIPT" || node.type !== "text/zepto") {
14     return null;
15   }
16   return zepto.eval(node.innerHTML);
17 }
18
19 // execute this on startup
20 window.onload = () => {
21   let scripts = document.getElementsByTagName("script");
22   scripts.map(zepto.handleDom);
23 }
24 // the extra arguments signify recursive listening
25 zepto.observer.observe(document, {childList: true, subtree: true});

```

Listing 1: 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.

5 Implementation

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 2: 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

5 Implementation

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 3: 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 interfere with existing JavaScript constructs.

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

5 Implementation

```
1 (module "dom"
2   (export
3     `("create" ,create)
4     `("insert" ,insert)
5     `("get" ,get))
6
7   (loads "html")
8
9   (update-dom! (lambda (node contents)
10     (js (++ "document.getElementById(" node ").innerHTML = "
11       (create contents)
12       "";")))
13
14   (create (lambda (tree)
15     ((import "html:build") tree)))
16
17   (insert (lambda (context tree)
18     (update-dom! context tree)))
19
20   (get (lambda (node)
21     ((import "html:parse")
22       (js (++ "document.getElementById(" node ");"")))))
```

Listing 4: 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 Education, Inc.
- Fogus, M. (2013). *Functional JavaScript*. O'Reilly Media.
- McCarthy, J. (1960). “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”. In: *Communications Of The ACM* (cit. on p. 3).
- Parr, T. (2010). *Language Implementation Patterns*. The Pragmatic Programmers, LLC.
- Queinnec, C. (2003). *Lisp in Small Pieces*. Cambridge University Press.