Beyond .*Script

Implementing A Language For The Web

Veit Heller

June 17, 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
June 17, 2016

Abstract

The modern web is comprised of an abundance of very different beasts. Technologies that powered the first versions of the World Wide Web, such as HTML, CSS and JavaScript, and relatively new conceptions like TypeScript, CoffeScript, PureScript, ClojureScript, Elm, LASS, SCSS, Jade and Emscripten - to name but a few - are shaping the internet as we know it. There is one flaw that many of the new technologies have in common, as different as they may look and feel - they are mere preprocessors. In the end, it all boils down to the classic technologies again and we are left with the same limited capabilities we have had for the last twenty years.

This thesis presents a port of the zepto programming language to the web. It aims to work as seamlessly with existing technologies as possible.

Contents

Abbreviations

API Application Programming Interface. 4

AST Abstract Syntax Tree. 2

DOM Document Object Model. 4

FFI Foreign Function Interface. iii, 4–6

GHC Glasgow Haskell Compiler. 2

IR Intermediate Representation. 2

REPL Read-Eval-Print Loop. 4

W3C World Wide Web Consortium. 4

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 to operator precedence. Prototypal inheritance has the reputation of being counter-intuitive, though it is older than JavaScript, the first commonly known programming language that implements prototypal objects being Self.

* things get better
* es 6 and es7 thank god
* a lot of research funneled into it
* still a fundamental rethinking might be necessary

1.2 Purpose of this work

1.3 Structure of this work

2 Motivation

> Practicality beats purity.
>
> (T. Peters—*The Zen of Python)*

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 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.

2.1 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 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, ecause 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) 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 wildly from JavaScript, this example makes way for

[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.

[3] The entire codebase is only about 4000 lines of Haskell code big

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

3 Implementation

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

As predicted in 2, 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)[1] - 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 Application Programming Interface (API)s, such as the Document Object Model (DOM).

3.1 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, 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`[2].

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.

[1] 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.

[2] This was chosen in analogy to the existing `text/javascript` node type

The code was then included in the zepto object, which is the global interpreter object used for the management and interactivity of the zepto interpreter[3].

```
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 3.1: The final mutation observer code (simplified)

[3]It also conveniently serves as the entrypoint for a FFI from JavaScript to zepto.

3.2 The FFI

The FFI

4 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*)*

5 Conclusion

References

Aho, A. V., M. S. Lam, R. Sethi, and J. D. Ullman (2006). *Compilers: Principles, Techniques, and Tools*. Pearson Eudcation, 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. 2).

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

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