Beyond .*Script

Implementing A Language For The Web

Veit Heller

August 1, 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
August 1, 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, 15–18

AST Abstract Syntax Tree. 11

DOM Document Object Model. 1, 15, 16, 19

FFI Foreign Function Interface. 6, 7, 15, 17–19

FRP Functional Reactive Programming. 8

GHC Glasgow Haskell Compiler. 6, 11, 14, 15

GUI Graphical User Interface. 8

IR Intermediate Representation. 11

JIT Just In Time Compiler. 6

LLVM Low Level Virtual Machine. 6, 10, 11

REPL Read-Eval-Print Loop. 15

W3C World Wide Web Consortium. 15

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 most widely-used example of this is arguably jQuery, a library that abstracts over the DOM. But more arcane topics have been covered as well: there is, for example, the Yahoo User Interface Library (YUI), a now-abandoned project of Yahoo that aimed to abstract over web Application Programming Interfaces (APIs) but also comes with its own inheritance model. This model likens 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 without them needing 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.

Transpilation from a different language is another increasingly popular way to use JavaScript as a platform. One could argue that CoffeScript is nothing short of a transpiler; but one could also argue against this idea, 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. On one hand, the language itself could be more easily implemented in JavaScript if it is reasonably close to it. Another point that speaks against using a functional - and especially pure - programming language is the possibility of making 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 JavaScript's ability 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.

GHCJS is the compiler used in this thesis to transpile the existing zepto codebase to JavaScript. Its plug-and-play design, complete enough implementation of the Haskell programming langauge and relative maturity all played a role in the choice to use it as an implementation language[2].

Emscripten

Emscripten as discussed in (Zakai, 2013) aims to be a transpiler from Low Level Virtual Machine (LLVM) to JavaScript. Its primary focus is the translation of C and C++ source code to JavaScript.

Its use case is similar to that of GHC-JS, enabling the user to both ”(1) Compile code directly into LLVM assembly, and then compile that into JavaScript using Emscripten, or (2) Compile a language's entire runtime into LLVM and then JavaScript, as in the previous approach, and then use the compiled runtime to run code written in that language” (ibid.), the latter of which sounding suspiciously like what zepto-js tries to achieve. There has been efforts to make the PyPy Just In Time Compiler (JIT)[3] work in the browser, and while it seems to have gained a fair bit of traction and work reasonably well[4], documentation on it is sparse and there is seemingly no academically viable source detailing its development and architecture.

ClojureScript

ClojureScript is a backend for the Clojure programming language that targets JavaScript. Initially developed by Rich Hickey, the author of Clojure, it is now maintained under David Nolen's lead (Hickey, 2016).

Being a Lisp, there are obvious syntactic similarites to zepto. Yet, as it is transpiled rather than interpreted directly in the browser, programming against it is quite difficult. Specifically, the way the FFI works is quite apparently** different, as ClojureScript has a lot of syntactic integrations that help embed JavaScript within it. **new sentence** all functions prefixed with a period are assumed to be JavaScript functions on the prototype

[2]Further discussion of the design and decision process can be found in 3 and 4.
[3]A JIT aimed to speed up Python programs by analyzing and compiling them, described in (Bolz et al., 2009).
[4]A live demonstration of how it works can be found on its website, accessible under https://pypyjs.org.

of the object provided as first argument. JavaScript values can be directly accessed through the js namespace. Zepto's design is quite different, as is discussed in 5.2.2.

```
; ClojureScript provides syntactic abstractions over
; the embedding of foreign code.
(.getElementById js/document "body")

; An equivalent zepto call, with strings
(js "document.getElementById(\"body\")")
; OR, if a variable is accessed
(js (++ "document.getElementById(\"" body "\")"))
```

Listing 1: A comparison of the FFI of JavaScript in zepto and 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[5]. Many of the constructs have been adopted by JavaScript by now, both in terms of syntax and semantics. 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.

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

Elm

Elm as it is presented in (Czaplicki, 2012) is a language centered around Functional Reactive Programming (FRP)[6] with a special regard to browser-based Graphical User Interfaces (GUIs). Formulated by Paul Hudak and Conal Elliott in a 1997 paper, it borrows its syntax largely from Haskell and is a pure, functional programming language. Notable features include GUI-centric tooling, such as a ”time-traveling” debugger that caches the state of a program at any given time, allowing for rewinding, forwarding and jumping.

Elm's sole similarity to the language discussed in this thesis is that it is a functional language for the Web. However, 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.

[6]An in-depth explanation of FRP is outside of the scope of this thesis, but it is a programming style (**frp**).

```
-- | Extract the first element of a list (as implemented in the language standard library).
-- | The Haskell version is unsafe and will throw an error
-- | if it encounters an empty list.
head :: [a] -> a
head (x:_) =  x
head [] =  badHead -- an exception function

-- | the list data type in PureScript is safe by design and
-- | the function returns a Maybe monad.
head :: forall a. List a -> Maybe a
head Nil = Nothing
head (Cons x _) = Just x
```

Listing 2: A juxtaposition of a simple function in Haskell and PureScript.

2.2 Existing Standards

When developing a programming language derived from the Lisp family, one builds on top of almost 70 years of development, formalization and research. Of course many of the standards that were formed during that time are now obsolete. Nonetheless, the effort of standardizing the many languages and implementations is still thriving.

Over the years, two main categories of languages have developed: Common Lisp and Scheme. While the syntactic proximity remains, they differ widely in concepts. Macro transformations are written quite differently and continuations are a feature that is exclusive to Scheme.

3 Concept Design

Practicality beats purity.

(T. Peters—The Zen of Python)

The concept of a version of zepto that would run on JavaScript arose rather naturally from the work done on zepto as a backend-agnostic, run-everywhere concept language. Concurrently to the work on a nanopass compiler that targets LLVM and Erlang but supports pluggable backends by design, efforts have been made to bring the language into the browser setting. After preliminary work on a compiler backend that emits JavaScript that was discarded in the early stages of its development for the reasons stated in 1.1, the idea presented in this thesis emerged.

In the following, a brief overview of the design of the concepts used in the port and how they impacted the construction of the system shall be given. A system-based design overview shall be given in 4.

3.1 Construction Design

The value of integrating zepto into existing web infrastructure lies in the addition of a host of new features that are not present natively in JavaScript and hard to implement in a stable manner[1].**Make sentence into two or eliminate clauses; restructure to eliminate intro** The goal of the prototype was thus to expose all of the features that define zepto in the JavaScript implementation as well.

A brief overview over why a Lisp was chosen for this thesis and, more specifically, why zepto, shall be given in the following.

[1] While stability is not guaranteed by the current version of zepto, it is tested and deployed in production systems and shows to function rather reliably.

3.1.1 Lisp

A common saying among programming language designers is that every programmer has written their own implementation of Lisp.[2] There are a lot of different implementations of Lisp in the wild, even ones that compile to JavaScript[3].

The simplicity of the language on a parsing level is often cited as the main reason for its popularity as a language to implement. 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 a handful special forms exist to allow not only for Turing-completeness, but also for expressiveness and elegance.

3.1.2 zepto

As explained in 2.2, Zepto is a new Scheme implementation that aims to be as small as possible, to be able to target a lot of different backends and simplify the process of writing new backend code. Currently, LLVM and Erlang Core[4] 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[5]. 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[6], a backend for the GHC targetting JavaScript instead of native code. It offers many advanced compilation features such as inlining of JavaScript into the Haskell code base using a technique called quasi-quoting, where a

[2]The README of the femtolisp project even goes so far to claim ”Some programmers' dogs and cats probably have their own lisp implementations as well.” (Bezanson, 2016).

[3]such as ClojureScript, the backend for Clojure referenced in 2.1.1.

[4]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. A small paper (**ERL**) that describes its' basics was used to implement the compiler from zepto.

[5]The entire codebase is only about 4000 lines of Haskell code.

[6]The transpiler mentioned in 2.1.1.

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 for how to bring a language into the browser. With both syntax and semantics differing strongly from JavaScript, this example enables languages more closely related to JavaScript to eventually make their way into the browser.

3.1.3 Macros

Macros are a mechanism for rewriting code at compilation time.

3.1.4 continuations

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 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 3: 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.

```
; this would return an integer
(js "1 + 1")

; this would return a hashmap
(js "{key: \"val\""})

; this is problematic, because it will return an object
(js "new Error()")
```

Listing 4: 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.

```
; the function string->number is a standard zepto function
(string->number (js "Math.pow(2, 32)"))

; this is an example of how to resolve the earlier problem:
; override the prototype of the object to return the value that is needed
(js "Error.prototype.toString = function() { return this.message; }")
(error (js "new Error(\"fatal error occured\")"))
```

Listing 5: 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 6: A minimal DOM module

5.2.4 Language definitions

One of the most advanced features are language definitions inspired by the parser macro system in Racket (Flatt, 2011). While this is handled fairly differently in zepto than it is in Racket, the syntax of the resulting language is compatible. Zepto handles the custom parser step with a dispatch at the time of file loading, which enabled the system to be implemented without changes to the core language, completely in zepto itself. While this was so designed with portability in mind and thought to ease the transition to different backends, this proved to actually complicate matters in the case of the JavaScript port, where the module loading system is not present due to the absence of files in the traditional sense.

```
; json-lang.zp
(load "json/json")

(zepto:implements-lang json:parse "json")

; an example for a file that can now be normally loaded
#lang json
{
 "hello": "json"
}
```

Listing 7: An example language definition that allows for inlining of JSON code.

As the detection what to parse is handled from within JavaScript - by the MutationObserver mentioned in 5.2.1 - the first implementation of the dispatch mechanism was written entirely in JavaScript. This proved to make the dispatch mechanism relatively clumsy, because registering additional languages had to be done in JavaScript, thus making this feature a JavaScript rather than a zepto feature. While building a parser dispatch mechanism for JavaScript would certainly also make for an interesting experiment, it is out of the scope of this thesis and would not really qualify as something unique about zepto[4].

[4]The code was salvaged for posteriority and can be found on Github under the name of `js-parse-dispatch`.

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

Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest.

(Isaac Asimov)

* porting efforts
  * compiler efforts
  * zeps
  * classes

8 Conclusion

There have been voices that insisted that the web need be fixed since its inception. Most disagreements over its inner workings have been philosophical and are a matter of ongoing dispute. While the web certainly can be improved - like any system can be, at any time - I do not believe it is fundamentally flawed. The technologies that power the current World Wide Web, from switches and cable technologies to protocols and development frameworks, have shown to be robust enough for a large part of the world to interconnect. We steadily evolve the way we work with it as trends and paradigms emerge and evolve.

One of the major redeeming qualities of the fundamental philosophies of the World Wide Web has certainly been its ability to adapt to changes. Standardization of web technologies has a reputation of being slow, but in terms of design processes on a global scale, it is actually a reasonably fast process.

Zepto is a young language that has adapted this mindset - changes rapidly and a stable release is yet to be announced. With this thesis, an important step towards reaching a fulfillment of its design goals has been made.

Many questions that this thesis addressed remain unanswered. Even with that in mind, I hope that my work on it has asked questions that are worth being asked and that the technologies presented in this thesis will allow for more technologies that come to the web, so that an even more heterogeneous, expressive and inclusive system can emerge, one that caters to different aesthetics, philosophies and mindsets.

I would happily welcome any contributions towards that end, both related to and unencumbered by the technology that is zepto.

References

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

Bezanson, J. (2016). *The FemtoLisp programming language*. URL: https://github.com/JeffBezanson/femtolisp (visited on 01/08/2016) (cit. on p. 11).

Bolz, C. F., A. Cuni, M. Fijalkowski, and A. Rigo (2009). "Tracing the meta-level: PyPy's tracing JIT compiler". In: *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. Genova, Italy: ACM, pp. 18–25. ISBN: 978-1-60558-541-3 (cit. on p. 6).

Czaplicki, E. (2012). *Elm: Concurrent FRP for Functional GUIs*. Senior Thesis (cit. on p. 8).

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*.

Flatt, M. (2011). "Creating Languages In Racket". In: *acmqueue* 9 (11) (cit. on p. 20).

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 pp. 2, 6).

McCarthy, J. (1960). "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". In: *Communications Of The ACM* (cit. on p. 11).

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. 16).

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 pp. 2, 6).