# FYP(!)

Kieran Manning
09676121

December 2012

## Contents

# 1 Acknowledgements

Lorem ipsum

# 2 Abstract: a short summary

This project explores the use of JavaScript as a target language for a Haskell EDSL. Program development in JavaScript is difficult due to various weaknesses in the language design. Rather than attempting to fix the flaws in the language we will explore the practicality of using JavaScript as a target language for developers who wish to work in the higher-level Haskell language. Rather than translating Haskell programs to JavaScript we provide an interpreter which can execute Haskell programs that have been translated to a core language; by taking this approach we are able to discuss the preservation of the program semantics with more confidence than in a direct-translation approach.

# 3 Goal

The goal of this project is try and improve on some of the problems inherent in writing programs directly in javascript by providing a secondary language into which javascript can be compiled. Javascript can produce perfectly acceptable client-side browser-executed web programs. In practice however, it is difficult to write Javascript which produces such perfect results. Javascript suffers from a number of problems. It has a very weak, dynamic type system which results in a lot of reliance on type coercion and programmer sanity. Unpredictable results which arise through this weak typing are frequent causes of failures and bugs in javascript programs, leading to its notoriety as a difficult language to write safe programs in, especially for programmers new to the language.

The syntax of javascript is verbose and arguably unpleasant, the product of a different age of language design. As such, learning to use javascript can be quite painful, tedious and counter-intuitive for the first-time programmer. This problem persists even for more experienced users, in the difficulty inherent in trying to find bugs or errors in failing javascript or attempting to become familiar with an existing javascript codebase. A number of other languages have been created in an attempt to solve this problem, a prime example being CoffeeScript, which is discussed further in section 5.4. The product of this project should aim to provide a more user-friendly syntax, inspired by the improvements in language design since the introduction of javascript and, which will be easier to write and read for experienced and beginning programmers alike.

We would also hope to introduce the notion of lazy evaluation into client-side programming with javascript in an accessible, straight forward manner. It is possible to write programs in a lazy style with javascript by viewing all evaluations as 'thunks' represented by javascript functions, but in practice this is an uncommon approach. Achieving this, however, requires what is essentialy creating a lazy domain specific language in Javascript and adapting your programs around this. Libraries such as stream.js have tried to achieve this in part. Lucky for us, adding such functionality becomes substantially easier when using javascript as a target language, allowing us to view javascript and a lazy DSl as the 'runtime' for our language.

If we are going to the trouble of implementing all of this using haskell and GHC it would be nice to get some benefit from our work! That is why I've opted to use GHC's Core intermediary language as an intermediary in our own language. By the time a program reaches the Core 'stage' of its compilation, GHC has already checked its validity at the type level. By

using core as our intermediary, we can take advantage of the type system and optimizations already implemented in GHC to do some of the heavy lifting for us.

# 4 Introduction

It would help to explain some of the concepts to which I will be referring throughout this report. Not everyone will be familiar with some of the functional programmming, and more specificially Haskell, ideas being discussed. While many programmers will have an idea of the workings of javascript, at least in so far as it is an imperative language, some of the concepts which we will be examining might go beyond the scope of casual javascript programming. For brevity's sake, I will explain some of the more specific concepts which I will be commonly referring to throughout this report.

## 4.1 Javascript Typing

There are seven data types in javascript, five of which concern us for the purposes of this project; three primary data types which are Number, String and Boolean, and two 'composite' data types, Objects and Arrays. When writing in straight forward javascript, types are more or less invisible to the programmer. To declare a variable, of any type, the usual format is...

```
var varname = value;
```

The type of varname is inferred from the type of the value. Types can be mixed in operators and functions with some type coercion, which can be very dangerous if used incorrectly. The composite Object type will be most interest to us later, when dealing with the javascript 'runtime' used both in our own implementation and in that of the Fay language, which we will be looking at.

An important point to remember, is that we can represent primitive objects as composite objects, albeit it with a performance penalty. This idea will prove significant later when dealing with lazy evaluation, where simple primitive types will be insufficient to represent lazy evaluation.

## 4.2 Lazy Evaluation

Lazy evaluation, also known as call-by-need evaluation, is an evaluation method that delays function evaluations until such time as their end values are directly required. The result of this is that a computation we write will only be evaluated when it is required by some other aspect of the program, or possibly never if its value is never required.

This is useful both from an efficiency point of view and when trying to represent concepts which might not fit into stricter, more finite ideas of programming.

For the former, it is easiest to demonstrate with an example...

```
take 1 ['a'..'e']
```

Here, *['a'..'z']* represents a list of characters beginning with 'a' and ending with 'e'. The function *take* evaluates and returns the number of elements of the second parameter specified by the first parameter, starting at the head of the list. In this example we're 'taking' one element from "abcde" which of course gives us the character 'a'. But what if we were to say...

```
take 1 [1..]
```

As you may have guessed, *[1..]* is the list of integers beginning with 1 and continuing to infinite. Strange concept? In strict evaluation, attempting to make use of this would be difficult, to say the least. However, with lazy evaluation we only need to evaluate as much of the list as we actually need. The result being that our take function will only evaluate and return the first item, ignoring the remainder of the list. What's more, it can do this as quickly with an infinite list as with any other length list. This is where the efficiency bonus comes in!

Of course, in order to represent lazy evaluation, we need more than just primitive types. A 'thunk' is a delayed computation, formally a *parameterless closure* which represents a computation that is not evaluated until required or 'forced'. We can view all our expressions as thunks, or series of thunks, which have yet to be forced or evaluated.

TODO: Quick overview of general lazy compilation. Alpha substiuititition, ASTs, graph reduction

### 4.3 Weak Head / Normal Form

*Normal form*, and *weak head normal form* or WHNF, are terms used in this context to denote the level of possible evaluation in an expression or thunk. An expression is in normal form when it can be evaluated no further, for example...

```
1
\x -> x
(1, 2)
```

are an integer, lambda expression and tuple respectively in normal form. "1 + 1" by comparison would not be in normal form as their is an addition operation to be performed. Expressions in weak head normal form are expressions that have been evaluated to the outermost data constructor. This is easier explained using an example

```
(1 + 1, 2 + 2)
'h' : ("e" ++ "ello")
```

In the first line,

# 5 Problem Space

## 5.1 Syntax

JS syntax is hella pants. Able to fix this though!

- Generally awkward, verbose
- JS is Old. New ideas on syntax have emerged
- Succinct simplicity of syntax in languages such as Python as contrast
- Overview of ease of writing DSLs in haskell as a solution

## 5.2 Weak Typing

- Dangers of type coercion 7 + 7 + "7"; // = 147 "7" + 7 + 7; // = 777 . Common complaint and cause of errors, much like PHP.
- Dynamic (runtime) vs. static (compiletime). Discussion on HindleyMilner.
- How much of this we can represent in Lambda Calculus/SystemF
- What do we lose by going static?

## 5.3 Late Binding

Efficiency, runtime vs. compile time typing

# 6 Existing Solutions

I am not the first person to examine the possibilities of client-side web programming with Haskell.

## 6.1 Fay

The Fay language, which lives at https://github.com/faylang/fay/wiki, is another language which has attempted to solve some of the problems we are interested in. It provides a Haskell subset DSL to programmers, taking advantage of existing Haskell syntax. The haskell FFI, or Foreign Function Interface is then used to connect this to a 'runtime' written in javascript. This runtime consists of javascript functions representing functional and lazy computations ie. thunks and variably saturated functions. This allows Fay to translate Haskell programs into Javascript while preserving notions such as laziness and purity, while also making use of Haskell's type system resulting in a strongly typed language.

Javascript types are represented in the DSL through the use of 'Fay' types, for example...

```
getEventMouseButton :: Event -> Fay Int
getEventMouseButton = ffi "%1['button']"

focusElement :: Element -> Fay ()
focusElement = ffi "%1.focus()"
```

Here, Fay Int is a type defined in the Fay subset which represents a javascript primitive integer. This will be compiled into an equivalent javascript function. Using a Haskell DSL allows Fay to make use of Haskell's type system and strong typing to produce equivalently safe code in Javascript. It also allows the programmer to benefit from many of the benefits that come with programming with Haskell, type signatures being of particular note in this example. The second snippet is included to show an example of representing a non-returning function in Fay. FocusElement will take an Element type and return no value but will execute a Fay action, which will correspond to some javascript function.

Fay's javascript 'runtime' uses functions to represent objects. In this manner, the idea of *first class functions* can be preserved from Haskell in the translation to Javascript. A 'thunk' for example looks like this...

```
// Thunk object.
```

```
function $(value){
  this.forced = false;
  this.value = value;
}
```

We can see Fay making use of javascript's concept of all-encompassing objects to make representing functional code easier.

```
function Fay$$mult(x){
  return function(y){
    return new $(function(){
      return _(x) * _(y);
    });
  };
}

function Fay$$mult$36$uncurried(x,y){
    return new $(function(){
      return _(x) * _(y);
    });
}
```

There are seven data types in javascript, five of which concern us for the purposes of this project; three primary data types which are Number, String and Boolean, and two 'composite' data types, Objects and Arrays. When writing in straight forward javascript, types are more or less invisible to the programmer. To declare a variable, of any type, the usual format is...

```
var varname = value;
```

The type of varname is inferred from the type of the value. Types can be mixed in operators and functions with some type coercion, which can be very dangerous if used incorrectly but that's a discussion for a different part of this report. When discussing Fay, we really only care about the composite Object data type. This is an object in the usual sense, with attributes and associated methods.

Representing data in this manner allows for the concepts of Laziness and currying to be carried over from Haskell and represented in javascript. Of course this could be written manually in javascript without the need for Haskell or a translator, but such programming would be tedious and error-prone. A disadvantage to this method is a loss of efficiency. This

would not be the intended manner of programming in javascript and the language is not necessarily optimized to deal with it. It also produces a larger output of code than programming in a more traditional imperative javascript manner. This is of more signficance in web languages such as javascript where a larger body of code will take longer to transfer from the server to the client/browser, slowing page load times.

* A proper syntactic and semantic subset of Haskell * Statically typed * Lazy * Pure by default * Compiles to JavaScript * Has fundamental data types (Double, String, etc.) based upon what JS can support * Outputs minifier-aware code for small compressed size * Has a trivial foreign function interface to JavaScript

### 6.2 iTasks? Relevant?

### 6.3 GHCJS

### 6.4 CoffeeScript

Worth mentioning from point of view of non-functional take on the js problem

## 7 Propose solution involving Haskell

- Discuss type safety, laziness, ease of haskell/JS interoperability.
- Compare our core-level output ideas with existing solutions. Include quick chats core/STG in general. Link to SPJ paper(s). Actually, maybe keep chats for later.
- Worth comparing to ClojureScript

### 7.1 FFI Method

### 7.2 Intermediate language

# 8 Description of implementation and choices

## 8.1 A subset language as a Haskell DSL

This will

## 8.2 Syntax

One of the earlier considerations for this project, by virtue of the natural order of implementing features in $LANG, was the syntax of the developer facing input language (the "language" itself really). I knew from the outset that the syntax used in javascript was not something I wished to reinvent, seeing it more as a problem than an inspiration.

The syntax in javascript feels like a relic of a different time. It is a common problem brought up by programmers new to the language, and an accepted hindrance for anyone more experience with javascript.

Keywords such as 'new', 'var', and 'function' serve as examples of ideas on syntax design which have become far less common since javascript was created. 'var', for instance, adds verbosity to the language while providing little benefit. Seemingly inspired by Scheme's 'define' concept, it adds additional cruft to declaration statements which are otherwise self-explanatory in languages with more minimal syntaxes such as Python. A declaration can be inferred sufficiently from a simple "x = y" statement and its context. 'function' as a keyword also seems to have been inspired by Scheme, in this case its liberal use of the 'lambda' keyword to represent anonymous functions. Languages such as Haskell dont overcomplicate their function defintions, content with statements as simple and declarative as 'f x y = x + y', as well as internally scoped (fix that terminology when less tired) functions using the where keyword etc. Type signatures are optional but do serve to make the code in question more readable and solve some programmer headaches (from a syntactical point of view, they obviously have other significant semantic effects).

Both Fay and CoffeeScript went the minimal, Pythonic route with syntax design. Statements such as...

- Variable declarations : "$number = 42$"
- Function declarations : "$square = (x) \rightarrow x * x$"
- List comprehensions : "$cubes = (math.cubenumfornuminlist)$"

...in coffeescript contrast sharply with the parentheses, "function" and semicolon ridden expanse of comparable javascript. The Fay approach is similar in its brevity.

My approach to syntax design would be influenced by these ideas. They are reflected in the simple declarative statements included in my language, and were I to continue this project beyond its current scope the same concepts would continue to be applied.

### 8.3 Core/STG

link to SPJ etc. papers. Talk about SAPL and the iTasks project.

### 8.4 Representing laziness?

Fay, thunks-as-JS-functions

### 8.5 Flow control

case statement continuations, link to paper

### 8.6 Higher order functions