University of Dublin

TRINITY COLLEGE

FYP

me

Ba(mod) Cookery Studies
Final Year Project, April 2012
Supervisor: Glenn Strong

School of Computer Science and Statistics
O'Reilly Institute, Trinity College, Dublin 2, Ireland

Contents

1 Abstract

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.

2 Acknowledgements

Lorem ipsum

3 Goal

The goal of this project is to try and improve on some of the problems currently inherent in writing browser executed programs using some of the common features of functional languages as inspiration. The ubiquitous standard programming language for writign browser side web programs is javascript. 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. Javascript is an interpreted language with weak/dynamic typing. Its syntax is inconsistent and verbose. It provides no support for lazy evaluation.

Javascript supports weak, dynamic typing. Types are inferred when the program is executed. The term 'duck typing' is used to describe such typing, where the types of objects are inferred based on their attributes. This means that programmers need not concern themselves with annotating the types of their functions and variables in Javascript, which some see as a plus. This advantage however is greatly offset by the ease with which it can be misused. Misunderstanding of javascript's typing frequently results in unexpected behaviour and bugs which can only be diagnosed at runtime.

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.

Javascript is strictly evaluated with no provisions nor means for writing lazily evaluated programs. Lazy evaluation is an evaluation strategy consisting of call-by-need evaluation as well as *sharing*. Call-by-need evaluation allows for the evaluation of computations to be delayed until such time as their values are needed (if ever). Sharing is a technique where expressions are overwritten with their evaluated value after their first evaluation meaning that the expression need not be evaluated in future when its value is needed, improving efficiency. These concepts are common in functional languages such as Haskell, Miranda etc. and allow for improved efficiency compared to equivalent strictly evaluated expressions in certain circumstances. They also allow us to make use of interesting concepts like infinite lists, better explained later.

These are problems which have long been identified in Javascript. Languages such as Haskell, have shown that they can be solved albeit in a different programming domain. Other languages such as CoffeeScript and Fay have attempted to fix some of these problems in the domain of web programming. We shall examine such existing solutions to this problems with the hope of using them as inspiration for functional solutions to some of the problems of browser side programming.

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 Types in Javascript

Is this really needed?
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. The types of javascipt objects are inferred from their attributes and methods. Types can be mixed in operators and functions with some type coercion, which can be 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 Functional Languages

4.3 Functional Compilation

I will be referring frequently to some of the more common ideas and practices invovled in compiling a functional language thoughout this report. Unsurprisingly, the field of functional compiler design is not easily summarized in a paragraph, let alone a paper; the implementation explained here only scratches the surface. However, it will be easier to understand some of the points I make if we start with a quick overview.

Most compiler implementations for simple functional languages start with an input language which consists of a list of function definitions and some entry or "main" function, which serves as the entry point for the run-time or evaluator. We wish to avoid writing programs with side effects and as such the syntax of our input program will reflect that. There is no concept of global state nor stateful computations and when we need to represent some concept of changing state we do so by passing state as paramaters between functions.

The input language is parsed into a simplified expression-based representation. One such representation, the one used later in this report, is defined as:

```
Program ::= [EVar, [a], Expr a]

Expr a
  ::= EVar Name
    | ENum Int
    | EConstr Int Int
    | EAp (Expr a) (Expr a)
    | ELet
        IsRec
        [(a, Expr a)]
        (Expr a)
    | ECase
        (Expr a)
        [Alter a]
    | ELam [a] (Expr a)
```

We say that a program in this representation consists of a list of super combinator definitions. A super combinator, for our purposes, is an expression body along with an identifying variable name and a list of arguments to the expressions. Expressions can consist of

1. EVars. Variable names corresponding to a defined value.
2. ENums. Primitive integers
3. EConstrs. Data constructors, along with the number of elements in the fully saturated constructor as well as the constructors tag, used to differentiate constructors in a given context.

4. EAp. Binary function applications which we use to apply exprs in order to construct larger exprs.

5. ELet. Let expressions, allowins us to define a list of arguments local to a specific expression.

6. ECase. Case expressions, containing first a condition expression which will evaluate to a tag identifying a data constructor in the context of the case expressions and secondly, a list of constructor tags and the code to be executed when a particular tag is found.

7. ELam. Lambdas, or anonymous expressions. Note the similarity to a super combinator, without the identifying variable name.

By way of example, a sample program taking two paramaeters and returning their sum, written in such a language, would resemble

$$["sum"], ["x", "y"], EAp(EAp(EVar"+")(EVar"x"))(EVar"y")$$

Note in particular that we address a supercombinator named "+" representing our integer addition primitive and that we use two EAp binary expression applications in order to call it on two parameters x and y.

Our compiler will take a program in such a simplified representation and convert it into an initial state representing our input program and the means to evaluate it to a final state. Depending on the compiler implementation in particular, this may consist just of an initial state or an initial state and a sequence of instructions denoting actions to be executed on the state. An interesting distinction between these compilation schemes and those of imperative languages is that we can view our initial (and continuing) program states as graphs built from a small set of nodes, namely

1. Integer nodes, representing literal integer values

2. Function nodes consisting of the arity of the function and the its behaviour along with the instructions needed to evaluate it.

3. Application functions, the only nodes capable of forming node connections from a graph point of view and used to represent expression applications in terms of our simplified expr language.

4. Data Constructor nodes, consisting of the arity of the data constructor and a list of addresses to the nodes containing its elements.

We say that our program can be represented by a graph of reducible expressions or *redexes* built from the above nodes. Our program state will consist of a heap containing the nodes which form or graph addressable by their address in the heap. We will also have a stack used to contain a working set of heap addresses which will be required in the evaluation of immediately pending expressions. A 'globals' dictionary will associate global function identifiers with the addresses of their function nodes in the heap.

GIVE EXAMPLE OF GRAPH
EXAMPLE OF REDEX REDUCTION

An evaulator or runtime evaluates our compiled program by iterating through states from the initial state to the final. On each iterartion, the state (and instruction sequence if present) are examined to determine the actions to execute in order to reach the next state. The appropriate actions are executed, producing a next state which we will then evaluate in the same manner.

4.4 Lazy Evaluation

Lazy evaluation, also known as call-by-need evaluation, is an evaluation method that delays evaluations until such time as their values are 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! **Is that right?**.

EXPLAIN IMPLEMENTATION RE: OUR GRAPH REDUCTION

A second characteristic often associated with lazy evaluation is that of updating. When an expression is evaluated in a lazy context, and we know that the contents of the expression will not change, we can overwrite the expression with its final value. Implementation wise, this generally involves replacing the pointer we previously associated with the expression in the heap with a pointer to an indirection node pointing to the result of evaluating the expression. From then on, if our program attempts to access the expression, it will be redirected to a value in the heap representing the evaluated expression. This means we need only evaluate the expression once.

This chractecteristic, known as sharing, is closely related to the idea of *referential transparency.* Referential transparency is the name given to a property of certain expressions which states that the expression can be replaced with its value without altering the semantics of the overall program. This property exists in programs which are *pure*, that is to say programs consisting of expressions or functions whose evaluation is dependant only on the body of the expression and it's parameters, and which cause no side effects which may effect other expressions upon execution. We cannot update an expression with its value if there is a danger that some aspect of the expression may change during the course of program execution. This creates certain challenges when dealing with non-deterministic concepts such as IO in functional languages. Such problems however are beyond the scope of this project and implementation. Those interested can find more information here **LINK**

4.5 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. We say an expression is in head normal form when it has been fully evaluated or reduced, and can neither be further evaluated nor contains any sub expressions which can be further evaluated. 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 there is an addition operation to be performed.

Expressions in weak head normal form are expressions that have been evaluated to the outermost data constructor. These can be data constructors, undersaturated primitive operations or lambda abstractions. For example...

```
Just (1 + 1)
\x -> 2 + 2
'h' : ("e" ++ "ello")
```

The first example contains two sub expressions which could be further evaluated, but as the outermost component (the Just data constructor) has been evaluated fully, it is still in WHNF. The same is true of the 3rd example, where the "++" sub expression could be further evaluated, however the outermost element ':' is a data constructor, specifically the list cons constructor. The second example is in WHNF by virtue of being a lambda abstraction.

REDEX EXAMPLE

5 Problem Space

An important aspect of web development is the ability to write dynamic websites which will adapt to users requirements, input etc. This can be achieved by writing adaptive client-side or server-side web programs which serve dynamic content. Client-side browser executed programming has become increasingly important as it is more scalable and allows for truly dynamic content without the need for page refreshing. The ubiqitous standardized programming language for client-side is JavaScript. This is the only language which can be executed on all major browsers and that seems unlikely to change in the forseeable future.

JavaScript was created in 1995 for Netscape by Brendan Eich. It is an object-orientated, imperative, interpreted language intended originally to be executed by a web browser (although server-side implementations also exist, notably Node.js in recent years). Despite the name, JavaScript is unrelated to Java. They both share a similar `C++`-inspired curly-brace syntax and object orientation but that is about as deep as the simalirities go. JavaScript is weakly, dynamically typed. Its objects are not class based but prototyped based. It is interpreted, meaning that the human readable source code is executed by the browser without any pre-processing.

Some features have been added to JavaScript over the years but in general the language has not greatly changed since its inception. This is understandably problematic. As of this moment of this writing the language is 18 years old. Few technologies can claim to have remained in use that long in this industry and fewer still without detractors. As web development and websites have become increasingly more complex problems inherent in JavaScript have started to become more obvious. JavaScript's syntax is unappealing and verbose. Its type system is weak in more than name alone, leading to frequent problems with ill- written JavaScript programs breaking silently. Many such problems would have been detected or prevented in languages with stronger type systems. JavaScript lacks any concept of a module system making inclusion difficult and naive. Programs are "included" in web pages by linking to their sources which are basically concatenated by the browser executing the page. Such issues existed from the first release of the language. As the websites and the JavaScript programs on which they rely have grown larger, unimaginably so since 1995, these issues have become increasingly problematic.

Many people have attempted to write alternative languages which could replace JavaScript but introducing new standards in the hopes of replacing old standards usually ends as predicted. As JavaScript began as the *the* *standard*, so it seems it shall continue. There is some light to the situation, however. Much time and effort has been put into making JavaScript run fast and it is capable of great efficiency. While JavaScript may be our only client side programming language, at least it *does* provide us with a standard upon which all major web browser vendors will agree. There have been attempts to improve JavaScript by augmenting its feature set through the use of user created libraries. One notable example is JQuery, a JavaScript library which adds new features and alternative syntax to the language. These are still only addons though, relying on the underlying semantics and executing of the original language.

5.1 Typing

JavaScript is weakly, dynamically typed. As it is an interpreted language, there is no notion of static, compile-time typing. The types of variables and objects are inferred at runtime based on their values, or attributes and methods in the case of objects. This typing style is referred to as *duck typing* and similar is used in other languages such as Python. The consequences of this typing lead to some of JavaScripts notable characteristics. Firstly, programmers do not need to concern themselves with annotating the types of their objects, functions etc. when declaring them and the syntax relfects this. The statement...

$$\text{var } x = 2;$$

...will, when evaluated at runtime, create a variable named x, infer it to be of type int based on the value assigned to it and assign it the value 2. Objects and functions are declared in a similar fashion, with their attributes and behaviour used to determine their types. This runtime inference exemplifies the dynamic nature of javascript typing. It also means that type errors can only be diagnosed at runtime, generally with unhelpful error messages. A strongly typed language would be capable of finding such errors at compile time. This is a somewhat imperfect argument as JavaScript is an interpreted language, however strong typing would make it easier to diagnose errors at runtime and allow us to perform useful type checking ahead of time if we wished.

We can see the effects of JavaScript's weak typing in its type coercions. Implicit casting occurs frequently in JavaScript programs. It could be argued that this is a useful convenience feature, though in practice such coercions can be vague and unintuitive. One such example is arithmetic in the presence of strings. If we call an arithmetic operator on N values, one or more of which is a digit string, they will be cast to numerals and the operation applied, returning a numeral result. for example

```
"2" * "2" => 4
 2  * "2" => 4
```

Calling the same result on non digit strings will return a NaN and program execution will continue (probably breaking soon). The + operator is even more interesting, as it is also overloaded as a string concatenation operator. Using + on numeral values will return a numeral value. Using it on some combination of numeral and string values however will break addition associativity:

```
("x" + 1) + 2 => x12
 "x" + (1 + 2) => x3
7 + 7 + "7" => "147"
"7" + 7 + 7 => "777"
```

It is very common for bugs to arise in JavaScript programs where variables have been implicitly cast to unexpected types. The resulting program will probably break with a completely unrelated error when some function or operator chokes on an unexpected, unintended value. Worse yet, the program may break silently and end up in production with an undiscovered bug.

JavaScript is also overly forgiving of type errors when they do occur. One such example is shown above, in the addition of non digit strings resulting in a NaN return. Another more worrying example is the Infinity numeric value, which occurs when a value goes outside the bounds of a floating point number. Much like our NaN example, the program will continue to run until it chokes on this Infinity value. This permissive behaviour along with javascript's weak typing and implicit (often unintuitive) casting makes it unfortunately easy to write programs which exhibit unintended behaviour with non-existant or silent errors.

5.2 Syntax

When javascript was created, its syntax was intended to resemble that of C and Java. At the time, these were two predominant languages and reusing ideas from their syntax design was intended to lessen the learning curve for programmers coming from C and Java backgrounds. Since 1995 many new ideas for syntax design have appeared in more recent languages.

JavaScript's syntax is awkward and verbose in places. Nested anonymous functions for example can quickly become ugly, requiring careful curly brace placement and indentation to remain vaguely readable. The use of curly braces, parenthesis and semicolons to seperate and sequence statements allows for horribly ugly, executable code. Languages such as Python and Haskell have found ways to overcome these problems through the use of whitespace and significant statement placement. Their forced coding styles produce cleaner, more standardized code which is more readable with less fluctuation from programmer from programmer.

A more significant problem with JavaScript can be its handling of operators. The `+` operator described above is a good example. By default, the same symbol is used for string concatenation as for arithmetic addition. Operator overloading is a matter of opinion, though overloading operators as common as `+` by default is probably not the wisest or most intuitive choice.

5.3 Lazy Evaluation

Javascript is a strictly evaluated language. This isn't a problem per se and and there is no "better" choice between lazy or strict evaluation. However, the option of lazy evaluation in browser side programming would be nice.

Lazy evaluation has shown itself to be useful in languages such as Haskell. Firstly, it allows us to make use of concepts such as infinite data structures. We could for example, create an infinite list of items forming a recurring pattern and take as many items as we wish from this list. Such operations in JavaScript are not possible. The syntax does not exist to allow ease of creation of such data structures and even if it were possible, we'd hit the obvious problem of trying to represent such structures in a strictly evaluated language. Lazy evaluation also provides for improved efficiency under certain circumstances. Lazily evaluated expressions are not universally faster than their strict counterparts but in many lazy languages the option exists to perform operations in a strict or lazy context, allowing the programmer to choose the better evaluation strategy for a given task. It would be great if we could bring similar flexibility to browser-side programming.

6 Existing Solutions

We are not the first to identify these problems, nor to attempt to rectify them. Much research and work has gone directly into studying issues in JavaScript. Many of the problems of JavaScript also exist in the general programming domain where solutions and alternatives likewise exist. Having looked at some of the problems in JavaScript and its place in browser side web programming, we will now examine some of these existing solutions.

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 DSL with primitives corresponding to those of JavaScript as well as various built-in methods to aid in browser side programming. Programs are written in this DSL then compiled using the Fay compiler into a javascript representation thereof. The choice of Haskell as the platform language gives Fay a number of useful properties. Programs compiled with Fay are:

- Type-safe
- Pure
- Lazy

The choice of JavaScript as the target language means that programs compiled with Fay are also compatible with all major web browsers. In order to evaluate compiled Fay programs, a number of primitive operations are needed. These are provided in a seperate runtime written in JavaScript.

6.1.1 A Haskell DSL

Fay source programs are written in Haskell, making use of a number of primitives defined in a haskell embedded domain specific language. In general, there are a number of advantages to using domain specific languages to accomplish tasks. Such languages can take advantage of the tools and features of their platform language. This greatly reduces the time taken to design and implement the language itself, as the great bulk of the necessary compilation tools probably already exist for the platform language. The domain specific language (DSL) will typically use constructs in the platform language to represent the data which we wish to work it. Methods will be provided to execute common actions on these representations allowing us to process the data our DSL represents.

The Fay domain specific language provides a number of important constructs. Most significantly, it provides access to the Fay () monad which represents Fay computations. Functions which are intended to interact with the browser make use of this monad to execute Fay operations. For example, the hello world function in Fay might look like this:

```
main :: Fay ()
main = alert "Hello, World!"

alert :: String -> Fay ()
alert = ffi "window.alert(%1)"
```

The type signature of the function *alert* shows this in action. We that the alert function takes a String and returns a Fay () action. In this case, the Fay action is a foreign function interface call to JavaScripts window.alert() method. Fay also contains a list of type definitions corresponding to the JavaScript types into which our Haskell and Fay types will be compiled. In this way, primitive Haskell types and Fay types can be mixed in Fay source programs.

This approach allows the programmer to take advantage of many of Haskell's existing features when writing Fay programs. The syntax remains the same, with the addition of a few keywords and constructs added by Fay. Someone coming from a Haskell background with seperate web developmente experience would have little difficult getting up to speed with Fay, once they figure out the Fay specific additions in the DSL. As Fay is created within Haskell, it is able to take full advantage of its type system. This means that Fay programs are type-safe. It also means that error messages and warnings from Haskell compilers can be used when debugging Fay programs. This is a significant improvement over writing JavaScript where debugging errors in such a weakly typed language tends to be tedious and vague.

6.1.2 Runtime

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

6.5 ClojureScript

7 A High Level Design

7.1 Overview

Having looked at the problems we wish to solve and some existing solutions, we now turn our attention to the implementation of our own solution. First, we form an image of the overall architecture of our solution. At a very abstract, high level we can say our project will require the following:

1. An input language rich enough to allow us to write programs that capture our intended semantics and features.
2. A transformation from this language into the actions we wish to be executed by a browser.

The latter requirement can be decomposed into a number of sub-requirements. We will need to split our transformation firstly into compilation and runtime phases. A source program written in our input language will first be compiled into some runnable representation thereof by a compilation stage. This representation of our program will then be executed by our runtime until a value is returned. We can be somewhat vague for the moment about the exact details of the input language and compilation phase, but we do know that our runtime must be capable of executing programs in a web browser. We could say trivially that we only need to compile our programs in advance to simple end values which could be interpreted by a browser, but this would leave us with a glorified static expression evaluator. Thus, our runtime must be executable from a web browser. With this in mind, we can say that it is the job of our compiler to take our input language and transform it into a representation which our runtime can then execute when called from a browser.

Seperating our compiler and runtime poses a small problem. We now have two disparate stages which will need to communicate in some sense. We could write both our compiler and runtime in a language which can be executed by a browser, but this would be unnecessary and an inefficient use of any such language. Instead, we should write our compiler in a more general purpose language and find a means of converting our compiled representation into one which can be understood by our runtime. Let us now re-examine our project requirements.

1. An input language rich enough to allow us to write programs that capture our intended semantics and features.

2. A compiler capable of transforming our input language into a program representation executable by our runtime.

3. A means of converting our compiled representation into one which can be executed by our runtime.

4. A runtime to execute our program in a browser.

7.2 Input Language

**Note we haven't commented on the format of core programs beyond **
*recognizing them at a glance. Need to go into detail on [ScDefn]**
*and others, although we could probably leave that until impl **

We will start by examining the first requirement. We will need a language that is capable of conveying at the very least the generic basics of a programming language, namely the notions of primitive value and function declarations and function applications. After this, our language will need some means of flow control, namely conditionally executed statements and loops. The last of these basic required features is some manner of stuctured data type, allowing for the creation of more complex types as composites of primitive types.

Our resultant compiled language will need to type-safe and support lazy evaluation. The former will require some variety of type-checking to be performed on our input language at some stage during compilation, and any type errors found to be dealt with and report. The latter is more of a compilation concern than one one of language choice, although a pure input language which we can guarantee side-effect free would be of great help.

At this point, we can start to see certain parallels with our ideal input language and currently existing programming languages. What we have described looks a lot like the generic template for functional languages such as Miranda or Haskell, with a very simple feature set. It could also be compared to classic LISP with the addition of structured data types.

On a more interesting note is the similarity of our required language to certain subsets of Lambda Calculus. Before examing these subsets, we'll take a quick look at some simpler dialects. The basic features of simple untyped lambda calculus are lambda terms, denoting well formed expressions in the lambda calculus which can consist of:

- Variables
- Lambda abstractions

- Lambda applications

where the latter two correspond with function abstractions (or definitions) and applications in more familar terminology. The lambda term representing a function that takes two parameters and returns their sum would be

$$\lambda xy \to x + y$$

The typed varieties of lambda calculus add the concepts of types and type notations, updating the syntax with a new construct $x : \Gamma$ indicating a variable x of type Γ. This would be the only significant difference in the simply typed dialect.

Let us now look at a variant of the language known as System F. This dialect adds the notion of type polymorphism or universal type quantification to the simply typed lambda calculus. The effect of this is to allow for the use of variables which range over types as well as functions, in comparison to the symply typed lambda calculus where variables only range over functions. In practice this allows us to reason about the types of functions, which also provides us with the ability to write functions which range over universally quantified types. As an example, we can express the identity function as

$$\Lambda\alpha.\lambda x^{\alpha}.x : \forall\alpha.\alpha \to \alpha$$

which we read as *the type level function which takes the type α and returns the id function $\lambda x^{\alpha}.x$ (which is of type* function that takes a paramater of type α and forall type α returns a value of type α*)*, where the id function can be read as *the function that takes an x of type α and returns same.* Those interested will find further information on System F here LINKLINK

The reason System F is so intereesting to us is its similarity to a number of functional language representations currently in use. Many high-level functional programming languages can be expressed in terms of System F while still preserving their full semantics. Of particular note however are the languages of Core and SAPL. These are intermediate languages used as minimal representations for Haskell and Clean respectively which can be emitted by specific compilers for each language midway through compilation. The term 'Core' is often used to refer to any such intermediate functional language based on typed lambda calculus, but from here on we shall refer to the GHC (Glasgow Haskell Compiler) specific intermediate language as Core or 'External Core' and similar languages as 'Core-like'. Core is a lambda calculus dialect (System F with added type coercions to be specific) which

can be obtained from GHC with the compiler argument -fext-core. SAPL can be similarly obtained from the Clean compiler.
The semantics of Core look like this..

```
type CoreExpr = Expr Var

data Expr b
  = Var   Id
  | Lit   Literal
  | App   (Expr b) (Arg b)
  | Lam   b (Expr b)
  | Let   (Bind b) (Expr b)
  | Case  (Expr b) b Type [Alt b]
  | Cast  (Expr b) Coercion
  | Note  Note (Expr b)
  | Type  Type
```

...when expressed in terms of a Haskell algebraic data type (which coincidentally is a nice way to express most things in life). Lam in this case stands for Lambda and the liberally added 'b's for the types of the binders in the expression, the rest is mostly self explanatory. Note that data constructors are not represented here although are included in the language seperate to the Core expr type, along with a %data denotation.

By the time a source program reaches the stage of GHC where it can be emitted as Core, it has already undergone type checking. It has also been substantially minimized to the simplest representation possible still perserving all of the original program's semantics. This makes Core an excellent candidate as an input language, as its minimalism makes it easy to parse and allows us to make assumptions about its type safety.

To give an idea of what a Core program looks like, we'll examine a small example. A factorial program is the canonical functional Hello World and will demonstrate a few important concepts, so lets go with that. Our source program will be

```
module MIdent where

fac :: Int -> Int
fac 0 = 1
fac x = x * (fac x - 1)

main = fac 4
```

Compiling this with ghc -fext-core produces the following output...

```
%module main:MIdent
  %rec
  {main:MIdent.fac :: ghczmprim:GHCziTypes.Int ->
                      ghczmprim:GHCziTypes.Int =
     \ (dsdl0::ghczmprim:GHCziTypes.Int) ->
      %case ghczmprim:GHCziTypes.Int dsdl0
      %of (wildX4::ghczmprim:GHCziTypes.Int)
        {ghczmprim:GHCziTypes.Izh (ds1dl1::ghczmprim:GHCziPrim.Intzh) ->
           %case ghczmprim:GHCziTypes.Int ds1dl1
           %of (ds2Xl6::ghczmprim:GHCziPrim.Intzh)
             {%_ ->
                base:GHCziNum.zt @ ghczmprim:GHCziTypes.Int
                base:GHCziNum.zdfNumInt wildX4
                (base:GHCziNum.zm @ ghczmprim:GHCziTypes.Int
                 base:GHCziNum.zdfNumInt (main:MIdent.fac wildX4)
                (ghczmprim:GHCziTypes.Izh (1::ghczmprim:GHCziPrim.Intzh)));
                (0::ghczmprim:GHCziPrim.Intzh) ->
                 ghczmprim:GHCziTypes.Izh (1::ghczmprim:GHCziPrim.Intzh)}}};
  main:MIdent.main :: ghczmprim:GHCziTypes.Int =
    main:MIdent.fac
    (ghczmprim:GHCziTypes.Izh (4::ghczmprim:GHCziPrim.Intzh));
```

Which is intended to be partially human readable, but mostly just parseable.
Lets tidy this up a bit...

```
%module main:MIdent
%rec
{fac :: Int -> Int =
\ (d0::Int) -> %case Int d0 %of (wildX4::Int)

{Types.Izh (d1::Int) -> %case Types.Int d1 %of (d6::Int)

{%_      ->
    * @ wildX4 (- @ (fac wildX4) 1::Int);

(0::Int) ->
    (1::Int)}}};

main :: Types.Int = fac (Types.Izh (4::Int));
```

That looks nicer. Here, I have shortened the type names in the primitive value type notations, removed the function return type notations and replaced the longwinded arithmetic operator names with their traditional symbols. We can see that the guards (Haskell pattern matching syntax, denoted with '|' symbols) have been removed and replaced with a case statement, in keeping with Core's minimal mindset. We can also see a lambda calculus esque structure, with our computations being represented in the form of $\backslash[params] \rightarrow functionbody$. The influence System F can be seen in the type signatures prefixing the fac function. Of note also is the '@' symbol, used to denote function application. More on this later, for the moment it's enough to know that x + y can be represented as + @ x y, that is *plus applied to x and y*.

SAPL is similar to core, with a few differences. SAPL uses a *select* expression in place of Core's case expressions although this is a difference in name only, the semantics of both being equivalent. SAPL also has a built in *if* expression which in Core would be expressed using case expressions instead. SAPL contains no type annotations, all such information having been removed after the Clean compilation type-checking stage.

Choosing a language such as Core or SAPL as our input language would provide a number of benefits for this project. Firstly, it would allow us to write programs in Haskell or Clean and make use of existing tools to translate these into their Core-like representations.. This would mean that we can write our browser programs in languages with pleasant high level syntax. We could leave the parsing of these programs to existing Haskell or Clean compilers and focus instead on the more interesting problems of compilation and runtime. We would also get type-checking for free, as the type checking for both Core and SAPL would have already been performed by the time they are emitted by their respective compilers. It would also reduce the learning curve for users who wish to use this project who already have experience with writing programs in languages with syntax similar to Haskell or Clean.

Having seen the various features of System F languages such as SAPL and Core, I was quite content to choose such a language as the input for my own project. From our research into the iTasks system, we have seen that such a language can provide a good starting point for a project such as this. Between SAPL and Core I was more inclined towards using the latter. I have far more experience using Haskell than Clean. Also, though we do not need them currently, it is worth noting that having Core's type annotations may prove useful for future improvements to our project. Such information would be of great help when diagnosing runtime errors, for example. It was decided that Core would be the input language.

7.3 Compilation Strategy

Now that we have decided on our input language, we need to plan our compilation strategy. Thankfully for us, there is a wealth of literature and previous work on writing a compiler for System F resemblant languages and a number of approaches already exist from which we can take inspiration. An overview of general functional language compilation is provided in the background chapter of this report so that this section and following sections discussing compilation can focus on the aspects of immediate interest to us.

ADD GRAPH REDUCTION TO BACKGROUND

First, we need some way to compile programs written in our input language into some form from which we can derive an initial state. The traditional way to do this is to view our program as a graph as described above and convert this graph into a programmatic representation. There are many existing functional compilation language strategies which achieve this, so it was decided to examine those before deciding on an implementation. In particular, there were two strategies which I investigated in-depth

1. The template instantation machine
2. The G-Machine

7.3.1 The template instantiation machine

The template instantiation machine is a simplistic approach to functional graph reduction and compilation but conveys many of the ideas used in more complicated approaches. When the project started moving towards implemenation of a solution, this is the approach I started with. It proved very useful in understanding some of the key ideas in writing a functional compiler and gave me an idea of the scope of such an implementation.

The machine is built upon a state consisting of

- A stack
- A dump
- A heap
- A globals array

The stack is a stack of addresses which correspond to nodes in the heap. These nodes form the spine of the expression being evaluated. The dump is used to record the state of the expression being evaluated. When we **EXPLAIN**. The heap is a list of nodes and their associated heap addresses. The nodes contained in the heap correspond to the examples given in the overview in the background section. The globals array associates the names of our declared supercombinators with their addresses in the heap, allowing supercombinators to be looked up by name.

The operation of the template instiation machine is described by a number of state transitions which are called if the state of the machine represents that of the transition. When no transition rule matches, we assume execution to be complete. At most one state transition rule will apply at any point; more than one would imply non-determinism which is not permissible. In general, the state transition rules are concerned with the type of node pointed to by the top of the stack and will react accordingly. For example, in the case where an application node is found, the appropriate rule is triggered which will pop the application and unwind its two arguments onto the stack. The rule for dealing with supercombinators is also of interest to us. When a supercombinator node pointer is found on top of the stack we instantiate it by binding the argument names of its body to the argument addresses found on top of the stack. We then discard the arguments from the stack, reduce the resulting redex and either push the root of the result to the stack or overwrite the root of the redex with the result, depending on whether the implementation in question is performing lazy updates.

The implemenation of the template instantiation machine is split into compilation and evaluation stages. The compiler in the first stage takes our input program, along with any built-in prelude definitions, and builds an initial state of the form described above. The evaluator in the second stage takes this initial state and runs our machine on it one step at a time, applying the necessary state transition rules, until we reach a final state. The stepping function takes as input a state and returns a resultant state. The machine is considered to have finished when the stack consists of a single pointer to an item which can be evaluated no further, for example and integer or a fully saturated data constructor.

PROVIDE EXAMPLE COMPILATION

This is about the extent to which I implemented my initial template instantiation machine. There were further improvements which could be made, such as adding primitive operations, let expressions etc. etc. My implementation was capable of taking a program representing an identity function, compiling it and evaluating it. This was sufficient for my purposes of understanding some of the basic concepts of functional compiler implementation. At this point, I was looking to move towards a different implementation which would better suit some of my requirements. Those interested will find a more complete implementation of a template instantiation compiler described in greather depth in **SJP BOOK AND TUTORIAL**.

The template instantiation machine suffers from one notable problem which made it particularly unsuitable for our purposes. The general operation of the template instantiation machine constructs an instance of a supercombinator body. Each time we attempt to instantiate a supercombinator we recurseivly traverse the template. This action is executed in the evaluation stage. We know that our end goal is the ability to execute programs in a browser and as such it would be of great benefit to us if we could minimize the amount of work needed at run time. Thankfully for us, there exist implemenations which are capable of doing this!

7.3.2 The G-machine

The G-machine differs from the template instantiation machine in a few ways but there is one significant difference in their principles of operation. The G-machine translates each supercombinator body to a sequence of instructions which will construct an instance of the supercombinator body when run. In comparison to the template instantiation machine, this allows us to execute the actions of a supercombinator without the need to instantiate it at run time, this having been achieved in advance at compile time.

The G-machine decouples the compiler from evaluator to a greater extent than the template instantion machine. The G-machine's compiler not only produces an initial state from our input program but also a list of instructions which when combined with the initial state can be evaluated to a final state. The set of instructions drawn from is designed to be minimal. The instructions and state produced by the compiler can be said to represent an *abstract machine* which can then be implemented on various different architectures and platforms. This makes it easier to write runtime evaluators for G-machine compiled programs, as the initial state can be expressed in an intermediate form between that of a reducible graph state representation and an easily executed program state. This fact will prove useful when it comes to writing the runtime to evaluate our compiled programs.

The general form of the G-machine is similar to that of the template instantiation machine. We take our input language and compile this into an initial state. This state is identical to the one listed previously, with the addition of a sequence of instructions to be called in order to evaluate our state. There still exists a heap and stack, the former still containing nodes of the same types as those previously encountered. This state is passed to the runtime which will execute until we reach a final state where we have no further instructions to evaluate and a single pointer to a node in the heap on top of the stack. Our runtime must now possess the means to interpret these instructions and execute them upon the current state, producing a new state. This is in contrast to the evaluation stage of the template instantiation machine which had a list of state transition rules which would fire when the state matched that of one of the rules. Here, only the initial state was necessary for evaluation.

The following are instructions one would find in a basic G-machine implementation.

- Unwind: unwinds the spine in much the usual manner.
- PushGlobal: finds a supercombinator by name from the globals array and places its heap address on top of the stack.
- PushInt: Places an integer node in the heap and its address on top of the stack.
- Mkap: Forms an application node from the top two node pointers on the stack.
- Slide: drops N pointers from the stack behind the current top pointer. Used to remove arguments pointers after evaluating a supercombinator.
- Update: updates the root of a redex with its reduced value. Used to implement laziness.

More in-depth information on these and other instructions can be found in later implementation chapters.

The G-machine compiler consists of a number of schemes which determine what sequence of instructions to output when we encounter certain expressions in our input language. These together with an initial heap built from the supercombinators of our input program along with any prelude definitions are then passed to the runtime.

The improved efficiency of the G-machine over the template instantion machine, along with the intended ease of writing an abstract machine to evaluate programs at runtime, makes it the better choice of the two explored implementations.

7.4 Runtime

Now that we have decided on the G-machine as the general form of our compilation strategy, we need to consider how we will evaluate G-machine compiled programs in a manner that is useful in web programming. As anticipated from early on in this project, JavaScript is the only realistic way to achieve this. JavaScript is far and away the most commonly used browser-side programming language. If we want the results of this project to be usable in any general sense, JavaScript is the only sensible option.

It is worth re-examining Fay at this point. We are taking some inspiration from its implementation in that we are using a runtime written in javascript to evaluator a low level representation of our compiled program. However, it's worth noting the differences. As we have decided to use the G-machine as our overall compilation strategy, we will not need to concern ourselves with the concept of thunks. Our smallest indivisible unit of computation will instead be singular graph nodes. In a sense, these are comparable to the thunks represented in Fay placed into the context of a graph reduction machine. Our primitive operation instructions will be implemented as operations on graph states returning new graph states, as opposed to thunks. We do not concern ourselves with forcing thunks, so to speak, but we will have to deal with evaluating computations to weak head normal form, albeit in a graph reduction context.

Our runtime will receive the graph state representation of our input program after compilation, along with the instruction sequence associated with this state. It will have to iterate through the instruction sequence, enacting the actions specified by the instructions upon our state. We can see that our runtime will need to understand how to interpret these instructions into concrete javascript actions which we can then call from our browser. Our runtime must also be capable of representing our compiled initial state and instructions in a javascript format for our actions to be enacted upon. We will need some abstract representations of the components of our state, our instructions and their attributes. The compiled state will need to be serialized from the language which we use to represent our program through the compilation stage (which as it turns out will be Haskell) into Javascript. This will be achieved by instantiating our abstract state javascript representations to represent the concrete components of our state and instructions.

At this point, we will have a compiled G-machine state in Javascript form and the means to evaluate it into subsequent states. A javascript function will iterate through our states appling the changes necessary. On each iteration, this will check the state of the stack and the instruction at the front of the instruction list, apply the evaluation actions required by the instruction upon the state and return a new state. This will need to be achieved for each state and instruction in our evaluation until we reach a final state (which we will also need a means of testing for). A call to this function will be included in our serialized compiled state which we can call to initiate execution. When we have finished executing, we will then need to extract the result of our evaluated program from our state and return it in a javascript format for use in our browser, other javascript functions etc.

7.5 Summary

We now have a

8 Implementation