University of Dublin

TRINITY COLLEGE

Browser-side Functional Programming

Kieran Manning

Ba(mod) Computer Science
Final Year Project, April 2012
Supervisor: Glenn Strong

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

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.

Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

Kieran Manning

Date

Permission to lend

Goal

The goal of this project is to provide alternative solutions to a number of problems identified in browser-side web programming. We will look to existing concepts in functional programming for inspiration when designing these solutions. 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 produce 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. As we are unable to change the semantics of JavaScript, we shall instead use it as a target language. Our project will provide a means of executing a minimal strongly-typed functional language in a web browser. Our implemenation will consist of a compiler to compile this functional language into an executable state and a runtime capable of evaluating this state to a result. The runtime will executable from within a browser. Programs written in this language will be type-safe and our runtime will be capable of evaluating them lazily. Our runtime will be written in JavaScript so as to be compatible with all major browsers and allow for the lazy evaluation of programs written in our input language.

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.

Programs written in JavaScript are strictly evaluated by default. It is possible to write lazily evaluated programs in JavaScript however doing so requires us write the underlying lazy representations and methods ourselves.

By default, JavaScript provides no means to write natively lazy 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 [3]. Languages such as Haskell, have shown that they can be solved albeit in a different programming domain. Other languages such as CoffeeScript [2] and Fay [4] 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.

Acknowledgements

First and foremost I'd like to thank Glenn Strong for agreeing to supervise this project and providing the inspiration. This project would not have been possible without his support and optimism, not to mention his lectures which first introduced me to these topic.

My second reader Andrew Butterfield for the time taken to review this project.

My family for their support through my degree and acceptance of my sometimes hermitic work schedule.

Dave and Gareth for emotional support and criticism.

Contents

1 Introduction

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 the moment of this writing the language is 18 years old. 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.

2 Background

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 to which I will be commonly referring to throughout this report.

2.1 Functional Programming

Functional programming is a programming paradigm distinct from imperative programming. It aims to view programs more as mathematical functions or series thereof and less as sequences of high level machine instructions. The concepts of global state and stateful computations are avoided. Where necessary, states are represented as values passed between stateless functions. Such stateless programs give rise to the notion of *purity*. This is the concept in many functional languages that states that programs will always evaluate to the same result unless stateful operations are intentionally added. Pure functions are often referrred to as *referentially transparent*. Referential transparency of an expression in a program allows us to say that replacing the expression with its value will cause no change in the execution of the program as a whole. Lazy evaluation is an evaluation strategy frequently found in functional programming. Distinct from strict evaluation, Lazy evaluation operates on a *call-by-need* basis, only evaluating aspects of an expression as they are required. A secondary characteristic in lazy evaluation is that of *updating*. Updating puts referential transparity to practice by rewriting expressions with their evaluated results after their first evaluation. Many functional langauges can be seen as abstractions built on top of (polymorphically typed) lambda calculus [7, pp.37], examples including Haskell, Miranda, ML and Lisp. This is particularly noticeable in Lisp where the influence of lambda calculus is still visible in its syntax. This is an extremely high-level overview of functional programm. I will attempt to explain any of the above mentioned concepts which appear frequently throughout this report.

2.2 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 runtime 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 [8, pp.12]. 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 [7, pp.185], 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.

2.3 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 [7, pp.33]. 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. Let's take two examples:

```
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 gives us the character 'a'.

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

Here, `[1..]` is the list of integers beginning with 1 and continuing to infinity. In both examples, the process for evaluating `take` is the same. The square bracket syntax is used to construct lists containing the items within the brackets as usual. The `..` syntax identifies a pattern between the items mentioned within the bracket and will construct a list as defined by that pattern. However, the list is not constructed in advance. In the context of functional programming languages such as Haskell, we view lists as concatenation functions on elements. When we call `take` on a list, we evaluate this function as many times as necessary to produce the number of elements required. In this manner, we do not need to know what items lay beyond the first element in the list when applying the `take 1` function as we do not attempt to evaluate them.

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 [7, pp.208]. 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 is only possible in cases where we can say that the expression in question is *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.

2.4 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 [7, pp.198]. 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

3 Problem Space

3.1 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.

3.2 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.

3.3 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.

4 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.

4.1 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.

4.1.1 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 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 [6]. 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 say 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 embedded 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 tend to be tedious and vague.

4.1.2 Runtime

Fay programs are compiled into a low level JavaScript representation. The compiled program is built from primitive thunk objects and operations which process these thunks. The abstract representation for these thunks exists in a seperate runtime written in JavaScript. This runtime also contains various operations to handle thunks in compiled programs along with thunk-level functions in JavaScript which represent Fay's primitive operations. When a Fay program is compiled the output is bundled with the runtime which is capable of evaluating the compiled program.

The thunk objects used to represent Fay programs are not dissimilar from the nodes found in the graphs described in the background information section. Singular expressions are built of thunks and complex expressions are built from series of thunk, much like our reducible graphs. Thunks are by default unevalated, or unreduced, and must be *forced* in order to reduce to weak head normal form. This again should sound familiar after examining the idea of reducible graph expressions. A thunk is represented in the abstract by a function as follows:

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

Instances of thunks are represented as instantiated function objects. We also need functions to handle primitive thunk operations. Two important examples are:

```
// Force a thunk (if it is a thunk) until WHNF.
function _(thunkish,nocache){
  while (thunkish instanceof $) {
    thunkish = thunkish.force(nocache);
  }
  return thunkish;
}

// Apply a function to arguments (see method2 in Fay.hs).
function __(){
  var f = arguments[0];
  for (var i = 1, len = arguments.length; i < len; i++) {
    f = (f instanceof $? _(f) : f)(arguments[i]);
  }
  return f;
}
```

The first forces a thunk to weak head normal form (using a JavaScript prototype "force" method tied to the thunk object). This is used to evaluate expressions represented in thunks to weak head normal form, much in the manner explained in the introduction section on graph reduction. The second example here applies a function to two arguments. We can see this function as being similar to the appliation nodes we'd find in reducible graph expressions.

Primitive operations in the runtime are also represented as JavaScript functions. For example, the two functions needed for primitive multiplication look like this:

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

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

The first represents a fully saturated multiplication operation ie. one in which we have both necessary arguments. We *return a function which when called* will force both operands into weak head normal form and multiply them together. As such, the return of the multiplication operation is not itself in weak normal form and will remain unevaluated until it is forced. Such behaviour allows for laziness in the language.

The second function represents an undersaturated multiplication operation. In this case, we have only one of the two necessary arguments. We take our provided operand and place it into wrapper function which takes a second operand. This function contains the means to return an unevaluated saturated multiplication operation. We then return the wrapper function, which when applied to the as of yet unprovided second operand, will return an unevaluated multiplication expression, just as in the full saturated example. Such behaviour is allows for curried operations in Fay.

There are many other functions in the runtime although these examples are sufficient to get a feel for how it works. A large part of the runtime is just implementations of primitives such as arithmetic and list operations etc. There are also two other somewhat significant functions which handle the serialization and unserialization of objects from Fay to JS. However these are mostly utility functions and are not particularly interesting to us, once we have seen the principles of the implemenation.

4.1.3 Analysis

Fay's take on the problem of improving browser side functional programming presents some interesting ideas. In terms of accomplishments, it manages to bring the ideas of laziness, static typing and purity to the problem space while also remaining compatible with all current browsers. Its implementation also provides some interesting concepts. Of note in particular is its runtime, which provides a means to evaluate a compiled program in a browser. This is an idea which will be important when it comes time to write our own implementation. Its use of a Haskell embedded domain specific language, and the benefits this brings, is also interesting to us.

4.2 iTasks

iTasks is a project built with the Clean programming language. Its aim is different from ours but its implementation manages to achieve some of our goals. iTasks is a workflow management system specification language, embedded in Clean, which generates workflow applications. As it is written as a DSL inside a functional language, much like Fay, it benefits from a number of the advantages of the host language. In particular, iTasks inherits Clean's concepts of generic programming and its type system.

4.2.1 Implementation

The domain specific language of iTasks makes available to the programmer a number of constructs and functions needed to express a series of workflow tasks when specifying a workflow system. Tasks are represented by a parameterized `Task a` type and functions can be constructed in the domain specific language which take these tasks and perform operations on them. A set of basic functions provides functionality such as task composition, splitting etc. The following is an example of a basic iTasks specification

```
:: Person = { firstName   :: String,
              SurName     :: String,
              dateOfBirth :: HtmlDate,
              gender      :: Gender }
:: Gender = Male | Female
```

```
enterPerson :: Task Person
enterPerson = enterInformation "Enter Information"
```

From this, iTasks will produce a standard form with fields for firstName, surName, dateOfBirth and gender. There are a number of interesting points in this example. Clean's concept of records is used to define a "Person", consisting of a number of fields of types String, HtmlDate and Gender. Gender is also a user defined type. The function enterPerson takes no parameters and returns a value of type Task Person, which is a Task carrying data of type Person. This is making use of Clean's concept of parameterized data types to carry aditional information about the task in particular. As well as iTasks specific functions, the programmer can write general Clean code to assist in their work with iTasks.

4.3 Core/SAPL paper

4.4 Others

We have discussed some existing projects which provide solutions to some of our problems at length. It is worth mentioning a few other projects in passing which are related to our work. As previously stated, this is far from the first project to identify problems with the state of browser-side programming and reliance on JavaScript. Many projects have been created to address these problems and many take similar approaches to the ones discussed above; using JavaScript as a target language, adding to its semantics and/or avoiding certain unwanted aspects.

Coffeescript is an example of such a project which has gained a lot of interest in the last few years. First released in late 2009, CoffeeScript provides a sort of wrapper language which compiles into equivalent JavaScript. The underlying semantics of JavaScript are unchanged. The aim of the project was to bring a more pythonic syntax to the language. It achieves this, making it possible to write what would normally be verbose code in JavaScript as smaller, equivalent code in CoffeeScript. Declaring a function for example, which would look like this in JavaScript:

```
race = function() {
  var runners, winner;

  winner = arguments[0],
          runners = 2 <= arguments.length ?
          __slice.call(arguments, 1) : [];
  return print(winner, runners);
};
```

Could be expressed as this in CoffeeScript:

```
race = (winner, runners...) ->
  print winner, runners
```

Coffeescript is not of great interest to us however as it has no effect on the underlying semantics of JavaScript much less bringing functional concepts to the language.

Another language worth mentioning is ClojureScript. It provides a compiler which compiles Clojure into JavaScript. This allows the use of Clojure concepts such as hashmaps, let bindings etc. in browser executed programs by representing them in compiled JavaScript. Clojure's syntax is also preserved, still showing the nested parenthesized expression syntax inherited from and associated with Lisp. However, my hope was to achieve similar goals with Haskell and as literature on ClojureScript is somewhat sparse, it was not of great use.

5 A High Level Design

5.1 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.

5.2 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 be 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 reported. 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 or derivites of System F [10] 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 'Core' and similar languages as 'Core-like'. We will use 'External Core' to refer to the specific output of GHC

when passed the -fext-core flag, as distinct frmo 'Core' in general which can refer to this output or the internal representation. Core is a lambda calculus dialect (System F with added type coercions to be specific [11]) 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 exprs look something like this [5] [9, pp.9]..

```
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 many things). 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 [9]. 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 (ds2X16::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 [1, pp.5] 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.

5.3 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

5.3.1 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 come across an expression which we need to be in weak head normal form, we save sufficient of the state of the machine to allow us to return to this point of execution. We then evaluate the expression in question in WHNF, leaving a pointer to it on top of the stack and return to the point of execution saved in the dump. 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 "The Implementation of Functional Programming Languages: A Tutorial", by Simon Peyton Jones and David Lester.

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!

5.3.2 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 [7, pp.294]. 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 similar to the one listed previously, with the addition of a sequence of instructions describing the evaluation of the current expression. There still exists a heap and stack, the former still containing nodes of the same types as those previously encountered, with the expection of our NGlobal nodes. These now contain the instructions needed to instantiate a supercombinator definition rather than the body of the supercombinator itself. 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 possess the means to interpret the provided 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 compiled supercombinators of our input program along with any prelude definitions and primitive operations, make up the initial state we pass to our 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, make it the better choice of the two explored implementations.

5.4 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 operations on 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.

5.5 Summary

We have decided that our project will consist of a Gmachine based compiler that accepts GHC's external Core. This will be compiled into a graph based representation of the initial state of our program. This initial state will be serialized to a JavaScript representation thereof. A Gmachine graph evaluator implemented in JavaScript will then evaluate this initial state to a final state and resultant value

6 Implentation

Now that we have a plan for the design of our project, it is time to begin implementing it. There are still many considerations to be made. Our strategy still leaves many of the finer details undecided. We'll now examine some of these details in more depth. The following section will explain the design choices made when implementing my project. It will also explain the implementation itself, providing select examples of the code used as well as the progress of a program as it passes through our compiler. For brevity's sake I will be trying to avoid adding unnecessary code snippets throughout this chapter. The code referenced can be found in the code repo attached to this project. I will provide file names where necessary.

6.1 Core Language Representation

We'll start with a description of the Haskell representation for our input language, GHC's external Core. This is implemented in haskell using algebraic data types (or ADTs) and our own defined types.

```
data 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)
deriving(Show, Read)

type ScDefn a = (Name, [a], Expr a)
type CoreScDefn = ScDefn Name

type Program a = [ScDefn a]
```

This is the ADT representing a Core expression. We can see many of the ideas we discussed earlier when discussing Core and its components. The 'a's represent the type of the expression. 'Names' are just renamed strings. The EAp expression is the Core equivalent of the function applicators we already discussed. ECase exprs are case statements. ELam and ELet are lambda abstractions and let expressions respectively, although in practice neither were implemented in our compiler. These were not deemed critical to the evaluation of our language as their behaviour can be expressed in terms of other expressions. The "deriving(Show, Read)" at the bottom of our data type definition will crop up frequently. This tells Haskell to automatically derive instances of Show and Read for our data type, allowing it to represented as and interpreted from String types respectively. Supercombinators are defined as tuples of their identifier, the expression forming the body of the supercombinator and the list of parameters to pass to the expression. A program is built from a series of these super combinator defintions, named ScDefn in the code.

We will provide an example of a simple program written in this representation. We'll use this same program as an example throughout this section, allowing us to see the exact steps it takes as it moves through the stages of our compiler. The example we will use is the representation of a program that defines a supercombinator named 'K' that takes two arguments and returns the former applies this supercombinator to two integers. When written in terms of our ADT, this would look like:

```
[
("K", ["x", "y"], EVar "x"),
  ("main", [], EAp (EAp (EVar "K")(ENum 1))(Enum 2)))
]

K x y  = x
main = K 1 2
```

A more readable version is also provided above. Note that our program consists of two supercombinator definitions. The first of these is our K function. The second is a function called "main" which applies our function K to integers 1 and 2. Our language requires an entry point for our runtime to begin evaluation from. In our implemention, as in many, the standard will be to name that entry point main. It is otherwise a regular supercombinator defintion. This project is capable of taking more complicated programs however these quickly become unwieldy to discuss. Those interested will find some subjectively cooler examples in GPrelude.hs in the supplied code repository.

6.2 Compiler

Our compiler will accept such representions of the core language as input and compile them to an initial state. We have decided to use the Gmachine as the template for our compiler implementation. This will make the problem of writing the compiler substantially easier as implemenations of the Gmachine already exist. Rather than re-implement the wheel we will re-use an existing implementation, specifically the one described in *Implementing Functional Languages: A Tutorial* by Simon Peyton Jones and David Lester. This will allow us to focus on the more interesting aspects of the project, most specifically the runtime. Those familiar with the implementation in the above mentioned text can easily skip this subsection. If not, I'd recommend reading the text as Jones and Lester describe it better than I could ever hope to.

6.2.1 Program State

We have mentioned our *initial state* countless times so far in this report. Lets see what that state looks like when represented in Haskell.

```
type GmState
= ( GmOutput,
GmCode,
GmStack,
GmDump,
GmHeap,
GmGlobals,
GmStats)

type GmCode = [Instruction]

type GmOutput = [Char]

type GmDump = [GmDumpItem]

type GmDumpItem = (GmCode, GmStack)

type GmStack = [Addr]

type GmHeap = Heap Node
type Heap a = (Int, [Int], [(Int, a)])

type GmGlobals = [(Name, Addr)]
```

The state of our program at any point throughout its evaluation is reprseneted by GmState. The definition of GmState and its constituent components is described above. We can ignore GmOutput and GmStats, they're not of particular interest to us.

GmHeap is defined as Heap Node, where Heap is a parameterized type definition and Node is the type of items contained in the heap in question. The Heap type itself is a three item tuple. The first item, an integer, represents the number of objects in the heap. The second item, a list of integers, represents unused addresses in the heap. The third item is the list of Nodes contained in the heap. It is defined as a list of tuples of integers and 'a's. In our heap, the 'a's are nodes, as defined in GmHeap. This list of tuples is the representation of the reducible graph described in the background section. Nodes are defined as follows:

```
data Node
= NNum Int
| NAp Addr Addr
| NGlobal Int GmCode  -- NGlobal Arity Code
| NInd Addr  -- Indirection
| NConstr Int [Addr]
deriving(Eq, Show)
```

These correspond with the nodes described in the background section with a couple of additions. NInd nodes are indirection nodes containing the address of another node in the stack. These are used to implement the lazy updates discussed previously. When we evaluate an expression, we create an indirection node to its value and replace the address of this node with that of the root of the expression. NConstr nodes are used to represent data constructors. They consist of an Int representing the arity of the data constructor and a list of addresses corresponding to the locations in the heap of the elements of the data constructor. NNum, NAp and NGlobal nodes correspond to integer primitives, applicators and function identifier nodes respectively. NGlobals consist of an Int representing the arity of the supercombinator they represent and a list of instructions describing the instantiation of the supercombinator.

GmStack represents our stack. It consists of a list of addrs which are user defined type synonyms for integers. These correspond to locations in the heap. GmStack is used in the same manner as the stack described in the background section. When we compile our input program, the stack in the initial state will be empty. In general, addrs of heap items needed in immediately pending expression evaluations are pushed to the stack and removed when no longer needed.

GmGlobals is a list of tuples of names and addresses. The name corresponds to a supercombinator definition identifier and the address associated with it corresponds to the location in the heap of the NGlobal node representing the supercombinator. When we wish to find a supercombinator by name, we look up its name in the GmGlobals array to find the address associated with it.

GmCode represents the list of instructions immediately pending execution. An instruction describes the actions to be next executed upon the program state. When we encounter a super- combinator in our input language, we compile it into a list of instructions representing its behaviour. We then place an NGlobal node containg these instructions in our heap. When we wish to apply our supercombinator, we lookup its node by name firstly in GmGlobals, then by address in the heap. We then place its instructions into GmCode and execute them. Instructions are defined as follows, we'll provide an explanation of their operations when discussing the implementation of the runtime:

```
data Instruction
= Slide Int
| Unwind
| PushGlobal Name
| PushInt Int
| Push Int
| Mkap
| Pop Int
| Alloc Int
| Update Int
| Eval
| Add | Sub | Mul | Div | Neg
| Eq  | Neq | Lt  | Le | Gt | Ge
| Cond GmCode GmCode
| Pack Int Int
| Casejump [(Int, GmCode)]
| Split Int
deriving(Eq, Show)
```

GmDump represents the dump described in the background section. It consists of a list of GmDumpItems which contain a tuple of GmCode and GmStack. When we encounter an expression that requires a sub expression to be in weak head normal form, we defer the current state of GmCode and GmStack to the dump. We then evaluate the sub expression until it is in WHNF and restore program flow, where upon we will be able to use the value of the sub expression in a strictly evaluated context. This is comparable to the steps taken to save program state when calling a subroutine in assembly languages.

6.2.2 Compilation Procedure

Compilation is started with the *compile* function in GCompiler.hs. This function takes a Core program and returns an intial GmState. The initial values for GmStack.. and GmDump are empty lists. The initial value for GmCode is [PushGlobal "main", Eval], which will tell the runtime to start by executing the main supercombinator. We construct an initial heap using the buildInitialHeap function. This calls compileSc on the supercombinator definitions present in the input program and in the compiler prelude and combines their compiled supercombinators with existing pre-compiled primitive operations.

Two compilation schemes exist within the compiler for handling expressions. The first, represented by the function compileE, compiles expressions in a strict context. There are cases where we know that an expression will need to be evaluated strictly. An example would be primitive addition. Strictly compiled expressions generally produce smaller graphs and can be evaluated quicker. Hence when given the choice, we would like to evaluate strictly where possible. Expressions which can be evaluated in this manner are found through pattern matching in compileE. Those that can't fall through to compileC, a function representing our lazily compilation scheme.

Each super combinator in the program is thus compiled into a list of instructions by which to instantiate it when it is called at runtime. These instructions are attached to NGlobal node data types and added to the heap. Our primitive operations will also exist in the heap in the same manner. The sequence of code generated from compiling our program is now placed in the initial GmCode. GmGlobals is generated while building the initial heap and will associate the names of our compiled supercombinators with their addresses in the heap. We Now have a compiled initial state representing our program.

Let's look at our 'K' program after compilation.

```
GmCode: [PushGlobal "main", Eval]
GmStack: []
GmDump: []
GmHeap: (2,
[3..],
[ (2,NGlobal 0 [PushInt 2,PushInt 1,PushGlobal "K",
 Mkap,Mkap,Update 0,Pop 0,Unwind]),
(1,NGlobal 2 [Push 0,Eval,Update 2,Pop 2,Unwind])
])
GmGlobals: [("K",1),("main",2))]
```

We can see that the first code to be evaluated at runtime will be `PushGlobal "main", Eval`. This will tell our runtime to instaniate the main supercombinator and provides the entry point previously mentioned. The stack is empty as expected, as is the dump. GmHeap contains firstly an integer 2. This is the number of objects in the heap; Secondly, a list of empty addresses (slightly edited for our purposes, the heap is represented by an infinite lazy list in Haskell which we cannot show in its original form for obvious reasons). Lastly, we have the representation of our compiled graph containing two NGlobal nodes representing the supercombinators 'K' and 'main'. The compilation environment has been greatly simplified for ease of demonstration. Normally all primitive operations defined in the compiler as well as any prelude functions would also be contained in the heap, in a similar fashion to the two supercombinators above. Lastly, GmGlobals contains the names and heap addresses of our two supercombinators.

Our initial state is now ready to be evaluated. However, as it is currently represented in haskell, we need a way to make it compatible with JavaScript.

6.3 Serialization To JavaScript

Our serialization code lives in Haskell2JS.hs. It isn't particularly exciting but is pretty crucial to the operation of our compiler. It consists of a number of functions which translate the components of our state into an equivalent JavaScript representation. Before showing the operation of this code, we'll look at the JavaScript implemenation representing the components of our state.

1. Lists
   We use lists in Haskell to represent globals, the heap etc. Haskell equivalent lists do not exist in JavaScript, but there is an array type. When we wish to represent a list such as GmGlobals, we create a JavaScript array object. GmStack becomes `var GmStack = [];` GmDump is represented as a list of two-item lists of GmCode and GmStack.

2. Associative Lists
   These are used to represent GmHeap and GmGlobals. They consist of a list of tuples. In the case of GmHeap for example, we have a list of tuples of supercombinator names and heap addresses ([Name, Addr]). Primitive JavaScript arrays wont help us here, so we use JavaScript objects. The GmGlobals from our compiled state now becomes

   ```
   var GmGlobals = {
   "K" : 1,
   "main" : 2
   }
   ```

3. Data Types
   We use Data Types to represent many aspects of our state in Haskell, such as nodes, instructions etc. As there is no such concept in JavaScript, we again need to find a new way to represent these values. In JavaScript, it is possible to instantiate functions as function objects. Function objects are similar to regular objects in a number of ways. They can be defined in the abstract and instantiated. They can take parameters and maintain values as attributes. For example, if we wanted to express an NNum node as one of these function objects, we could do so like this:

   ```
   function NNum(n){
   this.n = n;
   }
   ```

   We could instantiate this as:

   ```
   var node = new NNum(n);
   ```

where n is the value we wish our node to contain, must like the Int parameter passed to the NNum data constructor in our Haskell equivalent.

We will use this concept of instantiated function objects to represent nodes and instructions. In the case of data constructors of arity 0, our function will take no arguments and appear empty. They are still sufficient in this state to represent the equivalent data constructors.

4. Functions
   Functions in Haskell will be represented as same in JavaScript.

Our serialization code will take an initial state from our compiler and produce an equivelent representation in Javascript. This representation will consist of the components described above. The function GmState2JS in Haskell2JS.hs takes our state and calls the relevant functions on its constituent parts. Globals, stack and dump are trivial as we know those will be empty. The heap will require slightly more thought. GmHeap2JS iterates through each item in the heap, converting nodes to JavaScript representations. Each node is represented by an instance of the appropriate function, taking as parameters the parameters of the node. Converting node parameters is trivial in all cases except NGlobal nodes. These will contain instructions required to instantiate a supercombinator. We call gmInstruction2JS on these instructions, converting them also to instantiated function object representations, which we then include as parameters to the NGlobal node. GmCode is serialized in a similar manner although in practice that's trivial as we always know what its contents will be.

The output of this serialization step is a string parseable by JavaScript representing our compiled state. For our example 'K' program, that output looks like this:

```
var GmOutput = [];

var GmCode = [new PushGlobal("main"),new Eval()];

var GmStack = [];

var GmDump = [];

var GmHeap = {
objCount:2,
freeAddrs:[3,4,5,6,7,8,9,10],
addrObjMap:{
2:new NGlobal(0,[new PushInt(2),new PushInt(1),new PushGlobal("K"),new Mkap(),new
1:new NGlobal(2,[new Push(0),new Eval(),new Update(2),new Pop(2),new Unwind()])}
};

var GmGlobals = {"K":1,"main":2};

var GmState = [GmOutput, GmCode, GmStack, GmDump, GmHeap, GmGlobals];

function main(){
return evalProg(GmState);
}
```

I'll mention at this point that the free addresses in GmHeap are represented rather naively as a finite list of integers. This decision was originally intended to be temporary. I'll provide a solution outline in conclusions. It is relatively trivial to write correctly. For now it is sufficient to know that this method is far from ideal but works.

We now have a JavaScript representation of our compiled program state. The next step is to evaluate this state to a final result. This will be accomplish in our runtime, which we will now describe.

6.4 Runtime

We will now examine the implementation of the runtime. As described in the design choices section, this will be a JavaScript representation of an abstract machine capable of evaluating a Gmachine compiled state. We have already described the components of this state and their abstract representations in JavaScript so this section will be concerned mostly with the methods of evaluation we employ when executing our state. The code related to this section can be found in runtime.js in the supplied repo.

Our runtime consists of a number of stateless functions and the global definitions for our compiled state. Functions operating on our compiled state are generally of the following form:

```
function funcName(xState, otherParams){
var State = xState;
var component = getComponent(State);
alter_component(component);
var newState = putComponent(State);
}
```

We wish for our runtime operations to be somewhat pure out of concern for code cleanliness and sanity. However, JavaScript functions operate on a pass-by-reference basis. For this reason, we declare a new variable represented a copy of our state inside the scope of each such function. We apply our changes to scoped copy and return it, leaving the original copy passed to the function unchanged.

In the serialized example of our "K" function above, you may have noticed the function main()return evalProg(GmState). This serves as our browser's entry point to the evaluation of our state. The call to evalProg() tells our runtime to evaluate the GmState definition provided. The function evalProg iterates through each state in the evaluation of our compiled program, evaluating each in turn until we reach a final state. This final state is determined by the function gmFinal(), which checks if we have run out of instructions to execute in GmCode. This implies that there is no further evaluation to be performed on the current state and the node pointed to by the address on top of the stack should represent the final result of our program.

On each iteration of evalProg()'s evaluation loop, we call the function step() on the then current state. Step acts as an instruction dispatch function. It will pop one instruction from the sequence in GmCode, check it's value and call the relevant function representing its behaviour. The instanceof method in JavaScript is used to check if an instantiated function objects is an instance of a certain function. This is how we compare our node and instruction representations and perform pattern matching on them. The parameters of each instruction are passed as parameters to the functions representing their behaviour.

We will now examine some of the utility functions in our runtime. Between lines 66 and 91 of runtime.js, we have functions id(), head(), tail() and an array prototype method addition drop(). The first three functions work as expected. Functions head() and tail() are included to make reasoning on lists more intuitive. JavaScript has built in push and pop methods, however these operate in a destructive pass-by-reference sense which is unhelpful when trying to model execution avoiding the notion of global state. Drop() was included for similar reasons, implementing similar functionality to Haskell's drop.

hAlloc at line 233 is used to allocate a node on the heap. It takes the GmHeap from our state and a node and returns a new heap with the node at the last free address of the heap. The free addresses and object count of the returned heap reflect this change. hLookup and aLookup immediately following are used to lookup items in the heap and globals respectively. Lines 255 to 320 contain a number of similar looking functions. These are used to standardize access to components of our heap, making it easier to access and update our state in a clean and organized fashion.

6.4.1 Runtime Instructions

The implementations of our Gmachine instructions start at line 327 in our runtime. In general, the functions representing the actions of instructions are named for their namesake. The all take as paramaters a program state and any parameters required by the instruction. They all return a program state.

pushglobal() takes the name of a super combinator. It finds the address of the super combinator by name in the globals array of the state and pushes this onto the stack of the returned state.

pushint() takes a literal integer and allocates an NNum node in the heap to represent it. The address of the new node is pushed onto the stack of the returned state.

mkap() takes no additional parameters. It pops the top two addresses from the stack of the state passed to it. A new NAp node is created with these two addresses as parameters and is inserted into the heap. The address of the new node is pushed to the stack and both it and the updated heap are returned in a new state.

push() takes an integer representing an index in the stack. The address located at this index is pushed to the top of a stack which is returned in a new state.

pack() is used to create NConstr nodes, representing data constructors in our heap. The function takes two integer as additional paramaters, a tag and an arity respectively. We assume as many elements of the data constructor as indicated by the arity parameter are addressed by the top items on the stack. We pop these from the stack and use them along with the tag to create an NConstr node, which we insert into the heap. The new heap and stack are returned in a new state.

casejump() takes as an additional argument an associative array of integers and code sequences. The integers represent tags in data constructors. The code sequences contain the sequences of instructions to undertake if the associated tag is encountered. The function expects to find a constructor tag address on top of the stack. The node pointed to by the top address is found in the heap and its tag extracted. The code sequence associated with that tag is located in the array and is added to the existing code sequence in the program state. This new code sequence is returned in a new state.

split()

update() is used to overwrite the root of a reducible expression with its value after it is first evaluated. This corresponds with the typical update operation describe in the background section. The function takes an integer N as an additional paramater. It will

pop() takes an integer N and pops N items from the stack. The resultant stack is returned in a new state.

evalInst() corresponds to the Eval function but is so named because of naming conflicts with JavaScript's built-in eval method and another similarly named function in our runtime. Eval is used to evaluate an expression to weak head normal form, at least in theory. In practice, it defers the current stack and code state to the dump and replaces them with a single pointer to the expression to be evaluated and a single Unwind instruction. These are returned in a new state.

The next two functions handle boxing and unboxing of integers in NNum nodes. boxInteger() takes a value N and creates an NNum node representing this value. This is added to the heap and its address added to the front of the stack. unboxInteger() takes the address of an integer node, finds it in the stack and returns the integer value it contains.

boxBoolean() takes a boolean value and converts it to an integer representation. There is no boolean node representation in our compiler which is fine as one isn't needed. The function creates an NConster of tag 1 to represent true, 2 to represent false and adds it to the heap. The address is pushed to the front of the stack and both are returned in a new state.

Functions primitive1() and primitive2() are used to simplify arithmetic operations. They handle the application of monadic and dyadic operators respectively. Both accept as parameters:

1. A boxing function, to place values in nodes and add them to the heap.

2. An unboxing function to extract values from nodes.

3. The operator in question to apply.

4. The current state

using these functions greatly minimizes the code we need to write for the functions representing our arithmetic and equality checking instructions. A number of such primitives are implemented in the runtime, all with much the same format; functions that take a state, call the relevant arity primitive handling function and pass to it a function representing the JavaScript operator we wish to use.

A function cond() handles conditionals. This takes two instruction sequences as well as the usual state and expects to find the address of an NConstr node on top of the stack. This node is found in the heap and its tag checked. As with boxBoolean(), a tag of 1 represents true, 2 false. Based on the tag, the appropriate of the two instuction sequences passed as parameters if added to the current code sequence and returned in a new state.

Lastly, the unwind() function is used to

6.4.2 Example Evaluation

We will now provide an example of program evaluation in our runtime. As before, we will use our "K" program. For brevity's sake, we will describe the states we encounter informally or in Haskell as their JavaScript representations are less readable. When last we saw our program it consisted of an empty stack and dump, a heap and globals array containing 2 items each, and a code sequence of PushGlobal main, Eval. We start by calling the main function which in turn calls evalProg on our initial state. We then iterate through each state.

- Iteration 1: PushGlobal instruction encountered, find address of "main" super combinator passed as argument and add to stack.
- Iteration 2: Eval function encountered. Defer state of stack and code sequence to dump. Code is empty and stack contains only the address of "main" so dump is updated to [([],[])]. Unwind instruction added to code sequence.
- Iteration 3: Unwind finds pointer to WHNF super combinator "main" and places its code sequence in GmCode.
- Iterations 4, 5: "K" is being called on integers 1 and 2. PushInt 1 and 2 add the appropriate nodes to the heap and their addresses to the top of the stack. Globals also updated.
- Iterations 6 - 8: Address of super combinator "K" pushed to stack. Two subsequent Mkap instructions add two NAp nodes creating a graph of "K", 1 and 2.

- Iteration 9,10: Update 0 instruction creates NInd indirection node to heap node 6. Pop 0 pops no addresses from the stack. Both trivial operations in this particular example.
- Iterations 10 - 13: Series of Unwind instructions. Firstly redirects trivial NInd node by replacing its address on the stack with that of its pointer (address 6). Places left arguments of both Mkap nodes on top of stack. When Unwind encounters pointer to ”K” on stack, its code is added to the code sequence.
- Iteration 14: Push 0 instruction at head of code sequence pushes pointer to first argument to ”K” onto stack. This is the actual behaviour of ”K”.
- Iterations 15 - 20: Eval instruction following ”K” checks the return argument is in WHNF as before. Update 2 instruction overwrites the poitner to ”K” with the value of its evaluation. Pop 2 instruction removes the argument pointers passed to ”K” from the stack. Series of Unwind instructions follows, until we are left with a pointer to a single WHNF node on top of the stack.
- Iteration 21: Pointer to NNum 1 on top of stack. No further code to evaluate. Final state.

At this point, the GmCode part of our starting GmState is empty. GmHeap will contain a number of additional nodes compared to its starting state as we added nodes representing the parameters passed to our K function, applicator nodes to form a reducible graph expression and indirection nodes to aid in updating. GmStack will contain a single item pointing to the return value of evaluating our program, in this case an NNum node of value 1. This value will be extracted and returned by evalProg().

6.5 Specific Feature Implemenations

When we encounter a dyadic expression application in our input program, we check if the expression being applied is a built-in primitive defined in our compiler. If so, we know that we are compiling in a strict context. We compile the arguments to the operator in this same context and place the instruction representing our primitive operation after the compiled arguments in the code sequence. If the application is not recognized, we assume it is a user (or possibly prelude) defined supercombinator and compile it in a lazy context. This will result in a code sequence containing a Mkap instruction to construction an application node in the graph the arguments to the application, compiled in the same context. With one exception there are no non-dyadic operations defined in our compiler, so non-dyadic expression applications will default to the lazy compilation scheme. The exception is the negate function, a unary primitive operation.

Data constructors are represented using EConstr expressions in the abstract data type of our input language. These are treated as applicable expressions in the compiler, not unlike super combinators or primitive operations. When we encounter an application, we first check if the expression we are applying is an EConstr. This is done by recursively checking its arguments for the presence of an EConstr. If so, we compile its arguments and follow them in the emitted code sequence with a Pack t n instructions, where t will be the tag of the constructor and n its arity. It is presumed that the previous n items in the code sequence will be its arguments.

Case expressions are represented in the input language as ECase expressions, which take an expression and a series of case alternatives. The expression will evaluate to a data constructor whose tag will correspond to one of the case alternatives. The expression will be compiled ahead of the alternatives and its evaluated result placed on the stack. When the casejump instruction is encountered in the runtime, the data constructor will be found in the heap and the code from the compiled case alternative corresponding to its tag placed in the code sequence.

6.6 Others

There are a few other aspects of the implementation that are probably worth mentioning. A prelude for the compiler exists in GPrelude.hs. This would normally contain a list of prelude definitions but for testing purposes it currently contains a number of test program definitions. This would be useful for anyone looking to test the compiler. An evaluator written in Haskell exists in GEval.hs. This was written concurrently with the runtime, as it helped greatly in diagnosing errors which were somewhat vague in JavaScript. Lastly, in order to actually use the compiler, catalyst.hs provides a command line interface. It can be ran from the linux command line with the -h flag which will return some helpful information and further instructions.

7 Evaluation

We have examined the design and implementation of our solution. We will now evaluate its usefulness and the degree to which it solves the problems we identified earlier. We have already seen an example compilation and evaluation of a simple program containing our "K" function. This program demonstrated a number of the solutions we wished to implement.

7.1 Completeness

Firstly, we were capable of taking a representation of a functional language derived from Haskell and compiling it into an initial state. *Note that I say "representation" as the compiler is not currently able to take Core in its unparsed form. This point will be further explored below.* This representation is capable of expressing a number of features, of which we support the following:

- Primitive integer values
- Conditionals
- Data constructors
- Case statements
- Expressions built from the above

The input language also supports let expressions, lambda abstractions, characters and strings however our compiler is not currently able to handle these. It would not be particularly difficult to add support for these however there simply wasn't enough time. Information on possible solutions will be provided in the further work section. We may not have a Core parser but the representation we do accept is general enough to represent the majority of basic functional language concepts. We originally intended for this project to use Haskell as its base but it would not be a massive step to adapt our project to languages such as Miranda, ML and Lisp. Such languages can easily be traced back to polymorphic lambda calculus dialects [10, pp.10] and as such can all be expressed, albeit in somewhat minimal forms, in representations very similar to that of our input language.

The initial state we compile this representation into is that of the Gmachine compilation scheme. This gives us a heap representing our program from which our evaluator can build a reducible expression graph. It also provides a stack and state dump which will be required by the runtime evaluator. Our compiler can then serialize this state into an equivalent JavaScript represention which we can evaluate in a browser, given the right set of abstract definitions and methods.

These definitions and methods exist in our runtime, also written in JavaScript. This runtime takes our serialized initial state and, as demonstrated, evaluates it through a number of subsequent states until a final state is reached. This final state represents our fully evaluated program and from this we can extract the result of evaluation. The features described above can now be written in a functional language representation and, after compilation and serialization, be executed in a browser.

In order for our project to be a viable option when writing real-world programs, we would need to add a number of basic browser interation components. We can currently evaluate expressions but not effect the state of the browser. Possible solutions to this will be outlined in the further work section.

7.2 A Core Parser

There is currently no functionality in my project to parse GHC's external core into the representation provided in GADT.hs.As deadlines approached, features yet to be implemented had to be prioritized. Writing a parser for Core was deemed to be of less importance and interest than other features. Core is designed to be easily parsed. Its specification has changed subtley but frequently over the last decade through various updates to GHC. Completed parsers exist for previous versions of Core however sometime between version 5 and the current version of GHC, maintenance of the core language becamse less of a priority for the development team. As a result, tools such as the parser stagnated. Writing a core parser would not be a difficult task and will be explored in the further work section below. However, it was a task deemed uninteresting when there existed other outstanding work to be completed which was more directly related to the goal of this project.

7.3 Efficiency

Efficiency was not the main aim of this project. As such, there exists a lot of scope for optimization and improvement. The simple examples provided will execute in acceptable time in a browser. However, the execution time of more complex programs will certainbly become noticeably and problematically slower than programs with equivalent semantics written in optimized

JavaScript. It would not be possible to achieve what we wanted and compete directly with the efficiency of native JavaScript code. This is an inherent flaw of trying to represent alternative semantics in a target language. This aside, there is still much unexplored potential in make our solution more efficient. The compilation scheme we chose is not in-efficient, but more modern schemes have been developed since which are more efficient. The JavaScript code used in the runtime could also stand to be more efficient. When implementing the runtime i was most concerned with producing a clean, understandable proof-of-concept. There is currently no garbage collection in my implemenation. Possible solutions to these will be examined in the further work section.

8 Further Work

The following are outlines for solutions to some of the problems identified in our implementation.

8.1 External Core Parser

As mentioned in the evaluation section we provide no means of parsing GHC's external core into the representation our compiler accepts. Implementing this parser is not especially difficult. The Haskell algebraic data type we use as our input language representation is almost identical to the official definition GHC provide for their own algebraic data type representing Core. Examples of parsers written for previous versions of Core can be found in old versions of GHC. As Core and its tools have not been well maintained in recent versions, these parsers do not quite fit the corre language specification. However it should be entirely possible to adapt the last such parser to accept current core and convert it to into our algebraic data type representation. Even if this is not possible, Core is designed to be parseable and is built from a very small set of components. Writing a parser from scratch would be a little time-consuming, but not especially difficult.

8.2 Characters, String

Adding characters and strings would not be particularly difficult. We could add a node type to our Gmachine representing characters if we wished. Alternatively, we could represent characters as their ascii integer representations with the existing NNum node type. Strings could be built in much the same way we currently handle lists; A string could be constructed as a list of character nodes (or NNum node character represenations) concatenated together.

8.3 Browser Interaction Methods

In order to allow for the writing of useful browser executed programs our language would need a set of primitive operations corresponding to common browser actions. Such actions would include DOM manipulation methods, input handling etc. A number of such examples exist in the DSL of Fay examined earlier. There are two obvious ways we could add such functionality to our compiler. The first would be to add instructions to handle primitive operations. Much like the existing instructions for arithmetic, equality etc. these would be emitted when we encounter certain keywords in our input language. A list of supercombinators representing the browser actions we wish to represent would be added as compiled primitives to our compiler. When our runtime encounters these instructions it would convert them into the appropriate browser actions to execute.

Alternatively, we could write a DSL in the Haskell we compile to represent these actions. In much the same way as Fay, we'd use Haskell type and data type definitions to correspond to the browser actions we wish to represent. Our compiler and runtime would maintain a list of these definitions and the browser actions they correspond to. When the runtime encounters such a definition, it would execute the corresponding browser method.

8.4 Better Compilation Scheme

We investigated two compilation schemes before implementing our compiler. The latter of these, the Gmachine, was chosen for its various advantages as outlined in the design choices section. However, there exist other schemes which may suit the purposes of this project better. Of note is the Three Instruction Machine, a compilation scheme which uses only three instructions in its runtime and opts for a *spineless* graph representation in the heap. Such an implemenation would be of obvious interest to us, as it would allow for a very minimal runtime. This would improve load times for pages using our compiled language as there would be less data to transmit when loading the runtime. However, this comes at a tradeoff. The initial state is more complicated and evaluation will require more steps than our Gmachine implemenation. Such considerations would ultimately decide the choice of best compilation scheme for our purposes.

8.5 Garbage Collection

A significant issue in our implementation is the lack of a garbage collection system for our heap. This would prove to be quite problematic (even fatal) for any large program we wish to run. A lot of research has gone into functional language garbage collection and i would found it an intersting feature to implement. Sadly there was not enough time and it was decided that a compiler without a garbage collector would be of more use than a garbage collector without a compiler. A garbage collection implementation exists for our Gmachine implementation in the Jones and Lester implemenation tutorial. Further work related to garbage collection would probably start there, unless an alternative compilation scheme was used.

9 Conclusions

I am mostly satisfied with the outcome of this project. The set of features my compiler and runtime are capable of compiling and evaluating are sufficient to express most basic concepts one would expect in a functional language. With data constructors we are able to express lists and user-defined types. Our Case statements and conditionals can be used to define more complex flow control such as loops. Characters and strings are currently absent but as discussed in the evaluation and further works sections their addition would be complicated. I'm somewhat disappointed at the lack of garbage collection as this was a feature I'd hoped to have a chance to experiment with. As mentioned, the lack of a core parser is unfortunate but does not detract from the more interesting aspects of the project.

When I started this project, I was somewhat proficient at Haskell but could have benefitted from more experience writing it. This project provided that experience and improved my confidence with the language. My JavaScript abilities have improved greatly as a result of this project, having been quite weak at the beginning. I originally had no experience with functional compilation and as a consequence of the research conducted throughout it is now a topic in which I am quite interested. I look forward to further reading on the subject and hope that I will get another chance to apply some of what I've learned about functional compilation and functional languages in general.

Were I to start this project again, I probably would have implemented a basic compiler as early as possible. I somewhat underestimated the scope of writing a compiler for even a minimal functional language and in the end this cost me. Writing the JavaScript runtime was particularly arduous. Having to switch between writing JavaScript and Haskell really made obvious the benefits of writing in a langauge with a strong type system. If I was to write the runtime again, I would probably investigate ways of modelling strong typing in JavaScript through its use of prototypes etc. I would also have made use of some kind of testing framework. In reality, this would probably have meant rolling my own as JavaScript testing frameworks have proven themselves unhelpful to me in the past. While this would be time consuming it would have reduced many of the headaches I experienced as a consequence of trying to track down JavaScript bugs.

References

[1] Eddy Burel and Jan Martin Jansen. Implementing a non-strict purely functional language in javascript, 2010.

[2] CoffeeScript. Coffeescript. http://coffeescript.org/.

[3] Doug Crockford. Javascript: The good parts.

[4] Fay. Fay wiki and repository. `https://github.com/faylang/fay/wiki`.

[5] GHC. Ghc core definition. `http://www.haskell.org/ghc/docs/7.6.2/html/libraries/ghc-7.6.2/CoreSyn.html`.

[6] Paul Hudak. Modular domain specific languages and tools.

[7] Simon Peyton Jones. The implementation of functional programming languages, 1987.

[8] Simon Peyton Jones and David Lester. The implementation of functional programming languages: A tutorial, 1991.

[9] Simon Peyton Jones and Simon Marlow. Secrets of the glashow haskell compiler inliner, 1999.

[10] Martin Sulzmann, Manual M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions, 2011.

[11] Andrew Tolmach. An external representation for the ghc core language, 2001.