# Browser-side Functional Programming

Second reader demo

Kieran Manning
09676121

April 3rd 2013

## An Overview

The Problem?
- Javascript is unpleasant but ubiquitous. Javascript Suffers from..

- A lack of sane typing
- Verbose and inconsistent syntax
- A lack of laziness

These have been solved in languages such as Haskell, however it looks like we're stuck with Javascript for the forseeable future. What we need is a way of bringing some functional inspiration to browser-side programming via Javascript.

## Implementation

An approach is needed that would...

- Bring lazy evaluation and type safety to Javascript
- Be modular, packageable.
- Be as efficient as possible.
- Allow easy seperation of runtime evaluation from compilation
- Require no changes to Javascript itself

## Core

Name given to a number of intermediate functional language representations descended from System F lambda calculus, consisting notably of...

- Super combinator defintions and applications
- Variable names
- Integers
- Data constructors
- Case statements

GHC is capable of outputting a core-like language (see GHC external core) which is lazy and type-checked by GHC itself.

## The G-Machine

The G-Machine, a functional compilation strategy.

- Historically significant but somewhat outdated.
- Takes a core-like language and produces an initial graph state. and sequence of instructions...
- ...to be evaluated by a minimal runtime.
- Designed to allow for easy adaptation of runtime in target languages.
- More efficient and modular than alternative Template Instantation approach, also explored.

## A Graph Evaluation Runtime in Javascript

A runtime was required which would fulfill the following requirements:

- Take a representation of a G-Machine compiled graph state.
- Take a list of G-Machine evaluation instructions.
- Understand how to evaluate these instructions.
- Apply these instructions to the supplied graph and
- Return the resultant state of the graph.

## Implementation, an overview

1. Taking GHC's external core, we parse** that into a Haskell ADT representing the language.
2. We then use the G-Machine compilation schemes to parse this ADT into
   - A heap representing a graph and a stack to index this heap
   - A sequence of instructions to be executed by our runtime.
3. We package this state and instructions into a form javascript can handle and then pass to the runtime.
4. The runtime, written in javascript, executes until we have reached a final state (or something breaks...) leaving us with a final value on top of the stack.

## G-Machine implemenation

- Implementing a G-Machine based graph compiler is a solved problem, and rather than re-inventing the wheel I decided to work off the implementation given in Implementing Functional Languages: A Tutorial by Simon Peyton Jones and David Lester.
- Once our core representation has been compiled to a graph, it still needs to be serialized into a form that Javascrpt can handle. Haskell2JS.hs takes a state consisting of a stack, heap, list of globals and sequence of instructions and returns the equivalent represented using Javascript arrays and objects.

## Runtime implementation

Initial considerations

- Data representations?
  - Function objects to represent instructions, nodes:
  ```
  var x = new function PushInt(Int){this.n = Int;}
  ```
  - Arrays, objects to represent our state
  ```
  var GmHeap = {objCount, freeAddrs, addrObjMap};
  ```
- Instruction evaluation?
  - JS functions operating on a passed state
  ```
  function pushint(n, state){push, return state';}
  ```

## Runtime evaluation 1

Initially, we have

- A heap, containing compiled supercombinators representing our main function, additional functions and language primitives ( $+$, -, if ...)
- An empty stack
- A code sequence consisting of

  ```
  [ PushGlobal "main", Eval ]
  ```
- And a list of globals, mapping function names to heap addresses

  ```
  [("+", 5), ("main", 1)...]
  ```
- A dump to handle arithmetic operations when not in WHNF

## Runtime evaluation 2

An overview of our instructions

- Slide, Push, Pop etc. effect the stack directly, (de)allocating pointers to the heap as necessary. Update of interest.
- Mkap creates an application node between two heap items.
- Push{Int|Global} index new or existing values in the stack
- Unwind crawls the graph, rearranging as necessary (directing indirections, applying applications etc.)
- Eval handles non-WHNF operations, deferring to the dump until we can return a WHNF item to the top of the stack.

Nodes

- Indirections, @s, Ints, Globals, Constructors

## Runtime evaluation 3

Our Evaluation Strategy

- Encounter Pushglobal "main", push addr to stack
- Eval and Unwind stack, place main instructions in code sequence
- Push arguments of first function in main to stack, followed by behaviour of first function and sufficient Mkap instructions as to form an application tree.
- Defer state to dump if necessary.
- Reduce this expression and update root node with value
- Rinse and repeat until code sequence empty.

## Conclusions

Some problems

- Core versioning issues
- Time management
- Javascript in general (if only someone would write...)

Had I more time...

- Cleaner heap implementation
- More efficient graph compiler implementation
- More efficient Javascript
- Core parser would be nice but unexciting
- Continue runtime to completion (although it's not far off)