Browser-side Functional Programming

Second reader demo

Kieran Manning
09676121

April 4th 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

Existing Approaches

- Fay
  - Subset of Haskell
  - Lazy, pure, static typing
  - Compiles to JavaScript
  - Runtime operates at thunk level
- iTasks
  - Toolkit for workflow support applications
  - Written in Clean
  - Compiles SAPL into JS workflow apps
- ClojureScript - Compiles clojure into readable JS
- CoffeeScript - JavaScript wrapper language

A Plan Forms!

- Accept Core-level language a la iTasks, take advtange of GHC.
- As with both iTasks and Fay, we will be working in a language which already provides type safety and laziness.
- Aim to provide features from which we can build. Primitive values, function definitions, conditionals, data constructors. Ignore I/O etc. for purity and simplicity.
- G-Machine compiler implementation to compile our parsed Core representation into an instruction level language.
- Lastly, as with Fay, we will build a runtime in our target language to evaluate these instructions while preserving the lazy, type-safe semantics of our input.

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.

What does GHC ext-core look like?...

fac 0 = 1; fac n = (n * fac (n - 1)) becomes...

```
{fac :: Int -> Int =
  \ (d0::Int) -> %case Int d0 %of (wildX4::Int)
  {Types.Izh (d1::Int) -> %case Types.Int d1 %of (d6::Int)
  {%_ ->
  * @ Int NumInt wildX4 (
    - @ Types.Int Int (fac wildX4)
      (Types.Izh (1::Int)
    );
  (0::Int) -> Types.Izh (1::Int)}}};
  main :: Types.Int = fac (Types.Izh (4::Int));
```

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. We take GHC's external core, which we parse into a Haskell ADT representing the language.
2. We then use the G-Machine compilation schemes to compile 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: Initial State

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

Shall show such an example at some point...

Runtime Evaluation: Instructions Overview

The expected instructions for a G-Machine runtime. Slide, Push etc. to manipulate stack. Mkap, Eval, Unwind etc. to evaluate expressions. All executing on a passed state and returning a new state...

```
function push(N, xState){
    var State  = xState; // <- JS state issues
    var stack  = getStack(State);
    var newStack = [stack[N]].concat(stack);
    var newState  = putStack(newStack, State);
    return newState;
}
```

Runtime Evaluation: Nodes Overview

Our runtime must accept and evaluate the usual G-Machine graph nodes: indirections, applications, ints, globals, constructors. We represent these in the abstract as functions which we can instantiate as function objects representing graphs nodes, values.

```
function NConstr(t, a){
    this.t = t;
    this.a = a;
}
```

```
[newHeap, addr] = hAlloc(heap, new NConstr(t, addrs));
```

Runtime Evaluation: Evaluation Strategy

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.

Features

- Laziness
  Our runtime supports laziness, including redex updates and call-by-need evaluation.
- Function defintions and calls
- Arithmetic
- Conditionals
- Data Constructors (...kinda)

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)