Programming Language Design: Progress Report

Sam Auciello

08 December 2012

The main project of my plan of concentration is to study programming language design by creating a language. I have chosen to create a Lisp dialect aimed at transcompiling to and interoperating with JavaScript in a similar manner to CoffeeScript. I will call my language Hot Cocoa Lisp. In this paper I will describe my progress this semester on the project.

Language goals

The language has the following goals:

- intuitive interoperability
- powerful lisp features (macros, homoiconicity, etc.)
- fix many of the "bad parts" of JavaScript
- provide intuitive syntactic sugar

The first goal of Hot Cocoa Lisp is to cleanly interoperate with JavaScript. APIs written for JavaScript should be as natural as possible to use within Hot Cocoa Lisp and vice versa. Obviously there will be differences but the goal is for the translation be as unsurprising as possible with clear intuitive conventions. All of the "good parts" of Javascript should be exposed.

Lisp macros may be particulary useful in the context of a transcompiled language. Hot Cocoa Lisp will provide the power to extend the language using macros while having more control over the resulting source. It's not clear at this point how powerful macros will be.

Douglas Crockford has famously discussed the good and bad parts of JavaScript. One of the most important goals of Hot Cocoa Lisp is to mitigate the bad parts without losing any of the good parts. Specifically, my compiler will implement good practice with regard to global variables and semicolon insertion. Because both my language and JavaScript will be dynamically typed, there is only so much that can be done about type coersion but I will have separate functions for addition and concatenation (the latter of which might even explicitly call *.toString* on all of its arguments). The remaining bad parts are largely just language features which can be avoided.

To suplement the pure reverse polish syntax of Lisp, my language will contain JavaScript inspired syntactic sugar for list literals, object literals, and object access. Specifically:

```
;; list literals
[1 2 3] ; syntactic sugar
(list 1 2 3) ; parsed as
[1, 2, 3] // resulting JavaScript

;; object literals
{a 1 b 2} ; syntactic sugar
(object a 1 b 2) ; parsed as
{a: 1, b: 2} // resulting JavaScript

;; object access
object.parameter ; syntactic sugar
(. object parameter) ; parsed as
object.parameter // resulting JavaScript
```

Because of the way that methods work in javascript, whenever the . function would return a function it will wrap that function in a function that automatically calls it in the context of the object. This function will then have it's call and apply methods overloaded to allow the context to be respecified. This will look something like:

```
var _wrap_method = function(func, obj) {
  var new_func = function() {
    return func.apply(obj, arguments);
  };
  new_func.apply = function() {
    return func.apply.apply(func, arguments);
  };
  new_func.call = function() {
    return func.call.apply(func, arguments);
  };
  return new_func;
};
```

Call and apply will then be further exposed as bare functions in Hot Cocoa Lisp:

```
(call console.log console 1 2 3) ; Hot Cocoa Lisp source
console.log.call(console, 1, 2, 3) // Javascript result

(apply console.log console [1 2 3]) ; Hot Cocoa Lisp source
console.log.apply(console, [1, 2, 3]) // Javascript result
```

Implementation

Over the course of this semester the specification for the language has become increasingly clear in my head. What remains is to write that specification down, work out the details, and implement the language. These three things will likely happen concurrently. My implementation is at a fairly early stage. I have made several early passes at implementation much of which will ultimately be refactored/rethough/scrapped over the next few months. I have made a working general purpose scanner, parser, and semantic analyzer which can currently convert raw Hot Cocoa Lisp Code to an abstract syntax tree including the peviously mentioned syntactic sugar. There is a decent chance that I will actually end up replacing this with a simpler parser designed specifically for the language however that process will be made simpler by the tests I already have.

I have also begun writing the compiler which should be fairly simple. The comiler will first run all macros on the code, then interpret the code in the following way: The *js* function will be implented to insert a formatted string of JavaScript code. Any other function will be translated from the form *(funciton arg1 arg2)* to the form *function(arg1, arg2)*. Most of the language will then be implemented in macros. This means that much of the work will be in adding functionality to the macro system. Additionally I would like to add the ability to specify code that will be prepended to either the statement or the entire program so that for example:

```
(map (# (x) (* x x)) [1 2 3]) ; '#' is the lambda function
```

can become

```
var map = function(func, list) {
  var result = [];
  for (var i = 0; i< list.length; i++) result.push(func(list[i]));
  return result;
};

map(function(x) { return x * x; }, [1, 2, 3]);
```

I have also made a basic testing system using a combination of ruby and node. This system lets me specify pairs of javascript expressions that should be equal and then tells me which ones, if any, aren't. It has a number of useful features. It can detect files based on convention and automatically run all of the tests in my project. I can wrap error prone tests in anonymous functions which will then automatically be run in a try block which displays any error message without interupting the rest of the tests. Objects and arrays are automatically tested for equality using a recursive comparison.

Other topics

This semester I have also looked at a few interesting topics in language design that don't relate directly to my design project. I've loooked into a few parsing algorithms and implemented both recursive descent and CYK in the general case. I spent a lot of time thinking that my language would be interpretted and thus spent a fair bit of time looking into related implementation details. I spent a lot of time looking at implementing type systems with wrapped values using inheritance models before decided that my language would ultimately just have the same types as JavaScript. Also looked into ideas like L-values and R-values, tail recursive optimization in interpreters, and continuation passing as a control flow tool.

Conclusion

This semester I've made significant progress in my plan project while gaining a general grounding in the subject of language design. The most important major design issues have been mostly resolved and I'm ready to move forward with implementation details. Next semester I hope to finish and document the project. The field of programming language design seems to be a large and complex one and although this project only scratches the surface of it, I hope to come out of it with a much better understanding the nature programming languages and programming in general.