October 30, 2013

Lambda Calculus, Javascript and Functional Programming

Prepared For:

Mantis Cheng, CSC 330
University of Victoria

Written By:

Jasmine Mago, Freja Kelleris
Laura Rossetto, Yuma Katayama
Samuel Navarrete, Daniel Oon
Josh Stelting, Max Gunton
Trevor Schiavone, Breanne Gruenke

Table of Contents

1. History of JavaScript

1.1 Origin

Javascript was developed by Brendan Eich and was released in 1995. Initially, Javascript was developed under the name “Mocha”. When the beta release of Netscape Navigator arrived in September 1995, it was officially called “LiveScript”. Finally, in the Netscape Navigator 2.0B3 release in December of 1995, it was renamed “Javascript”. Javascript very quickly gained widespread success as a client-side scripting language for web pages. Microsoft introduced Javascript support in Internet Explorer 3.0 when it released August 1996.

In November 1996, Netscape announced that it had submitted Javascript to Ecma International for consideration as an industry standard. After some more work, the standardized version coined “ECMAScript” was produced. In June 1997, Ecma International published the first edition of the ECMA-262 specification. Brendan Eich commented on the standardized language, saying that “ECMAScript was always an unwanted trade name and sounds like a skin disease.”

1.2 Functional Characteristics and Lambda-Calculus

Function application is referred to as β-reduction, which basically means reducing arguments by applying them into a function. In the following example, the argument “`7`” is applied to a variable `n` and the whole expression turns into a simple expression.

$$((\lambda n.n \times 2)\ 7) \rightarrow 7 \times 2$$

In JavaScript, it reduces the argument first so we can take a function as a parameter.

The term “functional” simply describes a collection of traits that a language may or may not have. For example, Haskell has them all: immutable variables. pattern matching, first class functions, and others. Some languages have hardly any, like C. Javascript does not have immutable variables or pattern matching, but it does have a strong focus on first class functions.

When a language like Javascript has “first class functions”, this means that the functions are objects themselves. Because of this, they have methods and properties, such as `.call()` and `.bind()`. This also means that functions can be passed into other functions as well as the ability to have nested functions. Javascript makes use of what are called “anonymous functions” which is a direct offshoot of lambda calculus. For example:

```
1.  function(x, y) {
2.    return x + y;
3.  }
```

The only difference between a regular function and an anonymous function is in the name. You don't strictly need to give a function a name if it will be passed around. For example:

```
1.  // Function to calculate a total
2.  var CalcTotal = function(x, y) {
3.     return x + y;
4.  };
5.
6.  // Function to add taxes
7.  var AddTaxes = function(x) {
8.    return x * 1.2;
9.  };
10.
11. // Function that pipelines the other two
12. var CalcTotalPlusTaxes = function (fnCalcTotal, fnAddTaxes, x, y) {
13.     return fnAddTaxes(fnCalcTotal(x, y));
14. };
15.
16. // Execution
17. var result = CalcTotalPlusTaxes(CalcTotal, AddTaxes, 40, 60);
18. alert(result);
```

In the above code, `result` is a value returned by `CalcTotalPlusTaxes`, which takes in two functions as parameters. This is a very good example of how Javascript can pass in and return functions. This method is called "currying" which it to take multiple argument into a chain of functions and returns a single argument. In JavaScript, we are allowed to treat a function as an argument.

In the lambda calculus, the abstraction operator, λ, is said to be a binding of variable in the body of abstraction. Variables that fall within the scope of a function are called "*bounded*" variables whereas all the others are called "*free*" variables. A bounded variable is bounded by the nearest abstraction. Similarly in JavaScript, whenever there occurs a nested function, it stores the local references to the local variable that is present in the same scope as the function itself, even after the function return.

```
1.  function send(name) {
2.      // Local variable 'name' is stored in the closure
3.      // for the inner function.
4.      return function () {
5.          sendHi(name);
6.      }
7.  }
8.
9.  function sendHi(msg) {
10.     console.log('Hello ' + msg);
11. }
12.
13. var func = send('People');
14. func();
15. // Output:
16. // Hello People
17. sendHi('World');
18. // Output:
19. // Hello World
20. func();
21. // Output:
22. // Hello People
```

In the above example, the second call of `func` returns the same result as the first call because the passed parameter, `name`, for the outer function `send` is a local variable that is stored in the closure for the inner function `sendHi`.

---

2. Lazy Evaluation

2.1 Introduction

Lazy evaluation is an evaluation strategy in which the expression is not evaluated until it is needed. This means that lazy evaluation can help Javascript evaluate only the results of interest from an infinite loop, for example. It can also improve Javascript's expressiveness in string processing as it will not be required to declare an array of an adequate size to store the raw string, making it easier to manipulate. In general, lazy evaluation leads to enhanced expressiveness and performance in instances that would involve with large arrays otherwise.

An important benefit of lazy evaluation is the ability to create infinite data structures. Javascript libraries like jstream provide infinite list capabilities as such. This is done in a way similar to the Haskell programming language, through expressions with the head and tail of the list.

2.2 Infinite Lists

How is lazy evaluation important to infinite data structures?

Passing functions as parameters without immediately evaluating them can be used to implement an infinite stream. Take for example the following javascript code snippet:

Note the use of the stream.js library which provides a stream generator.
For example:

```
new stream( head, return_tail())
```

```
1. //Returns an infinite list of ones 1, 1, 1, …and so on.
2. function infiniteOnes(){
3.     return new Stream(1, infiniteOnes);
4. }
5.
6. //Returns an infinite list of natural numbers 1, 2, 3, 4, ….and so on
7. function naturalNumbers(){
8.     return new Stream(1, function () { return
   infiniteOnes().add(naturalNumbers()) } );
9. }
```

The second parameter (underlined above) of `new Stream` is not the tail but a function that returns the tail. Using a strict method of evaluation would instead produce a non-terminating function evaluating the tail over and over. The importance of lazy evaluation becomes clear.

The `naturalNumbers()` function should return an infinite list of natural numbers 1, 2, 3 ..and so on, but computers are only finite. This is possible because all the natural numbers are not yet created. They will be created once we need them. While this representation of an infinite list is only abstract, it allows for new ways of constructing expressions. This contributes to the overall expressiveness of the programming language.

For example:

```
naturalNumbers().take(10).print();
```

The `take(n)` function returns the first `n` elements from a stream.
When applied to our numbers stream it prints 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, the first 10 natural numbers.

2.3 String Processing & Lazy Implementation

Javascript's String.match and String.split returns an array of substrings. In the case that a string is very large, using String.split will store every substring. This is more work than necessary If you are looking to obtain the first few lines of a block of text. Because every substring is stored, this is also inefficient. With lazy evaluation, the string could be split and even be filtered, mapped, reduced, etc. before being evaluated or stored. This means that when obtaining the first few lines it would only store said lines. It would not evaluate the whole block of text beforehand.

There is a work in progress for a lazy Javascript called Lazy.js. It is essentially a utility library similar to Javascript's Underscore and Lo-Dash though it is still experimental. As discussed above, obtaining the first 10 lines, for example, will be done easily and efficiently with Lazy.split and Lazy.take without iterating over an in-memory collection. Lazy.js also supports indefinite sequence generation and stream processing.

Here is a line to get the first 5 lines from a block of text with Lazy.js:

```
var firstFiveLines = Lazy(text).split("\n").take(5);
```

---

3. Higher-Order Functions and Closure

3.1 Definition of Closure

A closure is a function combined with a table storing a reference to each of the free variables of that function, called a referencing environment. A variable is free if it is used inside a function where it is not declared. A closure allows a function to access those free variables even when invoked outside its immediate scope.

The concept of closures was developed in the 1960s and was first fully implemented as a language feature in Scheme. The use of closures today is associated mainly with functional programming languages. In traditional imperative languages, closures are not necessary as these languages do not have as much, if any, support for higher order functions.

Closures are usually implemented with a data structure containing a pointer to the function, plus a representation of the set of available variables and their values. The referencing environment binds the nonlocal names to the corresponding variables in scope at the time the closure is created, which extends their lifetime to at least as long as the lifetime of the closure itself.

```
1. def Function1(a):
2.   def Function2(b): return a + b
3.   return Function2
4.  inc = Function1(1)
5.  print inc(2)
6.  sum2 = Function1(2)
```

In the above example `a` is free variable in the body of `Function2`. Normally once a function returns its value, the space reserved for its activation record is deallocated. However, this creates a problem since variable `a` in the example would no longer have a storage location once `Function1` has returned a value. To solve this, a closure needs to be implemented.

3.2 Higher-Order Functions in JavaScript

A higher order function is defined as a function that can take one or more functions as an input or output a function. The map function is a common example of a higher-order function. It takes a function and a list of elements, and returns a new list with the function applied to each element. Implementations of the map function in both Haskell and JavaScript are shown below. Other examples of higher-order functions include fold/reduce, function composition, and integration.

JavaScript does not have higher order functions built-in; however, its use of functions as first class objects makes writing higher order functions fairly simple. Also, there is a *Functional* library available which includes many of the common higher order functions, such as map and reduce. The implementation of higher order functions is a lot more verbose than in Haskell (Haskell has built-in higher order functions).

Here is a simple example of a map function in JavaScript (using an array rather than a list):

```
1.  function Map(array, func) {
2.      for (var i = 0; i < array.length; i++) {
3.          if(i in array) {
4.              func(array[i]);
5.          }
6.      }
7.  }
```

JavaScript Map function (using Functional library):

```
Functional.map = function(fn, sequence, object)
```

`Type: (a ix → boolean) [a] -> [a]`

```
map(f, [x1, x2…]) = [f(x, 0), f(x2, 1), …]
```

Example:

```
map('1+', [1,2,3])→ [2, 3, 4]
```

Haskell Map function:

```
map:: (a → b) [a] -> [b]
map f  []               =  []
map f (x:xs)            =  f x : map f xs
```

Example:

```
map (add 1) [1,2,3] => [2,3,4]
```

3.3 JavaScript Implementation of Closure

3.3.1 Background

When implementing closure in JavaScript you need a function and an environment in which that function was created. The environment consists of any local variables that are in-scope at the time that the closure (function) is created. First thing to note is that any function always sits within some scope (the scope in which it is defined). Different programming languages have different policies on what the actual scope of the variables is, but in Javascript the scope of a variable exists from the point it is defined in the source code until the closing bracket of that section. This includes any nested functions that are DEFINED (not to be mistaken with CALLED) in that scope and is called lexical scoping. This means that a function defined from within another will have access to the variables of its parent. And taking advantage of these "free variables" is the idea behind closure. Meaning that depending on what the values of the parent function's variables are will directly affect those free variables (bind them) within the "inner" function to the values of the environment at the time of definition. IIn short, variables from the parent function of the closure remain bound from the parent's scope.

A very simple example of this is the following:

```
1. function parentFunction() {
2.    var name = "CSC 330"; //name is a local variable to parentFunction()
3.    function innerFunction() { // innerFunction() is a closure
4.    alert (name); //uses variable declared in parentFunction()
```

```
5.     }
6.     innerFunction();
7.  }
8.  parentFunction();
```

As you can see in the above code, **innerFunction()** does not define any of its own variables, but instead just uses the variable **name** defined in its parent function. This means that when **parentFunction()** is called on line 8 **innerFunction()** becomes a closure because the only variable used in it is **name** which was previously bound to the value "CSC 330" on line 2.

Slightly modifying the above code we are able to return the closure itself and moreover create two distinct instances of them:

```
1.  function parentFunction(string) {
2.     var name = string;
3.     function innerFunction() {
4.        alert (name);
5.        }
6.        return innerFunction();
7.  }
8.  var myFunc = parentFunction("CSC 330");
9.  var myFunc2 = parentFunction("CSC 100");
10. myFunc();   //alerts("CSC 330")
11. myFunc2();  //alerts("CSC 100")
```

In this example **parentFunction()** acts more like a factory function (producing and returning distinct functions). Instead of having the **name** variable statically defined to "CSC 330" as before it gets its value from the parameter passed into **parentFunction()** which in turn returns **innerFunction().** Because there are different environments in which **innerFunction()** can be defined it means that there are also different closures depending on those environments.

Normally, the local variables within a function only exist for the duration of that function's execution. Once **parentFunction()** has finished executing, it is reasonable to expect that the **name** variable will no longer be accessible. But since this code still works, this is obviously not the case. The reason being is that JavaScript's garbage collector doesn't consider the current instance of **name** garbage until it has no more references to it, but the returned closure maintains a reference back to that instance of **name** as long as the closure exists. Therefore since **myFunc()** and **myFunc2()** are closures, when each closure was created the current bound value of the **name** variable was stored with that closure.

3.3.2 Uses for Closures in JavaScript

One of the greatest features of the JavaScript language is closure. Closures are continually useful when doing a wide range of different types of programming. They can be used to implement object factories, to hide private variables/functions, to replace objects, to implement event driven programming (events and callbacks), etc. Below some examples of these different uses are described in more detail.

To Make Generic Functions

Javascript can be used to make generic functions. In turn, these functions can be attached to events that differ only by environment. In doing so, event driven code becomes a lot easier to write.

Example:

```
1.  function makeSizer(size) {
2.    return function() {
3.      document.body.style.fontSize = size + 'px';
4.    };
5.  }
6.
7.  var size12 = makeSizer(12);
8.  var size14 = makeSizer(14);
9.  var size16 = makeSizer(16);
```

`size12`, `size14`, and `size16` are functions which can be used to resize text. Below is an example of how these could be used is to attach them to button, but as they are generic functions, they could be attached to many other events as well.

```
1.  document.getElementById('size-12').onclick = size12;
2.  document.getElementById('size-14').onclick = size14;
3.  document.getElementById('size-16').onclick = size16;
```

To Hide Private Methods/Variables

JavaScript does not provide a native way of doing this, but it is possible to emulate private methods using closures.

```
1.  var Counter = (function() {
2.    var privateCounter = 0;
3.    function changeBy(val) {
4.      privateCounter += val;
5.    }
6.    return {
7.      increment: function() {
8.        changeBy(1);
9.      },
10.     decrement: function() {
11.       changeBy(-1);
12.     },
13.     value: function() {
14.       return privateCounter;
15.     }
16.   };
17. })();
18.
19. alert(Counter.value());
20. Counter.increment();
21. Counter.increment();
22. alert(Counter.value()); /* Alerts 2 */
23. Counter.decrement();
24. alert(Counter.value()); /* Alerts 1 */
```

In this example, `changeBy()` is a private method which can only be accessed through the public counter methods, which all share the same environment. In this way, it emulates a private method, as it cannot be accessed directly by an outside function.

---

4. List Comprehension

4.1 Introduction

In this section we will discuss the differences and similarities between features of list comprehension in Haskell and array comprehension in Javascript. To conduct this comparison we will look at the concepts of generators, predicates and range in terms of how each language supports and applies them.

4.2 Generators

Both Haskell and Javascript make use of generators. In Haskell, generators are used to generate a sequence of elements that can be used as part of the list. An example of this is found in Figure 4.2.1.

```
[(x, y) | x <- [1..10], y <- [1..x]]
```

Figure 4.2.1

In this code, two generators are used to set the values for variables x and y. This format is known as multiple generators as more than one generator is specified in one statement.

In Javascript, generators are functions that contain the word yield. When a generator is called, the formal parameters are bound to the actual parameters. The body is not yet evaluated, but the generator iteration is returned. Each call to the generator function performs another iterative pass, where the value is specified by the yield keyword. In Figure 4.2.2, range() is the generator function. By repeatedly generating the iteration , we can obtain our desired result of all the values between the given bounds begin and end. This general concept can be applied to array comprehension to obtain desired conditions of a given array.

```
1. function range(begin, end) {
2.   for (let i = begin; i < end; ++i) {
3.     yield i;
4.   }
5. }
```

Figure 4.2.2

4.3 Predicates and Range

List comprehension in Haskell has the support of predicates. For example, we can have the following code in Figure 4.3.1:

```
Items = [ 3*x | x <- [0..10], x^2 > 5 ].
```

Figure 4.3.1

Here, for any value x that the statement x^2 > 5 does not evaluate true, the value of x will not be accepted.

Javascript does not directly include the application of predicates in array comprehension, however, similar procedures can be conducted using additional statements to check if a variable meets certain conditions. To do this an IF statement could, for example, be applied as illustrated in the code in Figure 4.3.2:

```
var items = [2*x for (x in 10) if x^2<5)]
```

Figure 4.3.2

As in the Haskell example above, this only accepts values of x for which the statement `x^2 <5` is true. In this way, Javascript does support predication however it is applied in a different method than in Haskell.

Like predicates, range is handled differently in Javascript than it is in Haskell. In Haskell the range of a list can be defined in the list statement itself. For example, the code in Figure 4.3.3 will apply the list statement for every value x from 0 to 10.

```
Items = [ 3*x | x <- [0..10], x^2 > 5 ]
```

Figure 4.3.3

This can not be used in Javascript. In Javascript, the range can be specified in a number of different manners. One option is to prototype the number objective of the range using for loops to specify it. Another option is to use generators to construct a range function for the array. The range function can be initialised as shown in Figure 4.3.4:

```
1. function range(begin, end) {
2.   for (let i = begin; i < end; ++i) {
3.     yield i;
4.   }
5. }
```

Figure 4.3.4

Including this module in our code allows us to use range() to construct an array with a upper and lower bound specified by the variables we provide. The code in Figure 4.3.5 illustrates the application of this range function in the creation of the list items where we consider values of x from 0 to 10.

```
var items = [2*x for (x in range(0,10)) if (x*x<5)].
```

Figure 4.3.5

4.4 Conclusion of Comparison

As outlined in sections 4.2 and 4.3 of this report, there are certain differences between Haskell list comprehension and Javascript array comprehension. While the two support very similar features there are distinct differences in how these features are handled and applied. It can also be concluded that Haskell is the more expressive language in terms of list comprehension. This is evident as list in haskell can be constructed in a single statement whereas Javascript sometimes requires separate statements and/or separate functions to fulfill the specifications of a given list. It this way Haskell is a lot simpler to comprehend as it does not usually require excessive description and definitions

5. References

Cornell University (2011) *Lecture 23: Streams and Lazy Evaluation* [Online] Available: http://www.cs.cornell.edu/courses/cs3110/2011sp/lectures/lec24-streams/streams.htm

D, Berkholz (2013, March 25). *Programming languages ranked by expressiveness* [Online]. Available: http://redmonk.com/dberkholz/2013/03/25/programming-languages-ranked-by-expressiveness/

D. Esposito (2010, December 2). *Functional vs. Object-Oriented Javascript Development* [Online] Available: http://msdn.microsoft.com/en-us/magazine/gg476048.aspx

D. Tao (2013) *Lazy.js* [Online] Available: http://danieltao.com/lazy.js/

D. Zindros (2011) *stream.js* [Online] Available: http://streamjs.org/

Ecma International (2011, June). *ECMAScript Language Specification* [Online] Available: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

E. Shepherd (2013, May 28). *New in JavaScript 1.7* [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/New_in_JavaScript/1.7

M. Bilmann. (2011, October 3). *Is Haskell the Cure?* [Online]. Available: http://mathias-biilmann.net/posts/2011/10/03/is-haskell-the-cure

M. Might (2013). *Church encodings, the U combinator and the Y combinator in JavaScript* [Online] Available: http://matt.might.net/articles/js-church/

Microsoft (2013) *Internet Explore Dev Center: Variable Scope (JavaScript)* [Online] Available: http://msdn.microsoft.com/en-us/library/ie/bzt2dkta%28v=vs.94%29.aspx

Mozilla Development Network (2013). *A re-introduction to JavaScript (JS Tutorial)* [Online] Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript

Mozilla Development Network (2013). *Closures* [Online] Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Closures

N. Smith (2012, September 14). *Functional Programming in Javascript* [Online] Available: http://dailyjs.com/2012/09/14/functional-programming/

O. Steele (2007). *Functional Javascript* [Online] Available: http://osteele.com/sources/javascript/functional/

R. Thomas (2000). *A Gentle Introduction to Haskell* [Online] Available: http://www.haskell.org/tutorial/functions.html

T, Ujihisa (2008, January 24). *From Javascript To Haskell* [Online]. Available: http://www.slideshare.net/ujihisa/from-javascript-to-haskell

Wikipedia Contributors. *Closure (Computer Science)* [Online] Available: http://en.wikipedia.org/wiki/Closure_%28computer_science%29

Wikipedia Contributors. *Higher-order function* [Online] Available: http://en.wikipedia.org/wiki/Higher-order_function

Wikipedia Contributors (2013, October 28). *List Comprehension* [Online]. Available: http://en.wikipedia.org/wiki/List_comprehension