Executive Summary

In this paper, we explain the history of and motivation behind the Javascript programming language. We then use this background as a framework for explaining the use of prototypes and inheritance in Javascript. In the Javascript Overview section, we go over when, where, why, and by who Javascript was developed. We continue into a discussion of some derivative and related languages as well as its formalization into ECMAScript. After that, we go over the type system used by Javascript including a formal analysis with inductive definitions. Next, we talk about which programming language paradigms that it falls into. We also discuss Javascript’s use of first-class functions and how it handles operator precedence. We end this section with the major revisions and additions that the language has had since its inception. In the Prototypes and Inheritance in Javascript section, we begin by defining the concept of a prototype and explaining the implications of their use. Following that is a simple example of how to implement an object using a prototype. We then compare and contrast prototype-based programming with the class-based programming of the Java language. We give examples of the differences in implementation of an object’s constructor, attributes, and functions. We then present several concrete examples of the similarities and differences in terms of inheritance and subtyping in both Javascript and Featherweight Java. This is followed by a formal analysis of the machinations behind inheritance and subtyping in JavaScript.
1 Background

Javascript was spawned in 1995 as a feature of the Netscape browser[1]. It was originally called LiveScript, but its name was soon changed to JavaScript as a strategic marketing move with Sun. "Javascript" is a misnomer because the language is only very loosely related to Java. JavaScript was originally developed to manipulate Java applets in the browser, but from the beginning, it was mostly used to bring interactivity to browser content.

A year after the launch of JavaScript, Microsoft launched its own port for the Internet Explorer 3 browser called Jscript. Jscript was technologically a version behind the current JavaScript and did not support the most popular use of JavaScript at the time, image swapping. As a response to Jscript, Netscape and Sun standardized the language with European Computer Manufacturers Association (ECMA). The standard, called ECMAScript, was adopted by ISO in 1998.

ECMAScript is meant to run in a host environment where a number of host objects are provided. ECMAScript was originally designed as a Web scripting language, but it can provide scripting capabilities in a variety of host environments. For example, it is implemented as ActionScript to manipulate Flash animations, and it is also implemented in Adobe Acrobat to provide interactivity with PDF documents.

2 Language Overview

ECMAScript is an object-based language. An object is an unordered collection of properties. A program is a cluster of communication objects. Each object property has zero or more attributes. Properties are containers that hold other objects, primitive values, or methods.

ECMAScript is a multi-paradigm programming language. Like Java, everything is an object, including functions. The usefulness of this approach will be explained later in this paper.

The primitive types of ECMAScript are Undefined, Null, Boolean, Number and String. The only other built-in type is Object, and all objects are of this type. ECMAScript also contains a set of built-in objects. These are the Global object, the Object object, the Function object, the Array object, the String object, the Boolean object, the Number object, the Math object, the Date object, the RegEx object and the Error objects Error, EvalEr-
ror, RangeError, ReferenceError, SyntaxError, TypeError, and URIError. ECMAScript also defines a set of built-in operators, such as unary, multiplicative, additive, bitwise shift, relational, equality, binary bitwise, binary logical, assignment, and comma operators.

ECMAScript syntax was meant to emulate that of Java/C++, but is more relaxed. From a programmers perspective, one big difference is the ability to dynamically type. The only type identifier is var and it’s use it optional. The language is also very liberal about use of whitespace and terminating semicolons. The semicolons at the end of statements is completely optional, but if no semicolon is used, a line break must be entered instead.

Objects and Prototypes ECMAScript does not contain classes. As an object oriented language, this is a notable difference from other object oriented languages such as C++ and Java. It instead supports constructors which create objects by executing code that allocates storage for the objects and initializes all or part of them by assigning initial values to their properties. Each constructor has a Prototype property. Prototypes are used for sharing properties, and for inheritance. To create an object, a new expression must be used. This will be discussed in more detail in the section Prototypes and Inheritance in Javascript.

3 Prototypes in Javascript

Javascript is a prototypes-based language, meaning it uses prototypes to define objects. This is a departure from the more familiar class-based languages, such as Featherweight Java, which use classes as a basis for created objects. In the following section we will illustrate the similarities and differences between prototypes and classes, using code examples to increase clarity.

3.1 Objects As Associative Arrays

In order to understand the use of prototypes in Javascript, it is helpful to first understand how objects are stored by the Javascript interpreter. In Javascript, objects are stored simply as hashtables, otherwise know as associative arrays. Essentially, these arrays store their data as key/value pairs. The code below will create a new associative array with three keys and three values. It then accesses the value for the key name. The keyword var is used
to declare the array, but this is optional. The output for this program is the string Roy.

```javascript
var myArray = {name: "Roy", age: 23, city: "Amherst"};
print(myArray["name"]);
```

Alternatively, we could use dot notation to access the same value. The syntax, shown below, is now starting to look more object-oriented. The output for this program is still Roy.

```javascript
var myArray = {name: "Roy", age: 23, city: "Amherst"};
print(myArray.name);
```

It is also perfectly valid to store functions inside an associative array. Now what we are doing is combing values and functions that can operate on those values inside a single structure. In other words, we are creating an object. This is illustrated in the example below. The output for the code is now Ed.

```javascript
var myArray = {name: "Roy", age: 23, city: "Amherst", setName: function (n) {this.name = n;});
myArray.setName("Ed");
print(myArray.name);
```

### 3.2 Classes Vs. Prototypes

The above examples are fine for creating single instances of objects, but they are not adequate for creating a template from which to make multiple instances of an object. In Featherweight Java, this would be accomplished by writing a class, such as the one below. (Note that it assumes a String class and an int class are defined.)

```java
class Person extends Object {
    String name;
    int age;
    String city;
    Person (String n, int a, String c) {
        super();
        this.name = n;
        this.age = a;
    }
}
```
In Javascript, an equivalent Person prototype can be written. This prototype is essentially a normal function that will be used like a constructor to create new Person objects. The example is shown below. In it, two separate Person object instance are created and their respective name values are accessed. The keyword new is used to instantiate an object based on a prototype defined by a normal function containing fields and other functions. The output will be Roy followed by Ed.

```javascript
function Person(n, a, c) {
    this.name = n;
    this.age = a;
    this.city = c;

    this.getName = function () {return this.name;}
}

var myPerson1 = new Person("Roy", 23, "Amherst");
var myPerson2 = new Person("Ed", 18, "Sunderland");

print(myPerson1.getName());
print(myPerson2.getName());
```

### 3.3 Inheritance

Inheritance is an important feature of any object oriented language. If one wanted to extend the functionality of the Person class into a more specific Student class, the Featherweight Java method for doing so would be:

```java
class Student extends Person {
    String school;
    Student (String n, int a, String c, String s) {
        super(n, a, c);
        this.school = s;
```
Now Student objects can be instantiated that will contain all of the properties of Person and Student. Javascript also supports inheritance, but it is based on an object’s built-in prototype property. In Javascript, every object has a field called prototype that points to the prototype on which the current object is based. When a field or function of an object is access during code execution, the object checks all of its regular properties for the field or function name and if it is not found, it checks its prototype for that field or function. If it does not exist there, the prototype’s prototype is checked, and so on down the line until the Object prototype is reached, which is the default basis for all prototypes. In that way, inheritance can be implemented. An example of the use of the prototype property is shown below. It follows from the Person prototype example above. The output will be Roy, Bob, and Umass.

```javascript
function Student(n, a, c, s) {
  this.prototype = new Person();
  this.base = Person;
  this.base(n, a, c);
  this.school = s;
}

var myPerson = new Person("Roy", 23, "Amherst");
var myStudent = new Student("Bob", 20, "Northampton", "UMass");

print(myPerson.getName());
print(myStudent.getName());
print(myStudent.school);
```

In the code above, the Student prototype’s prototype property is set to the Person prototype. Student now inherits all of the fields and functions from Person. Secondly, the Person constructor is set as a normal function called base. This constructor is then called in order to initialize the fields inherited from Person. Finally, the school field of the Student prototype is initialized. Much like class-based languages such as Featherweight Java, a deep hierarchy of objects can be created. This is done by repeatedly extending the prototype chain until the desired functionality is met. Inheritance
with Dynamic Objects

So far, we have only seen how prototypes are similar to classes. Now we will show how prototype-based languages differ from class-based languages. To do this, we will first go back to associative arrays, on which Javascript objects are built. In the example below, we create an associative array/object as before, but then after it is created we add an additional key/value pair to it. The output for this example will be 6.1.

```javascript
var myArray = {name: "Roy", age: 23, city: "Amherst"};
myArray.height = 6.1;
print(myArray.height);
```

Because Javascript objects are based on associative arrays, we can similarly change and augment objects after they are created. In this way, Javascript is said to support dynamic objects, which class-based languages do not. One use for this feature is the ability to add a new property to an object on the fly without having to change the prototype on which the object is based. This can be seen in the following example, which follows the Student prototype example above.

```javascript
var myStudent = new Student("Bob", 20, "Northampton", "UMass");
myStudent.specialDiet = "Peanut allergy";
print(myStudent.specialDiet);
```

This use of dynamic objects could be helpful in the event that a single object must be extended on the fly. But, a much more interesting use would be to extend an entire set of objects on the fly. For example, the code below will add an additional property to all Person objects, even ones that already have been instantiated. The following code will output 2 followed by 2.

```javascript
var myPerson1 = new Person("Roy", 23, "Amherst");
var myPerson2 = new Person("Ed", 18, "Sunderland");
Person.prototype.numEyes = 2;
print(myPerson1.numEyes);
print(myPerson2.numEyes);
```

Another interesting use of this feature is adding features to the Object class which propagate down to all objects created. One example would be to add an enumerate function that lists all of the properties of its associated object. If this function was run on an instance of Person, it would return name,
age, city, getName, and enumerate. If called on a Student instance, it would
return all of those as well as school, base, etc. The code for this example is
shown below. Using the power of prototypes and dynamic inheritance, other
functions could be added to Object that would extend the functionality of
the language myriad ways.

Object.prototype.enumerate = function () {
    for (id in this)
        print (id);
}

myPerson.enumerate();
myStudent.enumerate();

3.4

4 Formal Semantics

We will formally analyze Javascript prototypes with a stripped down ver-
sion, called Feathermini Javascript, FMJ for short. FMJ is designed to ana-
lyze inheritance in Javascript. The only types it has are Objects, Functions,
and Numbers. And the only statements it has are constructor definitions,
prototype binding, and object property selections. The descriptions we use
for FMJ borrow heavily from Featherweight Java [2].

Numbers
  \text{Num} ::= \text{NUM}(n) \quad n \text{ is an unsigned integer}

Variables
  \text{Var} ::= \text{VAR}(s) \quad s \text{ is a valid identifier string}

Constructor
  \text{Constructor} ::= \text{functionVar}\{\text{FBody}\}

Fields Body
  \text{FBody} ::= \text{this.Var} = \text{Num}; \text{FBody} \mid \text{this.Var} = \text{Num};

Prototyping
  \text{Proto} ::= \text{Var}().\text{prototype} = \text{new Var}()

Object Instantiation
  \text{ObjInst} ::= \text{new Var}()

Property selection
  \text{PropSel} ::= \text{ObjInst.Var}

ConstructorSequence
  \text{ConstSeq} ::= \text{Constructor} | \text{ConstructorConstSeq}

PrototypeSequence
  \text{ProtoSeq} ::= \text{Proto} | \text{Proto;ProtoSeq}

Program
  \text{Prog} ::= \text{ConstSeq} \text{ProtoSeq PropSel}
The following is an example of a program in FMJ. Circle is the prototype of EmbeddedCircle. This program evaluates to 1, showing that when an object was constructed, it was possible to retrieve property values that were bound by the constructor’s prototype.

```javascript
function Circle() {
    this.radius = 1;
}
function EmbeddedCircle() {
    this.x = 2;
    this.y = 3;
}
EmbeddedCircle.prototype = new Circle();
new EmbeddedCircle().radius;
```

Now that we have a grammar, we will work out the static and dynamic semantics of FMJ.

A constructor table $T$ is a finite function assigning constructor functions to constructor names. A prototype table $P$ is a finite function assigning constructor name, $C_1$, to constructor name $C_2$ such that the $C_1.prototype = C_2$. A program is a tuple $(T, P, e)$ consisting of:

- A constructor table $T$.
- A prototype table $P$.
- An expression $e$.

We assume a fixed constructor table and prototype table. When a constructor does not have a prototype, we assume its prototype to be a constructor `Undefined`, which does not appear in the table. The auxiliary functions that look up properties will return the empty sequence of properties for the special case of Undefined.

For the statics and dynamic semantics rules, we require an auxiliary definition for properties of a constructor. Note that `this.p = x` is shorthand for `this.p_1 = x_1; this.p_2 = x_2; ...` and `p = x` is shorthand for `p_1 = x_1; p_2 = x_2; ...`. Finally, `x : number` is shorthand for `x_1 : number, x_2 : number, ...`

\[
T(c) = \text{function } c() \{ \text{this.q = y} \} \quad P(c) = \text{d properties}(d) = p = x \quad y : \text{number} \quad x : \text{number} \quad \text{properties}(c) = p = x, q = y
\]

(1)
We first will list and then define the judgement forms.

\[ c \trianglelefteq c' \quad \text{sub-prototyping, i.e. } c' \text{ is in the prototype chain of } c \]

\[ \Gamma \vdash e : \tau \quad \text{expression typing} \]

\[ c \ ok \quad \text{well-formed constructor} \]

\[ p \ ok \quad \text{well-formed prototype} \]

\[ T \ ok \quad \text{well-formed constructor table} \]

\[ P \ ok \quad \text{well-formed protototype table} \]

\[ \text{properties}(c) = p = x \quad \text{property lookup} \]

The definition of a well-formed constructor:

\[
\begin{array}{c}
\text{function } c() \{ \text{this.q = y} \} \quad c\#T \\
\hline
\end{array}
\]

(2)

The definition of a well-formed prototype:

\[
\begin{array}{c}
T(c)ok \quad T(d)ok \quad c.\text{prototype} = d \\
\hline
P(c)ok
\end{array}
\]

(3)

The definition of a well-formed class table:

\[
\begin{array}{c}
\forall c \quad T(c)ok \\
\hline
Tok
\end{array}
\]

(4)

We describe the definition of sub-prototyping, which is very analogous to sub-classing of FWJ.

\[
\begin{array}{c}
P(c) = d \quad P(c)ok \\
\hline
\]

(5)

Reflexivity:

\[
\begin{array}{c}
T(c)ok \\
\hline
\]

(6)

Transitivity:

\[
\begin{array}{c}
c \trianglelefteq c' \quad c' \trianglelefteq c'' \\
\hline
c \trianglelefteq c''
\end{array}
\]

(7)
There is only one value judgement for FMJ. Each FMJ program returns a single value of type number.

\[
x : \text{number} \\
\x \text{val}
\]

(8)

We now describe object creation and property selection. As Objects are a single type that works as a hashtable, we have chosen to not require an object to be typed, but only require that its constructor is well-formed. We have limited all fields to be the primitive type number.

\[
\frac{\text{c ok properties}(c) = p_1 = x_1, p_2 = x_2, \ldots}{\Gamma \vdash \text{new c}.p_i : \text{number}}
\]

(9)

We now show the dynamic semantics of property selection.

\[
\frac{\text{c ok properties}(c) = p_1 = x_1, p_2 = x_2, \ldots, x_i}{\text{new c}.p_i \mapsto x_i}
\]

(10)

As you can see here, once the properties judgement was properly defined with the prototype chain, it was quite easy to make the typing and evaluation rules for field selection in FMJ.

References
