1 Static Semantics

We will formally analyze Javascript with a stripped down version, called featherminiJavascript, FMJ for short. FMJ is designed to analyze 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.

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

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

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

Fields Body
\[\text{FBody} ::= \text{this.Var} = \text{Num}; \text{FBody} | \text{this.Var} = \text{Num};\]

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
\[\text{ProtoSeq} ::= \text{Proto} | \text{Proto};\text{ProtoSeq}\]

Program
\[\text{Prog} ::= \text{ConstSeq ProtoSeq PropSel}\]

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.
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, ...

\[
T(c) = \text{function } c() \{ \text{this.$q$ = $y$} \} \quad P(c) = d \quad \text{properties(d)} = p = x \quad y : \text{number} \quad x : \text{number} \\
\text{properties(c)} = p = x, q = y
\]  
(1)

We first will list and then define the judgement forms.
c \sqsubseteq 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 \text{ ok} \quad \text{well-formed constructor}
p \text{ ok} \quad \text{well-formed prototype}
T \text{ ok} \quad \text{well-formed constructor table}
P \text{ ok} \quad \text{well-formed prototype table}
properties(c) = p = x \quad \text{property lookup}

The definition of a well-formed constructor:
\[
\begin{array}{c}
\text{function } c() \{ \text{this.q = y} \} \\
c\#T \\
c
\end{array}
\] (2)

The definition of a well-formed prototype:
\[
\begin{array}{c}
T(c) \text{ ok} \quad T(d) \text{ ok} \quad c.\text{prototype} = d \\
P(c) \text{ ok} \\
\end{array}
\] (3)

The definition of a well-formed class table:
\[
\begin{array}{c}
\forall c \; T(c) \text{ ok} \\
Tok
\end{array}
\] (4)

We describe the definition of sub-prototyping, which is very analogous to
sub-classing of FWJ.
\[
\begin{array}{c}
P(c) = d \quad P(c) \text{ ok} \\
c \sqsubseteq d
\end{array}
\] (5)

Reflexivity:
\[
\begin{array}{c}
T(c) \text{ ok} \\
c \sqsubseteq c
\end{array}
\] (6)

Transitivity:
\[
\begin{array}{c}
c \sqsubseteq c' \quad c' \sqsubseteq c'' \\
c \sqsubseteq c''
\end{array}
\] (7)
There is only one value judgement for FMJ. Each FMJ program returns a single value of type number.

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

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.

\[
\begin{align*}
\text{c ok} & \quad \text{properties}(c) = p_1 = x_1, p_2 = x_2, \ldots \\
\Gamma & \vdash \text{new } c().p_i : \text{number}
\end{align*}
\]  

We now show the dynamic semantics of property selection.

\[
\begin{align*}
\text{c ok} & \quad \text{properties}(c) = p_1 = x_1, p_2 = x_2, \ldots x_i \\
\text{new } c().p_i & \mapsto x_i
\end{align*}
\]  

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.