An executable formal specification of ECMAScript

Cristian Dittamo Vincenzo Gervasi Antonio Cisternino Egon Börger

February 8, 2011

1 Introduction

Web applications (i.e., applications whose interface is presented to the user via a web browser, whose state is split between a server and a client, and where the only interaction between server and client is through the HTTP protocol) are become more and more widespread, and integrated in most users' everyday work habits. The glue linking the desperate technologies involved, from dynamic HTML to XML RPC, is the Javascript language. Yet, no formal definition of its semantics exists, which in turn makes it impossible to formally prove correctness, liveness and security properties of Web applications. As a first step towards improving this situation, we provide a formal semantics for ECMAScript [2], by means of an Abstract State Machines (ASMs) specification [1]. We follow the path established by other specification efforts for similar languages (e.g. Java and C#), but in addition we establish a formal trace between parts of our specification and the ECMA standard, thus facilitating the proof of correctness of the specification. More specifically, we define the dynamic semantics of ECMAScript by providing (in terms of ASMs) two distinct interpreters, one (ESInterpret) traversing the ECMAScript abstract syntax tree (AST), the other (AOInterpret) is step-wise equivalent to the ECMA standard, as shown in Figure 1.

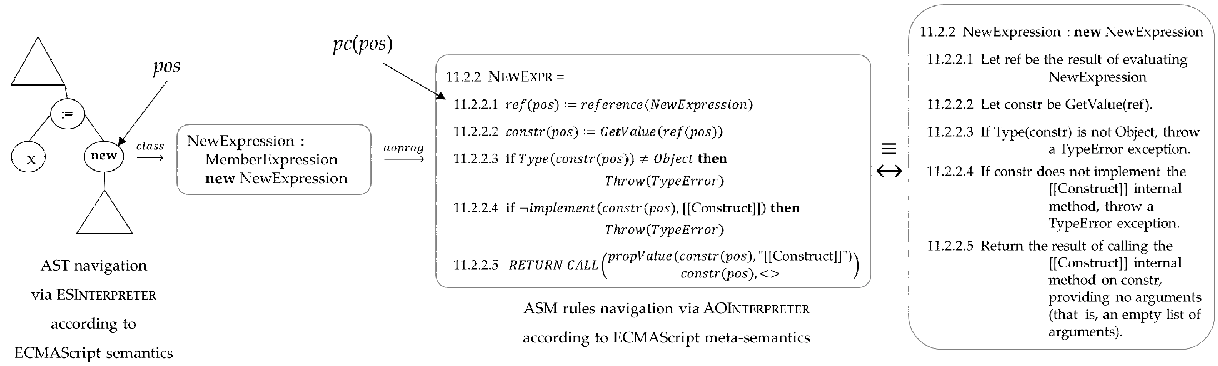

Figure 1: Overview of our specification.

We use an algebra to represent (i) the state of the ECMAScript program, (ii) the state of the Interpreter, (iii) the state of the host environment (typically, the browser). In our modeling we do not over-specify; all the points left open in the standard are described through non-deterministic choose statements. Moreover, we make our specification executable by implementing it in CoreASM, which provides a concrete implementation of the ASM language.

2 Notation

We assume the necessary syntactic information (read: the Abstract Syntax Tree (AST) of the given ECMAScript program) to be available, namely described by background functions. Using ASM we detail a visit of the AST, whose effect is to simulate the execution of the program. We specify the Interpreter as a collection of rules which traverse an AST while evaluating values and locations. The notation we will use is inspired by the one first introduced in [3]. We state the following assumptions:

1. nodes in the tree are in the domain of the following (mostly partial) functions:

   - three static functions implement tree navigation:
     - *first* : *Node* $\rightarrow$ *Node* returns the first child of a given AST node.
     - *next* : *Node* $\rightarrow$ *Node* returns the next child, if present, of a given AST node.
     - *parent* : *Node* $\rightarrow$ *Node* returns the parent node of a given AST node.

     by using these functions, the Interpreter can access all the children nodes of a given node, or go back to its parent;
   - *class* : *Node* $\rightarrow$ *Class* returns the syntactical class of a node (i.e., the name of the corresponding grammar non-terminal class); for example Literal.
   - a special variable *pos* : $\rightarrow$ *Node* holds at all times the current position in the AST, that is, the root node of the subtree which is being evaluated at a given time instant;
   - $[\![\cdot]\!]$: *Node* $\rightarrow$ *Reference* $\times$ *Completion* $\times$ *Value* holds the result of the interpretation of a node, given by a tuple formed by a *reference* (that is, the l-value of an expression, when it is defined), a *completion* status (indicating whether an expression or statement was evaluated normally, or if some non-sequential transfer of control is in order), and a *value* (that is, the r-value of an expression)[1]. We access elements and establish properties of such tuples through the following derived functions:
     - *ref* : *Node* $\rightarrow$ *Reference* returns the location (l-value) associated to the given node, i.e. $ref(n) \equiv [\![n]\!] \downarrow 1$.
     - *compl* : *Node* $\rightarrow$ *Completion* returns the completion status of the evaluation of a node, i.e. $compl(n) \equiv [\![n]\!] \downarrow 2$.
     - *value* : *Node* $\rightarrow$ Value returns the value (r-value) associated to the given node, i.e. $value(n) \equiv [\![n]\!] \downarrow 3$.
     - *evaluated* : *Node* $\rightarrow$ *Boolean* indicates if a node has been fully evaluated. We have,

       $$evaluated(n) \equiv [\![n]\!] \neq undef$$

2. the behavior of ECMAScript constructs is operationally described in [2] using so-called abstract operations (AO), coming as numbered lists of steps, upon which we assume the standard ordering and nesting structure. Therefore, AOs are themselves organized as a tree, called Abstract Operation Tree (AOT). Let AOT be the domain of trees of ASMs rules or instructions, we define the following function:

   $$aoprog : Class \rightarrow AOT$$

   that given the node's syntactical class returns a tree of ASM rules indexed by an hierarchical enumeration conformant to the one used in the ECMA standard.

   Let *InstrRef* be the domain of instructions references. Let $i \in InstrRef$ and $t \in AOT$, i refers an instance of an instruction in a node of t. We define the following functions:

   - *pc* : *Node* $\times$ *Integer* $\rightarrow$ *InstrRef* returns the reference of the current instruction in the AOT (i.e. ECMA AO) of a given AST node at a given level of calls stack. This level is held in a special variable $sp(pos)$, such that we can:
     - define the scope of each variable used inside an AOT (i.e. ECMA AO) for each AST node.
     - avoid to use a global stack for ECMA AOs call.

     The use of $sp(pos)$ will be more clearer in Section 3.3.
   - *root* : $AOT \rightarrow$ *InstrRef* returns the root instruction reference of a given AOT.
   - three static functions implement AOT navigation:
     - *aofirst* : *InstrRef* $\rightarrow$ *InstrRef* returns the first child of a given AOT node.
     - *aonext* : *InstrRef* $\rightarrow$ *InstrRef* returns the next child of a given AOT node.

[1] The structure of the tuple is intended to be mnemonic, with the l-value in the leftmost and the r-value in the rightmost position in the tuple.

– *aoparent* : *InstrRef* → *InstrRef* returns the parent node of a given AOT node.

by using these functions, the AOInterpret can access all the children nodes of a given node, or go back to its parent;

- *ret* : *Node* × *Integer* → *Value* returns the value obtained by an AO execution at a given: AST node and level of calls stack.

3 Core Specification

When control is transferred to ECMAScript code the Interpreter is called upon, passing it the position of the AST root node (corresponding to the Program production of [2](§A.5)). The current position *pos* is thus initialized to the root node, the $[\![\cdot]\!]$ function of all the nodes in the subtree is set to *undef*, and the following rule is invoked.

3.1 The Interpreter

We define the semantics of ECMAScript by means of two distinct interpreters, one, called ESInterpret, traversing an ECMAScript AST, the other, called AOInterpret, executing AOs as specified by the ECMA standard for each ECMAScript construct. At each moment at most one Interpreter instance is called. The main rule of our Interpreter is thus

Interpreter rules

```
Interpreter ≡
  if pos ≠ undef then
    if aomode(pos) = idle then
      ESInterpret
    else
      AOInterpret
```

The Interpreter starts checking whether a given AST is already traversed (i.e. *pos* is *undef*) or not. In the former case the Interpreter state will be not changed. In latter case, if the current AST node must be evaluated (i.e. control state *aomode* is not *idle*) then the AOInterpret is executed. Otherwise ESInterpret is performed.

3.2 ESInterpret interpreter

This interpreter determines whether the root node of the current AST subtree is already evaluated or not. In the former case, the control state *aomode* is changed to *initialize*, such that in the next step the AOInterpret sub-interpreter will execute. In the latter case, all nodes in the current AST subtree are already evaluated thus the *pos* must be changed back to current node's parent (recall Section 2). This formalized as follows:

ESInterpret main rule

```
ESInterpret ≡
  if ¬evaluated(pos) then
    aomode(pos) := initialize
  else
    pos := parent(pos)
```

3.3 AOInterpret interpreter

AOInterpret determines which is the tree of ASM rules (i.e. AOT) corresponding to the current AST node class. Therefore, if a new AST subtree must be evaluated (i.e. *aomode* = *initialize*), AOInterpret updates *pos* to that subtree first child, and *aomode* to *running*, such that in the next step the node evaluation can be performed. The main rule of our AOInterpret is thus

AOINTERPRET main rule

```
AOInterpret ≡
  let il = aoprog(class(pos)) in
    if aomode(pos) = initialize then
      pc(pos, sp(pos)) := root(aoprog(class(pos)))
      aomode(pos) := running
  if aomode(pos) = running then
    let R = il(pc(pos, sp(pos))) in R(...)
```

where each R updates $pc(pos,sp(pos))$ and $aomode(pos)$ as needed.

```
If(cond) ≡
  if cond then aofirst(pc(pos, sp(pos)))
  else aonext(pc(pos, sp(pos)))

Else ≡ aonext(pc(pos, sp(pos)))
```

```
For(list, element) ≡
  if ¬list.Empty then
    element := list.Dequeue()
    aofirst(pc(pos, sp(pos)))
  else
    pc(pos, sp(pos)) := aoparent(pc(pos, sp(pos)))

EndFor ≡
  pc(pos, sp(pos)) := aonext(aoparent(pc(pos, sp(pos))))
```

these allow to traverse an ASM rules tree using the same syntax of control flow instructions in AOs. In the For macro we assume to have a queue of ASM rules as input and the next rule in that queue as output. The For upper bound is given by the input queue length.

In ECMA standard there are calls to properties and methods. Properties will be discussed in Section 3.5. We define only one macro, called Call, to account all AOs' calls. However, the ECMA standard states that host objects may implement AOs in any manner unless specified otherwise. For this reason, we define the oracle function

$$isHostObj : \mathsf{Object} \to \mathsf{Boolean}$$

that determines whether a given object is an host object or not. In the former case, the ExecNativeCode macro executes in an unspecified way the given AO. In the latter case, the current state is saved in a stack before a given AO execution and restored immediately after.

```
Call(prog, obj, args) ≡
  if isHostObj(obj) then
    Return(ExecNativeCode(obj, prog)(args))
  else
    let sp(pos) = sp(pos) + 1 in
      pc(pos, sp(pos)) := aofirst(prog)
      arg(pos, sp(pos)) := args
      aonext(pc(pos, sp(pos)))
```

```
Return(v) ≡
  if sp(pos) > 0 then
    sp(pos) := sp(pos) − 1
    ret(pos, sp(pos)) := v
  else
    [[pos]] := (undef, normal, v)
    aomode(pos) := idle
```

```
Throw(e) ≡
  [[pos]] := (undef, throw, undef)
  aomode(pos) := idle
```

3.4 Types and values

ECMAScript defines nine types, three of which are used only as types for intermediate values of expressions, and cannot be stored in a property (see Section 3.5). The types who are visible in the language itself are briefly discussed in the following:

- the Undefined type only contains a single value, that we will denote with undefined. Reading a property to which no value has been assigned returns undefined. Notice that undefined should not be confused with *undef*, which is the denotation for undefined values in the ASM state.
- the Null type only contains a single value, that we will denote with null.
- the Boolean type contains two values, true and false, with the obvious meaning. Notice that these should not be confused with *true*and *false*, which are the boolean values of the ASM signature.

- the String type contains strings, which we will consider as sequences of characters. We will abstract here from the details of the encoding, assuming that the ASM operators and functions operating on characters and character sequences will have the intuitively correct behaviour regardless of the particular in-memory representation of Strings.
- the Number type contains floating-point numbers, plus the three special values NaN, +Infinity and -Infinity, whose semantics is according to [**?**]. Again, for the purposes of this specification we will abstract from the details of the in-memory representation of these values[2], assuming instead that the operations will be well-behaved according to [**?**]. For expository purposes, we will use real numbers as an abstraction of floating-point numbers.
- the Object type contains collections of properties; values of this type are better described in Section 3.5

in addition, the following types are defined for expository purposes (we will use different typograhic style for them to denote their special nature):

- the Reference type (*ECMAReference*), is used to hold a reference to a property of a particular object; it will be discussed more fully in Section 3.7, but for now it will suffice to say that its values are triples $\langle b, p, s\rangle$ where b is either an Object, a Boolean, a Number or an Environment Record (*EnvRec*) value, p is a String value (holding the name of the property), and s is the Boolean valued *strict reference flag*.
- the List type (*List*) contains ordered sequences (lists) of values. Values $\in$ *List* are used to hold the results of evaluating arguments lists in function calls, in **new** expression, and in other algorithms where a simple list of values is needed.
- the Completion type (*Completion*) contains references to program positions, used to alter the control flow in an unstructured manner for those statements (`break`, `continue`, `return` and `throw`) which require a non-local transfer of control. Values $\in$ *Completion* are triples of the form $\langle type, value, target\rangle$ where *type* is one of normal, break, continue, return, or throw, *value* is any ECMAScript language value or empty, and *target* is any ECMAScript identifier or empty.
- the Property Descriptor type (*PropDescr*) is used to explain the manipulation and reification of named property attributes. Values $\in$ *PropDescr* are records composed of named fields where each field's name is an attribute name and its value is a corresponding attribute value as specified in Section 3.5. Property Descriptor values may be further classified:
  - *Data property* descriptor (*DataDescr*), that includes any fields named either [[Value]] or [[Writable]].
  - *Accessor property* descriptor (*AccessDescr*), that includes any fields named either [[Get]] or [[Set]].
  - *Generic property* descriptor, that is neither a data not an accessor property descriptor.

  A Property Descriptor value may not be both a data property and an accessor property descriptor; however, it may be neither. Any property descriptor may have fields named [[Enumerable]] and [[Configurable]].

  A Data descriptor property can be assigned to (and created if non-existing) or written, according to their attributes. Formally, this is modeled by the following rules.

[2] We also abstracts from the existance of two distinct values for zero, namely +0 and /0.

Descriptor types

Data property

$dataDescAttrib = \{[[Value]], [[Writable]]\}$
$dataDescValue : PropDescr \times \mathsf{String} \rightarrow Boolean$
$dataDescAttrs : PropDescr \times \mathsf{String} \rightarrow \{\mathsf{Undefined},\mathsf{Null},\mathsf{Boolean},\mathsf{String},\mathsf{Number},\mathsf{Object}\ \}$

$hasValue : PropDescr \rightarrow Boolean$
$hasValue(sr) = dataDescValue(sr, "[[Value]]")$

$hasWritable : PropDescr \rightarrow Boolean$
$hasWritable(sr) = dataDescValue(sr, "[[Writable]]")$

$isWritable : PropDescr \rightarrow Boolean$
$isWritable(sr) = dataDescAttrs(sr, "[[Writable]]")$

An Accessor descriptor properties can be called (for executable ones) according to their attributes. Formally, this is modeled by the following rules.

Descriptor types

Accessor property

$accessDescAttrib = \{[[Get]], [[Set]]\}$
$accessDescValue : PropDescr \times \mathsf{String} \rightarrow Boolean$

$hasSet : PropDescr \rightarrow Boolean$
$hasSet(sr) = accessDescValue(sr, "[[Set]]")$

$hasGet : PropDescr \rightarrow Boolean$
$hasGet(sr) = accessDescValue(sr, "[[Get]]")$

A Generic descriptor property can be enumerated or deleted according to their attributes. Formally, this is modeled by the following rules.

Descriptor types

Generic Descriptor

$genDescrAttrib = \{[[Enumerable]], [[Configurable]]\}$
$genDescAttrs : PropDescr \times \mathsf{String} \rightarrow Boolean$

$isEnumerable : PropDescr \rightarrow Boolean$
$isEnumerable(sr) = genDescAttrs(sr, "[[Enumerable]]")$

$isConfigurable : PropDescr \rightarrow Boolean$
$isConfigurable(sr) = genDescAttrs(sr, "[[Configurable]]")$

- the Property Identifier type (*PropId*) is used to associate a property name with a Property Descriptor. Values $\in$ *PropId* are pairs of the form $\langle name, descriptor \rangle$, where $name \in \mathsf{String}$ and $descriptor \in PropDescr$. A complete description will be done in Section 3.6.
- the Environment Record type (*EnvRec*) is used to record the identifier bindings created within the scope of its associate Lexical Environment. There are two kind of Environment Record values:
  - Declarative Environment record (*DeclER*) is used to define the effect of ECMAScript language syntactic elements (e.g. FunctionDeclaration, VariableDeclaration and Catch clauses) that directly associate identifier bindings with ECMAScript Language values.
  - Object Environment record (*ObjER*) is used to define the effect of ECMAScript elements (e.g. *Program* and WithStatement) that associate identifier bindings with the properties of some objects.

- the Lexical Environment type (*LexEnv*) is used to define the association of *Identifiers* to specific variables and functions based upon the lexical nesting structure of ECMAScript code. Values $\in$ *LexEnv* type are pairs of the form $\langle envRec, refLexEnv \rangle$, where $envRec \in EnvRec$ and $refLexEnv \in LexEnv$ is a reference to an outer Lexical Environment. The latter is used to model the logical nesting of Lexical Environment values. This will discussed more fully in Section 3.13.

3.5 Objects

An *ECMAScript Object* or Object for short, is a set of *Properties*, where each property is a triple $\langle name, value, attributes \rangle$. For our purposes, *name* can be considered a string, *value* is any legal ECMAScript value (see Section 3.4) or an ASM rule (specifying the actions to be taken for internal methods, as per [2](§8.6.2)), whereas *attributes* is a set of values from a given set of predefined features ([2](§8.6.1)) that we will denote with names enclosed in double square brackets. Each property can be:

- A Named Data property, that associates a name with an ECMAScript language value [[Value]] and a set of Boolean attributes, such as [[Writable]], [[Enumerable]] and [[Configurable]].
- A Named Accessor property, that associates a name with one or two accessor functions (i.e. [[Get]] and [[Set]]) and a set of Boolean attributes, such as [[Enumerable]] and [[Configurable]]. The accessor functions are used to store or retrieve an ECMAScript language value that is associated with the property.
- An Internal property, that has no name and is not directly accessible via ECMAScript language operators. Internal properties exist purely for specification purposes.

These properties can be assigned to (and created if non-existing), read from, deleted, called (for executable ones) or enumerated, according to their attributes. Object can be either *native* objects (i.e., created by ECMAScript code) or *host* objects (i.e., created by the environment in which the ECMAScript code is running).

Formally, this is modeled by the following rules.

Object signature

OBJECT
ATTRIBUTE $= \{Value, Writable, Enumerable, Configurable, Get, Set\}$

$propValue$: Object $\times$ String $\rightarrow$ ATTRIBUTE
$propAttrs$: Object $\times$ String $\rightarrow$ {Undefined,Null,Boolean,String,Number,Object }

$isWritable : ECMAReference \rightarrow Boolean$
$isWritable(sr) = propAttrs(sr, "[[Writable]]")$

$isEnumerable : ECMAReference \rightarrow Boolean$
$isEnumerable(sr) = propAttrs(sr, "[[Enumerable]]")$

$isConfigurable : ECMAReference \rightarrow Boolean$
$isConfigurable(sr) = propAttrs(sr, "[[Configurable]]")$

If the value of an attribute is not explicitly specified by the ECMA standard for a named property, the default value is given in Table 1.

3.5.1 Internal Properties

All objects have a few internal properties which, following [2](§8.6.2), we will denote with names enclosed in double square brackets.

Internal methods correspond to behaviour expected of a object. All objects must implement properties named [[Get]], [[GetOwnProperty]], [[GetProperty]], [[Put]], [[CanPut]], [[HasProperty]], [[Delete]] and [[DefaultValue]]; for internal objects (with the exception of arrays, see **??**), the corresponding behaviour is given by the following functions and rules. As stated in the ECMAScript standard, host objects may

Attribute Name	Default Value
[[Value]]	undefined
[[Get]]	undefined
[[Set]]	undefined
[[Writable]]	false
[[Enumerable]]	false
[[Configurable]]	false

Table 1: Default Attribute Values defined in Table 7 of [2](§8.6.1)

implement these internal methods in any manner unless specified otherwise. However, if any specified manipulation of a host objects' internal properties is not supported by an implementation, that manipulation must throw a *TypeError* exception when attempted.

In the following ASM rules, we assume o is a native ECMScript Object, p is a String, $desc$ is $\in$ $PropDescr$, and $throw$ is a Boolean flag.

The [[GetOwnProperty]] property provides the method for retrieving the value of a Property Descriptor of the named (own) property of a given object:

Internal Methods - GetOwnPropertyNoStr

GetOwnPropertyNoStr$(o, p) \equiv$

1. **if** $propValue(o, p) =$ undefined **then** RETURN(undefined)
2. $D(pos, sp(pos)) :=$ **new** $(PropDescr)$
3. $X(pos, sp(pos)) := propValue(o, p)$
4. CALL$(IsDataDescriptor,$ **this**$, D(pos, sp(pos)))$
5. IF$(ret(pos, sp(pos)))$

5.*a.* $propAttrs(D(pos, sp(pos)),$ "[[Value]]"$) := propAttrs(X(pos, sp(pos)),$ "[[Value]]"$)$

5.*b.* $propAttrs(D(pos, sp(pos)),$ "[[Writable]]"$) := propAttrs(X(pos, sp(pos)),$ "[[Writable]]"$)$

6. ELSE

6.*a.* $propAttrs(D(pos, sp(pos)),$ "[[Get]]"$) := propAttrs(X(pos, sp(pos)),$ "[[Get]]"$)$

6.*b.* $propAttrs(D(pos, sp(pos)),$ "[[Set]]"$) := propAttrs(X(pos, sp(pos)),$ "[[Set]]"$)$

7. $propAttrs(D(pos, sp(pos)),$ "[[Enumerable]]"$) := propAttrs(X(pos, sp(pos)),$ "[[Enumerable]]"$)$
8. $propAttrs(D(pos, sp(pos)),$ "[[Configurable]]"$) := propAttrs(X(pos, sp(pos)),$ "[[Configurable]]"$)$
9. RETURN$(D(pos, sp(pos)))$

At row 2, **new** $(PropDescr)$ creates a new empty property descriptor with no fields.

The [[GetProperty]] property provides the method for retrieving the fully populated Property Descriptor of a named property of a given object:

Internal Methods - GetProperty

GetProperty$(o, p) \equiv$

1. CALL$(propValue(o,$ "[[GetOwnProperty]]"$), o, p)$
2. $prop(pos, sp(pos)) := ret(pos, sp(pos))$
3. **if** $prop(pos, sp(pos)) \neq$ undefined **then** RETURN$(prop(pos, sp(pos)))$
4. $proto(pos, sp(pos)) := propValue(o,$ "[[Property]]"$)$
5. **if** $proto(pos, sp(pos)) =$ null **then** RETURN(undefined)
6. CALL$(propValue(proto(pos, sp(pos)),$ "[[GetProperty]]"$), proto(pos, sp(pos)), p)$
7. RETURN$(ret(pos, sp(pos)))$

The [[Get]] property provides the method for retrieving the value of a property of an object, possibly looking up through the prototype chain if needed:

Internal Methods - Get

Get$(o, p) \equiv$
1. CALL($propValue(o,$ "[[GetProperty]]"$), o, p$)
2. $desc(pos, sp(pos)) := ret(pos, sp(pos))$
3. **if** $desc(=$ undefined **then** RETURN(undefined)
4. CALL($IsDataDescriptor$, **this**, $desc(pos, sp(pos))$)
5. **if** $ret(pos, sp(pos))$ **then** RETURN($propAttrs(desc(pos, sp(pos))$, "[[Value]]"$)$)
6. $getter(pos, sp(pos)) := propAttrs(desc(pos, sp(pos))$, "[[Get]]"$)$
7. **if** $getter(pos, sp(pos)) =$ undefined **then** RETURN(undefined)
8. CALL($propValue(getter(pos, sp(pos))$, "[[Call]]"$), getter(pos, sp(pos)), o$)
9. RETURN($ret(pos, sp(pos))$)

Conversely, the [[Put]] property provides the method for setting the value of a property of an object; it should be noted that whether a property is read-only or not is not determined directly by looking at its attributes, but rather by executing the [[CanPut]] method (which by default will honour the ReadOnly attribute, but could be implement differently in host objects). Differently from [[Get]], [[Put]] never accesses properties of its prototype. The behaviour of [[Put]] is thus:

Internal Methods - Put

Put$(o, p, v, throw) \equiv$
1. CALL($propValue(o,$ "[[CanPut]]"$), o, p$)
2. IF($\neg ret(pos, sp(pos))$)
2.*a.* IF($throw$)
2.*a.i.* THROW($TypeError$)
2.*b.* ELSE
2.*b.i.* **return**
3. CALL($propValue(o,$ "[[GetOwnProperty]]"$), o, p$)
4. $ownDesc(pos, sp(pos)) := ret(pos, sp(pos))$
5. CALL($IsDataDescriptor$, **this**, $ownDesc(pos, sp(pos))$)
6. IF($ret(pos, sp(pos))$)
6.*a.* $valueDesc(pos, sp(pos)) :=$ **new** $(PropDescr)$
6.*b.* $propAttrs(valueDesc(pos, sp(pos))$, "[[Value]]"$) := v$
6.*c.* CALL($propValue(o,$ "[[DefineOwnProperty]]"$), o, p, valueDesc(pos, sp(pos)), throw$)
6.*d.* RETURN($ret(pos, sp(pos))$)
7. CALL($propValue(o,$ "[[GetProperty]]"$), o, p$)
8. $desc(pos, sp(pos)) := ret(pos, sp(pos))$
9. CALL($IsAccessorDescriptor$, **this**, $desc(pos, sp(pos))$)
10. IF($ret(pos, sp(pos))$)
10.*a.* $setter(pos, sp(pos)) := propAttrs(desc(pos, sp(pos))$, "[[Set]]"$)$
10.*b.* CALL($propValue(setter(pos, sp(pos))$, "[[Call]]"$), setter(pos, sp(pos)), o, v$)
11. ELSE
11.*a.* CALL($CreatePropDescriptor$, **this**, v, true, true, true)
11.*b.* $newDesc(pos, sp(pos)) := ret(pos, sp(pos))$
11.*c.* CALL($propValue(o,$ "[[DefineOwnProperty]]"$), o, p, newDesc(pos, sp(pos)), throw$)
12. RETURN($ret(pos, sp(pos))$)

where $CreatePropDescriptor$ creates a new Property Descriptor as defined in Section 3.6. At row 3.a, **new** $(PropDescr)$ creates a new empty property descriptor with no fields.

As already mentioned, [[CanPut]] is used to determine if a given property can be assigned a new value or not. Its definition for internal objects is the following:

Internal Methods - CanPut

CanPut$(o, p) \equiv$
1. CALL($propValue(o,$ "[[GetOwnProperty]]"$), o, p$)
2. $desc(pos, sp(pos)) := ret(pos, sp(pos))$
3. IF($desc(pos, sp(pos)) \neq$ undefined)
3.*a.* CALL($IsAccessorDescriptor$, **this**, $desc(pos, sp(pos))$)
3.*b.* IF($ret(pos, sp(pos))$)
3.*b.i.* **if** $propAttrs(desc(pos, sp(pos)),$ "[[Set]]"$) \neq$ undefined) **then** RETURN(false)
3.*c.* ELSE
3.*c.i.* RETURN($propAttrs(desc(pos, sp(pos)),$ "[[Writable]]"$)$)
4. $proto(pos, sp(pos)) := propValue(o,$ "[[Prototype]]"$)$
5. **if** $proto(pos, sp(pos)) =$ null **then** RETURN($propValue(o,$ "[[Extensible]]"$)$)
6. CALL($propValue(proto(pos, sp(pos)),$ "[[GetProperty]]"$), proto(pos, sp(pos)), p$)
7. $inherited(pos, sp(pos)) := ret(pos, sp(pos))$
8. **if** $inherited(pos, sp(pos)) =$ undefined **then** RETURN($propValue(o,$ "[[Extensible]]"$)$)
9. CALL($IsAccessorDescriptor$, **this**, $inherited(pos, sp(pos))$)
10. IF($ret(pos, sp(pos))$)
10.*a.* RETURN($propAttrs(inherited(pos, sp(pos)),$ "[[Set]]"$) =$ undefined)
11. ELSE
11.*a.* IF($\neg propValue(o,$ "[[Extensible]]"$)$)
11.*a.i.* RETURN(false)
11.*b.* ELSE
11.*b.i.* RETURN($propAttrs(inherited(pos, sp(pos)),$ "[[Writable]]"$)$)

The property [[HasProperty]] is used to determine if an object holds a value for a given property.

Internal Methods - HasProperty

HasProperty$(o, p) \equiv$
1. CALL($propValue(o,$ "[[GetProperty]]"$), o, p$)
2. $desc(pos, sp(pos)) := ret(pos, sp(pos))$
3. RETURN($desc(pos, sp(pos)) \neq$ undefined)

The property [[Delete]] holds the behaviour needed to remove a property from an object. Notice that after deletion of a property, [[HasProperty]] can still return true, and [[Get]] can still return a value — for example, because the prototype has the same property.

Internal Methods - Delete

Delete$(o, p, throw) \equiv$
1. CALL($propValue(o,$ "[[GetOwnProperty]]"$), o, p$)
2. $desc(pos, sp(pos)) := ret(pos, sp(pos))$
3. **if** $desc(pos, sp(pos)) =$ undefined) **then** RETURN(true)
4. IF($propAttrs(desc(pos, sp(pos)),$ "[[Configurable]]"$)$)
4.*a.* $propValue(o, p) := undef$
4.*b.* RETURN(true)
5. ELSE
5.*a.* **if** $throw$ **then** THROW($TypeError$)
6. RETURN(false)

Finally, the [[DefaultValue]] property returns the default value for an object, which is always a primitive type of the language.

3.6 The Property Descriptor and Property Identifier Specification Type

The Property Descriptor type is used to explain the manipulation and reification of named property attributes. Values $\in$ *PropDescr* are records composed of named fields where each field's name is an attribute name and its value is a corresponding attribute value as specified in the ECMA standard [2](§8.6.1). In addition, any field may be present or absent.

The following ASM rules provide a formal description of AOs that operate upon Property Descriptor values.

IsAccessDescriptor$(desc) \equiv$
1. **if** $desc =$ undefined **then** RETURN(false)
2. RETURN($hasGet(desc)$ **or** $hasSet(desc)$)

IsDataDescriptor$(desc) \equiv$
1. **if** $desc =$ undefined **then** RETURN(false)
2. RETURN($hasValue(desc)$ **or** $hasWritable(desc)$)

IsGenericDescriptor$(desc) \equiv$
1. **if** $desc =$ undefined **then** RETURN(false)
2. CALL($IsAccessorDescriptor$, null, $desc$) **in**
3. $isAccDesc(pos, sp(pos)) = ret(pos, sp(pos))$
4. CALL($IsDataDescriptor$, null, $desc$)
5. $isDataDesc(pos, sp(pos)) = ret(pos, sp(pos))$
6. RETURN($\neg(isAccDesc(pos, sp(pos))$ **or** $isDataDesc(pos, sp(pos))))$

In order to make easier the following ASM rules definition we introduce a new ASM macro to create a Property Descriptor that takes a STRINGv, three *Boolean* w, e and c as input.

CreatePropDescriptor$(v, w, e, c) \equiv$
1. $desc(pos, sp(pos)) :=$ **new** $(PropDescr)$
2. $propAttrs(desc(pos, sp(pos)),$ "[[Value]]"$) := v$
3. $propAttrs(desc(pos, sp(pos)),$ "[[Writable]]"$) := w$
4. $propAttrs(desc(pos, sp(pos)),$ "[[Enumerable]]"$) := e$
5. $propAttrs(desc(pos, sp(pos)),$ "[[Configurable]]"$) := c$
6. **return** $desc(pos, sp(pos))$

3.7 The Reference Specification Type

A Reference Specification type is used to explain the behaviour of such operation as **delete**, **typeof**, and the assignment operators. Let *ECMAReference* be the domain of resolved name binding, its values are triples of the form:

$$ECMAReference = (Base \times \mathsf{String} \times \mathsf{Boolean})$$

where $Base = \{$undefined, Object, Boolean, String, Number, *EnvRec* $\}$. The ECMA standard defines Boolean, String and Number as primitive types. Therefore, we define $PrimitiveBase = \{$Boolean, String, Number $\}$ as the set of primitive types.

We access elements and establish properties of those triples through the following derived functions:

- $GetBase : ECMAReference \rightarrow Base$ returns the base value component of a reference, i.e. $GetBase(v) \equiv v \downarrow 1$.
- $GetReferencedName : ECMAReference \rightarrow$ String returns the referenced name component of a reference, i.e. $GetReferenceName(v) \equiv v \downarrow 2$.
- $IsStrictReference : ECMAReference \rightarrow Boolean$ returns the strict reference component of a reference, i.e. $IsStrictReference(v) \equiv v \downarrow 3$.
- $HasPrimitiveBase : ECMAReference \rightarrow Boolean$ returns true if the base value is $\in PrimitiveBase$, i.e.

$$HasPrimitiveBase(v) = \begin{cases} \mathsf{true} & \text{if } GetBase(\mathrm{v}) \in PrimitiveBase \\ \mathsf{false} & \text{otherwise} \end{cases}$$

- $IsPropertyReference : ECMAReference \rightarrow Boolean$ returns if either the base value is an Object or HasPrimitiveBase is true, i.e.

$$IsPropertyReference(v) = \begin{cases} \mathsf{true} & \text{if } GetBase(\mathrm{v}) = \mathsf{Object} \text{ or } HasPrimitiveBase(\mathrm{v}) \\ \mathsf{false} & \text{otherwise} \end{cases}$$

- *IsUnresolvableReference* : *ECMAReference* → *Boolean* returns **true** if the base value is **undefined** and **false** otherwise, i.e.

$$IsUnresolvableReference(v) = \begin{cases} \textsf{true} & \text{if } GetBase(\text{v}) = \textsf{undefined} \\ \textsf{false} & \text{otherwise} \end{cases}$$

There are several AOs that operate on references.

AO on references - ECMA standard [2](§8.7.1)

GetValue$(V) \equiv$
1. **if** $Type(V) \neq ECMAReference$ **then** RETURN(V)
2. $base(pos, sp(pos)) := GetBase(V)$
3. **if** $IsUnresolvableReference(V)$ **then** THROW$(ReferenceError)$
4. IF$(IsPropertyReference(V))$
4.*a.* IF$(\neg HasPrimitiveBase(V))$
4.*a.i.* $get(pos, sp(pos)) := propAttrs(base(pos, sp(pos)),$ "[[Get]]")
4.*b.* ELSE
4.*b.i.* $get(pos, sp(pos)) := SpecGet$
4.*c.* CALL$(GetReferencedName,$ null, false, $V)$
4.*d.* CALL$(get(pos, sp(pos)), base(pos, sp(pos)), ret(pos, sp(pos)))$
4.*e.* RETURN$(ret(pos, sp(pos)))$
5. ELSE
5.*a.* CALL$(IsStrictReference,$ null, $V)$
5.*b.* $strictRef(pos, sp(pos)) := ret(pos, sp(pos))$
5.*c.* CALL$(GetReferencedName,$ null, $V)$
5.*d.* $refName(pos, sp(pos)) := ret(pos, sp(pos))$
5.*e.* CALL$(GetBindingValue, base(pos, sp(pos)), refName(pos, sp(pos)), strictRef(pos, sp(pos)))$
5.*f.* RETURN$(ret(pos, sp(pos)))$

The following [[Get]] internal method is used by GetValue when V is a property reference with a primitive base value. It is called using *base* as its *this* value and with property p as its argument.

AO on references - ECMA standard [2](§8.7.1)

SpecGet$(base, p) \equiv$
1. CALL$(ToObject,$ **this**, $base)$
2. $o(pos, sp(pos)) := ret(pos, sp(pos))$
3. CALL$(propValue(o(pos, sp(pos)),$ "[[GetProperty]]"$), o(pos, sp(pos)), p)$
4. $desc(pos, sp(pos)) := ret(pos, sp(pos))$
5. **if** $desc(pos, sp(pos)) =$ undefined **then** RETURN(undefined)
6. CALL$(IsDataDescriptor,$ **this**, $desc(pos, sp(pos)))$ **then** RETURN(undefined)
7. **if** $ret(pos, sp(pos))$ **then** RETURN(undefined)
8. $getter(pos, sp(pos)) := propAttrs(desc(pos, sp(pos)),$ "[[Get]]")
9. **if** $getter(pos, sp(pos)) =$ undefined **then** RETURN(undefined)
10. CALL$(propValue(getter(pos, sp(pos)),$ "[[Call]]"$), getter(pos, sp(pos)), base)$
11. RETURN$(ret(pos, sp(pos)))$

The following [[Put]] internal method, called *SpecPut*, is used by *PutValue* when v is a property reference with a primitive base value. It is called using *base* as its *this* value and with property p, value w, and a Boolean flag *throw* as arguments.

AO on references - ECMA standard [2](§8.7.2)

SpecPut$(base, p, w, throw) \equiv$
1. CALL$(ToObject, \textbf{this}, base)$
2. $o(pos, sp(pos)) := ret(pos, sp(pos))$
3. CALL$(propValue(o(pos, sp(pos)), \text{“[[CanPut]]”}), o(pos, sp(pos)), p)$
4. IF$(\neg ret(pos, sp(pos)))$

4.*a* **if** *throw* **then** THROW$(TypeError)$

4.*b* **return**

5. CALL$(propValue(o(pos, sp(pos)), \text{“[[GetOwnProperty]]”}), o(pos, sp(pos)), p)$
6. $ownDesc(pos, sp(pos)) := ret(pos, sp(pos))$
7. CALL$(IsDataDescriptor, \textbf{this}, ownDesc(pos, sp(pos)))$
8. IF$(ret(pos, sp(pos)))$

8.*a* **if** *throw* **then** THROW$(TypeError)$

8.*b* **return**

9. CALL$(propValue(o(pos, sp(pos)), \text{“[[GetProperty]]”}), o(pos, sp(pos)), p)$
10. $desc(pos, sp(pos)) := ret(pos, sp(pos))$
11. CALL$(IsAccessorDescriptor, \textbf{this}, desc(pos, sp(pos)))$
12. IF$(ret(pos, sp(pos)))$

12.*a* $setter(pos, sp(pos)) := propAttrs(desc(pos, sp(pos)), \text{“[[Set]]”})$

12.*b* CALL$(propValue(setter(pos, sp(pos)), \text{“[[Call]]”}), setter(pos, sp(pos)), base, w)$

13. ELSE

13.*a* **if** *throw* **then** THROW$(TypeError)$

14. **return**

3.8 Lexical Environment

A Lexical Environment (*LexEnv*) associates an identifier to a specific variable and function based upon the lexical nesting structure of ECMAScript code. A Lexical Environment value is a pair $\langle env, refLexEnv \rangle$ where $env \in EnvRec$ and $refLexEnv \in LexEnv$ is a reference to an outer Lexical Environment. We access elements and establish properties of such pairs through the following derived functions:

- *envRecord* : *LexEnv* $\rightarrow$ *EnvRec* returns the environment record related to a given lexical environment.
- *outerLexEnv* : *LexEnv* $\rightarrow$ *LexEnv* returns the outer Lexical Environment of a given lexical environment.

The following abstract operations are used in this specification to operate upon lexical environments. The following *GetIdentifierReference* AO is called with a Lexical Environment *lex*, an identifier String *name*, and a Boolean flag *strict*. The value of *lex* may be null.

AO on Lexical Environment - ECMA standard [2](§10.2.2)

GetIdentifierReference$(lex, name, strict) \equiv$
1. **if** $lex =$ null **then** RETURN(**new** $(ECMAReference)$(undefined, $name, strict)$)
2. $envRec(pos, sp(pos)) := envRecord(lex)$
3. CALL$(HasBinding, envRec(pos, sp(pos)), name)$
4. $exists(pos, sp(pos)) := ret(pos, sp(pos))$
5. IF$(exists(pos, sp(pos)))$

5.*a* RETURN(**new** $(ECMAReference)(envRec(pos, sp(pos)), name, strict)$)

6. ELSE

6.*a* $outer(pos, sp(pos)) := outerLexEnv(lex)$

6.*b* CALL$(GetIdentifierReference, \text{null}, outer(pos, sp(pos)), name, strict)$

6.*c* RETURN$(ret(pos, sp(pos)))$

The following *NewDeclarativeEnvironment* AO is called with either a Lexical Environment or null as argument E.

AO on Lexical Environment - ECMA standard [2](§10.2.2)

NewDeclarativeEnvironment$(E) \equiv$
1. **let** $env(pos, sp(pos))$:= **new** $(LexEnv)$ **with**
2. $envRecord(env(pos, sp(pos)))$:= **new** $(DeclER)$
3. $bindObj(envRecord(env(pos, sp(pos))))$:= null
4. $outerLexEnv(env(pos, sp(pos)))$:= E
5. RETURN$(env(pos, sp(pos)))$

where *bindObj* function is defined in Section 3.11.

The following *NewObjectEnvironment* AO is called with an Object O, a Lexical Environment E (or null) and a *Boolean* B as arguments. The last parameter is introduced to make the formal definition of the *WithStatement* (see Section **??**) easier.

AO on Lexical Environment - ECMA standard [2](§10.2.2)

NewObjectEnvironment$(O, E, B) \equiv$
1. **let** $env(pos, sp(pos))$:= **new** $(LexEnv)$ **with**
2. $envRecord(env(pos, sp(pos)))$:= **new** $(ObjER)$
3. $outerLexEnv(env(pos, sp(pos)))$:= E
4. $bindObj(envRecord(env(pos, sp(pos))))$:= O
5. $provideThis(envRecord(env(pos, sp(pos)))$:= B
6. RETURN$(env(pos, sp(pos)))$

where *provideThis* function is defined in Section 3.11.

3.8.1 The Global Environment

The global environment, called *GlobalEnv*, is a unique Lexical Environment which is created before any ECMAScript code is executed. The global environment's Environment Record is an object environment record whose binding object is the *global object* [2](§15.1) and such that *outerLexEnv*(*GlobalEnv*) is null.

3.9 Environment Record

An Environment Record records the identifier bindings $\in$ *Binding* that are created within the scope of its associated Lexical Environment. For specification purposes Environment Record values can be thought of as existing in a simple object-oriented hierarchy where Environment Record is an abstract class with two concrete sub-classes, declarative and object environment records. An identifier binding has the following attributes and functions on them:

Identifier Binding attributes

ATTRIBUTES = $\{Mutable, Deletable, Initialize\}$
bindAttrs : *Binding* $\rightarrow$ ATTRIBUTES

isMutable : *Binding* $\rightarrow$ *Boolean*
isMutable(b) = $(Mutable \in bindAttrs(b))$
isDeletable : *Binding* $\rightarrow$ *Boolean*
isDeletable(b) = $(Deletable \in bindAttrs(b))$
isInitialized : *Binding* $\rightarrow$ *Boolean*
isInitialized(b) = $(Initialize \in bindAttrs(b))$

3.10 Declarative Environment Record

Each declarative environment record is associated with an ECMAScript program scope containing variable and/or function declarations. A declarative environment record binds the set of identifiers defined by the declarations contained within its scope. Therefore, we define the following function:

$$bindValue : EnvRec \times \text{STRING} \rightarrow Binding$$

that returns a binding for the name that is given as input, if present, *undef* otherwise.

In addition to the mutable bindings supported by all Environment Records, declarative environment records also provide immutable bindings. An immutable binding is one where the association between an identifier and a value may not be modified once it has been established.
The following ASM rules provide a formal description of AOs for this type of record.

The concrete environment record method *HasBinding* for declarative environment records simply determines if the argument identifier is one of the identifiers bound by the record.

Declarative Environment Record - ECMAScript [2](§10.2.1.1)

HasBinding$(lex, n) \equiv$
1. $envRec(pos, sp(pos)) := envRecord(lex)$
2. RETURN$(bindValue(envRec(pos, sp(pos)), n) \neq undef)$

The concrete Environment Record method $CreateMutableBinding$ for declarative environment records creates a new mutable binding for the name n that is initialized to the value undefined. A binding must not already exist in this Environment Record for n. If Boolean argument d is provided and has the value true the new binding is marked as being subject to deletion. This is formalized as follows:

Declarative Environment Record - ECMAScript [2](§10.2.1.1)

CreateMutableBinding$(lex, n, d) \equiv$
1. $envRec(pos, sp(pos)) := envRecord(lex)$
2. CALL$(HasBinding, \textbf{this}, lex, n)$
3. **if** $ret(pos, sp(pos))$ **then** THROW$(ReferenceError)$
4. $bindValue(envRec(pos, sp(pos)), n) :=$ undefined
5. **if** d **then** CALL$(DeleteBinding, \textbf{this}, bindValue(envRec(pos, sp(pos)), n))$

The concrete Environment Record method $SetMutableBinding$ for declarative environment records attempts to change the bound value of the current binding of the identifier whose name is the value of the argument n to the value of argument v. A binding for n must already exist. If the binding is an immutable binding, a TypeError is always thrown. The s argument is ignored because strict mode does not change the meaning of setting bindings in declarative environment records.

Declarative Environment Record - ECMAScript [2](§10.2.1.1)

SetMutableBinding(lex, n, v, s)
1. $envRec(pos, sp(pos)) := envRecord(lex)$
2. CALL$(HasBinding, \textbf{this}, lex, n)$
3. **if** $ret(pos, sp(pos))$ **then** THROW$(ReferenceError)$
4. IF$(isMutable(bindValue(envRec(pos, sp(pos)), n)))$
4.*a.* $bindValue(envRec(pos, sp(pos)), n)) := v$
5. ELSE
5.*a.* THROW$(TypeError)$

The concrete Environment Record method $GetBindingValue$ for declarative environment records simply returns the value of its bound identifier whose name is the value of the argument N. The binding must already exist. If S is true and the binding is an uninitialized immutable binding throw a ReferenceError exception.

Declarative Environment Record - ECMAScript [2](§10.2.1.1)

GetBindingValue(lex, n, s)
1. $envRec(pos, sp(pos)) := envRecord(lex)$
2. CALL$(HasBinding, \textbf{this}, lex, n)$
3. **if** $ret(pos, sp(pos))$ **then** THROW$(ReferenceError)$
4. IF$(bindValue(envRec(pos, sp(pos)), n) =$ undefined$)$
4.*a.* IF$(\neg s)$
4.*a.i.* RETURN(undefined)
4.*b.* ELSE
4.*b.i.* THROW$(ReferenceError)$
5. ELSE
5.*a.* RETURN$(bindValue(envRec(pos, sp(pos)), n))$

The concrete Environment Record method *DeleteBinding* for declarative environment records can only delete bindings that have been explicitly designated as being subject to deletion.

Declarative Environment Record - ECMAScript [2](§10.2.1.1)

DeleteBinding(lex, n)

1. $envRec(pos, sp(pos)) := envRecord(lex)$
2. CALL$(HasBinding, \textbf{this}, lex, n)$
3. **if** $\neg ret(pos, sp(pos))$ **then** RETURN(true)
4. CALL$(isDeletable, bindValue(envRec(pos, sp(pos)), n))$
5. **if** $\neg ret(pos, sp(pos))$ **then** RETURN(false)
6. $bindValue(envRec(pos, sp(pos)), n)) := undef$
7. RETURN(true)

Declarative Environment Records always return undefined as their $ImplicitThisValue$.

Declarative Environment Record - ECMAScript [2](§10.2.1.1)

ImplicitThisValue()

1. RETURN(undefined)

The concrete Environment Record method $CreateImmutableBinding$ for declarative environment records creates a new immutable binding for the name N that is initialized to the value undefined. A binding must not already exist in this environment record for N.

Declarative Environment Record - ECMAScript [2](§10.2.1.1)

CreateImmutableBinding(n)

1. $envRec(pos, sp(pos)) := envRecord(lex)$
2. CALL$(HasBinding, \textbf{this}, lex, n)$
3. **if** $ret(pos, sp(pos))$ **then** THROW$(ReferenceError)$
4. $bindValue(envRec(pos, sp(pos)), n)) :=$ **new** $(Binding)$
5. $isMutable(bindValue(envRec(pos, sp(pos)), n)) :=$ false
6. $isInitialized(bindValue(envRec(pos, sp(pos)), n)) :=$ false

The concrete Environment Record method $InitializeImmutableBinding$ for declarative environment records is used to set the bound value of the current binding of the identifier whose name is the value of the argument N to the value of argument V. An uninitialized immutable binding for N must already exist.

Declarative Environment Record - ECMAScript [2](§10.2.1.1)

InitializeImmutableBinding(n, v)

1. $envRec(pos, sp(pos)) := envRecord(lex)$
2. CALL$(HasBinding, \textbf{this}, lex, n)$
3. **if** $ret(pos, sp(pos))$ **then** THROW$(ReferenceError)$
4. $bindValue(envRec(pos, sp(pos)), n)) := v$
5. $isMutable(bindValue(envRec(pos, sp(pos)), n)) :=$ false
6. $isInitialized(bindValue(envRec(pos, sp(pos)), n)) :=$ true

3.11 Object Environment Record

An Object Environment record binds the set of identifier names that directly correspond to the property names of its binding object. Each Object Environment record is associated with an object called its binding object. We define the following

$$bindObj : EnvRec \rightarrow \text{Object}$$

as the function that returns the object binds to a given environment record. In order to make the formal definition of the *WithStatement* (see Section **??**) easier we define the following

$$provideThis : EnvRec \rightarrow Boolean$$

as the function that returns whether for a given Environment Record the *this* reference is provided or not.

The following ASM rules provide a formal description of AOs for this record.

The concrete Environment Record method *HasBinding* for object environment records determines if its associated binding object has a property whose name is the value of the argument n.

Object Environment Record - ECMAScript [2](§10.2.1.2)

HasBinding$(lex, n) \equiv$

1. $envRec(pos, sp(pos)) := envRecord(lex)$
2. $bindings(pos, sp(pos)) := bindObj(envRec(pos, sp(pos)))$
3. CALL($propValue(bindings(pos, sp(pos))$, "[[HasProperty]]"$), bindings(pos, sp(pos)), n$)
4. RETURN($ret(pos, sp(pos))$)

The concrete Environment Record method *CreateMutableBinding* for object environment records creates in an environment record's associated binding object a property whose name is the String value and initializes it to the valueundefined. A property named n must not already exist in the binding object. If Boolean argument d is provided and has the value true the new property's [[Configurable]] attribute is set to true, otherwise it is set to false.

Object Environment Record - ECMAScript [2](§10.2.1.2)

CreateMutableBinding$(lex, n, d) \equiv$

1. $envRec(pos, sp(pos)) := envRecord(lex)$
2. $bindings(pos, sp(pos)) := bindObj(envRec(pos, sp(pos)))$
3. CALL($propValue(bindings(pos, sp(pos))$, "[[HasProperty]]"$), bindings(pos, sp(pos)), n$)
4. **if** $ret(pos, sp(pos))$ **then** $Throw(...)$
5. $configValue(pos, sp(pos)) := d$
6. CALL($CreatePropDescriptor$, undefined, true, true, $configValue(pos, sp(pos))$)
7. $newDesc(pos, sp(pos)) := ret(pos, sp(pos))$
8. CALL($propValue(bindings(pos, sp(pos))$, "[[DefineOwnProperty]]"$), bindings(pos, sp(pos)), n,$
   $newDesc(pos, sp(pos))$, false)

The concrete Environment Record method *SetMutableBinding* for object environment records attempts to set the value of the environment record's associated binding object's property whose name is the value of the argument n to the value of argument v. A property named n should already exist but if it does not or is not currently writable, error handling is determined by the value of the Boolean argument s.

Object Environment Record - ECMAScript [2](§10.2.1.2)

SetMutableBinding(lex, n, v, s)

1. $envRec(pos, sp(pos)) := envRecord(lex)$
2. $bindings(pos, sp(pos)) := bindObj(envRec(pos, sp(pos)))$
3. CALL($propValue(bindings(pos, sp(pos))$, "[[HasProperty]]"$), bindings(pos, sp(pos)), n, v, s$)

The concrete Environment Record method *GetBindingValue* for object environment records returns the value of its associated binding object's property whose name is the String value of the argument identifier n. The property should already exist but if it does not the result depends upon the value of the s argument

Object Environment Record - ECMAScript [2](§10.2.1.2)

GetBindingValue(lex, n, s)

1. $envRec(pos, sp(pos)) := envRecord(lex)$
2. $bindings(pos, sp(pos)) := bindObj(envRec(pos, sp(pos)))$
3. CALL($propValue(bindings(pos, sp(pos))$, "[[HasProperty]]"$), bindings(pos, sp(pos)), n$)
4. $value(pos, sp(pos)) := ret(pos, sp(pos))$
5. IF($\neg value$)

5.*a*. IF($\neg s$)

5.*a.i* RETURN(undefined)

5.*b*. ELSE

5.*b.i*. THROW($ReferenceError$)

6. CALL($propValue(bindings(pos, sp(pos))$, "[[Get]]"$), bindings(pos, sp(pos)), n$)
7. RETURN($ret(pos, sp(pos))$)

The concrete Environment Record method *DeleteBinding* for object environment records can only delete bindings that correspond to properties of the environment object whose [[Configurable]] attribute have

the value true.

Object Environment Record - ECMAScript [2](§10.2.1.2)

DeleteBinding(lex, n)

1. $envRec(pos, sp(pos)) := envRecord(lex)$
2. $bindings(pos, sp(pos)) := bindObj(envRec(pos, sp(pos)))$
3. CALL$(propValue(bindings(pos, sp(pos)),$ "[[Delete]]"$), bindings(pos, sp(pos)), n)$
4. RETURN$(ret(pos, sp(pos)))$

Object Environment Records return undefined as their *ImplicitThisValue* unless their *provideThis* flag is true.

Object Environment Record - ECMAScript [2](§10.2.1.2)

ImplicitThisValue(lex)

1. $envRec(pos, sp(pos)) := envRecord(lex)$
2. IF$(provideThis(envRecord(lex)))$
2.a. RETURN$(bindObj(envRecord(lex)))$
3. ELSE
3.a. RETURN(undefined)

3.12 Executable Code

There are three types of ECMAScript executable code that differ based on the part of a ECMAScript code they belong to.

- *Global*, it is the source text of an ECMAScript *Program*. It does not include any source text that is parsed as part of a *FunctionBody*.
- *Eval*, it is the source text supplied to the built-in eval function.
- *Function*, it is the source text of a *FunctionBody*. The function code of a particular *FunctionBody* does not include any source text that is parsed as part of a nested *FunctionBody*. Function code also denotes the source text supplied when using the built-in Function object as a constructor.

An ECMAScript executable code may be processed using either *unrestricted* or *strict* mode syntax and semantics. When processed using strict mode the three types of ECMAScript code are referred to as *strict global* code, *strict eval* code, and *strict function* code.
This is formalized by the following oracle functions:

$$isFunCode : \text{code} \rightarrow \text{Boolean}$$

that determines whether a given ECMAScript code is a function code or not.

Let FuncDecl be the domain of function declarations in a function code

$$getFunDeclaration : \text{code} \rightarrow \text{FUNCDECL}$$

returns a list of function declarations in a given ECMAScript code.

Let FunIdentifier be the domain of identifiers in a function declaration

$$getFuncIdentifier : \text{FUNCDECL} \rightarrow \text{FUNIDENTIFIER}$$

returns an identifier in a given function declaration.

$$getFuncDeclInstance : \text{FUNCDECL} \rightarrow \text{Object}$$

that returns an instance of a given function declaration.

$$isEvalCode : \text{code} \rightarrow \text{Boolean}$$

that determines whether a given ECMAScript code is a eval code or not.

$$isStrictCode : \mathsf{code} \rightarrow \mathsf{Boolean}$$

that determines whether a given ECMAScript code is a strict mode code or not.

Let VarDecl be the domain of variable declarations in a function code

$$getVarDeclaration : \mathsf{code} \rightarrow \textsc{VarDecl}$$

returns a list of variable declarations in a given ECMAScript code.

Let VarIdentifier be the domain of identifiers in a variable declaration

$$getVarIdentifier : \textsc{VarDecl} \rightarrow \textsc{VarIdentifier}$$

returns an identifier in a given variable declaration.

3.13 Execution context

When control is transferred to an ECMAScript executable program it enters an *execution context*. Multiple execution contexts may be active while the program is running, and they are organized in a logical stack whose top element is the *running execution context*.

Execution contexts are created either when control is transferred to ECMASCript executable code or when a function or a constructor get called. Their purpose is to store the local state of the invocation (i.e. arguments and local variables), and all the information needed to resolve scopes and access to the program state.

An execution context contains:

- A $LexicalEnvironment \in LexEnv$ that identifies the Lexical Environment used to resolve identifier references made by code within this execution context. We define the following function

  $$lexEnvEC : ExeCtx \rightarrow LexEnv$$

  to access this execution context's component.

- A $VariableEnvironment \in LexEnv$ that identifies the Lexical Environment whose environment record holds bindings created by *VariableStatements* and *FunctionDeclarations* within this execution context. We define the following function

  $$varEnvEC : ExeCtx \rightarrow LexEnv$$

  to access this execution context's component.

- A reference, called $ThisBinding$, to an object called *this*. We define the following function

  $$thisBindingEC : ExeCtx \rightarrow ECMAReference$$

  to access this execution context's component.

The value of the $VariableEnvironment$ component never changes while the value of the $LexicalEnvironment$ component may change during execution of code within an execution context.

3.13.1 Establishing an Execution Context

Evaluation of *global* code or code using the eval function establishes and enters a new execution context. Every invocation of an ECMAScript code *function* also establishes and enters a new execution context, even if a function is calling itself recursively. Every return exits an execution context. A thrown exception may also exit one or more execution contexts.

When control enters an execution context, the execution context's *ThisBinding* is set, its *VariableEnvironment* and initial *LexicalEnvironment* are defined, and declaration binding instantiation is performed.

The following ASM rules provide a formal description of AOs for establishing an execution context having a code C as input.

Establishing an Execution Context - ECMAScript [2](§10.4)

InitGlobalExeCtx$(C) \equiv$

1. $varEnvEC(codeExCtx(C)) := GlobalEnv$
2. $lexEnvEC(codeExCtx(C)) := GlobalEnv$
3. $thisBindingEC(codeExCtx(C)) := bindObj(GlobalEnv)$

Establishing an Execution Context - ECMAScript [2](§10.4)

EnteringGlobalCode$(C) \equiv$

1. CALL$(InitGlobalExeCtx, \text{null}, C)$
2. CALL$(BindInst, \textbf{this}, bindObj(GlobalEnv), C, \text{null})$

The following steps are performed when control enters the execution context for function code contained in function object F, a caller provided $thisArg$ and an $argumentsList$:

Establishing an Execution Context - ECMAScript [2](§10.4)

EnteringFuncCode$(F, code, thisArg, argumentsList) \equiv$

1. IF$(codeMode(code) =$ "strict")
1.*a.* $thisBindingEC(codeExCtx(code)) := thisArg$
2. ELSE
2.*a.* IF$(thisArg =$ null **or** $thisArg =$ undefined)
2.*a.i.* $thisBindingEC(codeExCtx(code)) := bindObj(GlobalEnv)$
2.*b.* ELSE
2.*b.i.* IF$(Type(thisArg) \neq$ Object)
2.*b.i.*1. CALL$(ToObject, \textbf{this}, thisArg)$
2.*b.i.*2. $thisBindingEC(codeExCtx(code)) := ret(pos, sp(pos))$
2.*b.ii.* ELSE
2.*b.ii.*1. $thisBindingEC(codeExCtx(code)) := thisArg$
3. CALL$(NewDeclarativeEnvironment, F, propValue(F,$ "[[Scope]]"$))$
4. $localEnv(pos, sp(pos)) := ret(pos, sp(pos))$
5. $lexEnvEC(codeExCtx(code)) := localEnv(pos, sp(pos))$
6. $varEnvEC(codeExCtx(code)) := localEnv(pos, sp(pos))$
7. $cod := propValue(F,$ "[[Code]]"$)$
8. CALL$(BindInst, F, cod, argumentsList)$

3.14 Declarations

We define the following functions:

- *varDeclId* : VARIABLEDECLARATION → String that returns the identifier of a given variable declaration.
- *funcDeclId* : FUNCTIONDECLARATION → String that returns the identifier of a given function declaration.

3.14.1 Binding Instantiation

Which Environment Record is used to bind a declaration and its kind depends upon the type of ECMAScript code executed by the execution context. On entering an execution context, bindings are created in the VariableEnvironment as follows using the caller provided code and, if it is function code, argument List args.

Execution Contexts - Declaration Binding

BindInst$(obj, code, args) \equiv$

1. $env(pos, sp(pos)) := varEnvEC(codeExCtx(code))$
2. $configurableBindings(pos, sp(pos)) := (codeType(code) =$ "eval")
3. $strict(pos, sp(pos)) := (codeMode(code) =$ "strict")
4. Call$(isFunCode, code)$
5. If$(ret(pos, sp(pos)) =$ "function")

5.a. Call$(FunCodeBind, obj, args)$
5.b. Call$(getFunDeclaration, code)$
5.c. For$(ret(pos, sp(pos)), f(pos, sp(pos)))$
5.c.i. Call$(getFuncIdentifier, f(pos, sp(pos)))$
5.c.ii. $fn(pos, sp(pos)) := ret(pos, sp(pos))$
5.c.iii. Call$(getFuncDeclInstance, f(pos, sp(pos)))$
5.c.iv. $fo(pos, sp(pos)) := ret(pos, sp(pos))$
5.c.v. Call$(HasBinding, fn(pos, sp(pos)))$
5.c.vi. $funcAlreadyDeclared(pos, sp(pos)) := ret(pos, sp(pos))$
5.c.vii. **if** $\neg funcAlreadyDeclared(pos, sp(pos))$ **then** Call$(CreateMutableBinding, fn(pos, sp(pos)),$ $configurableBindings(pos, sp(pos)))$
5.c.viii. Call$(SetMutableBinding, env(pos, sp(pos)), fn(pos, sp(pos)), fo(pos, sp(pos)), strict(pos, sp(pos)))$
5.d. EndFor

6. Call$(HasBinding, env(pos, sp(pos)),$ "arguments")
7. $argumentsAlreadyDeclared(pos, sp(pos)) := ret(pos, sp(pos))$
8. Call$(isFunCode, code)$
9. If$(ret(pos, sp(pos)) ==$ "function" **and** $\neg argumentsAlreadyDeclared(pos, sp(pos)))$

9.a. Call$(CreateArgumentsObject,$ **this**$, func(pos, sp(pos)), names(pos, sp(pos)), args(pos, sp(pos)),$ $env(pos, sp(pos)), strict(pos, sp(pos)))$
9.b. $argsObj(pos, sp(pos)) := ret(pos, sp(pos))$
9.c. If$(strict(pos, sp(pos)))$
9.c.i. Call$(CreateImmutableBinding, env(pos, sp(pos)),$ "arguments")
9.c.ii. Call$(InitializeImmutableBinding, env(pos, sp(pos)),$ "arguments"$, argsObj(pos, sp(pos)))$
9.d. Else
9.d.i. Call$(CreateMutableBinding, env(pos, sp(pos)),$ "arguments")
9.d.ii. Call$(SetMutableBinding, env(pos, sp(pos)),$ "arguments"$, argsObj(pos, sp(pos)),$ false)

10. Call$(VarCodeBin, code)$
11. Return

Execution Contexts - Declaration Binding

FunCodeBind$(obj, args) \equiv$

1. $func(pos, sp(pos)) := obj$
2. Call$(propValue(func(pos, sp(pos)),$ "[[FormalParameters]]")
3. $names(pos, sp(pos)) := ret(pos, sp(pos)))$ 4. $v(pos, sp(pos)) :=$ Undefined
5. For$(names(pos, sp(pos)), argName(pos, sp(pos)))$

5.a. $v(pos, sp(pos)) := args(n)$
5.b. Call$(HasBinding, env(pos, sp(pos)), argName(pos, sp(pos)))$
5.c. $argAlreadyDeclared(pos, sp(pos)) := ret(pos, sp(pos))$
5.d. **if** $\neg argAlreadyDeclared(pos, sp(pos))$ **then** Call$(CreateMutableBinding, env(pos, sp(pos)),$ $argName(pos, sp(pos)))$
5.e. Call$(SetMutableBinding, env(pos, sp(pos)), argName(pos, sp(pos)), strict(pos, sp(pos)))$

6. EndFor

where args is a oracle function that returns a list of argument if a given code is a function code.

VarCodeBind$(code) \equiv$
1. CALL$(getVarDeclaration, code)$
2. FOR$(ret(pos, sp(pos)), d)$
2.*a.* CALL$(getVarIdentifier, d)$
2.*a.* $dn(pos, sp(pos)) := ret(pos, sp(pos))$
2.*b.* CALL$(HasBinding, env(pos, sp(pos)), dn(pos, sp(pos)))$
2.*c.* $varAlreadyDeclared(pos, sp(pos)) := ret(pos, sp(pos))$
2.*d.* IF$(\neg varAlreadyDeclared(pos, sp(pos)))$
2.*d.i* CALL$(CreateMutableBinding, env(pos, sp(pos)), dn(pos, sp(pos)), configurableBindings(pos, sp(pos)))$
2.*d.ii.* CALL$(SetMutableBinding, env(pos, sp(pos)), dn(pos, sp(pos)),$ undefined$, strict(pos, sp(pos)))$
3. ENDFOR

4 Conclusions

Web applications (i.e., applications whose interface is presented to the user via a web browser, whose state is split between a server and a client, and where the only interaction between server and client is through the HTTP protocol) are become more and more widespread, and integrated in most users' everyday work habits. The glue linking the desperate technologies involved, from dynamic HTML to XML RPC, is the Javascript language. Yet, no formal definition of its semantics exists, which in turn makes it impossible to formally prove correctness, liveness and security properties of Web applications. As a first step towards improving this situation, we provide a formal semantics for ECMAScript (standard ECMA-262), by means of an Abstract State Machines (ASMs) specification. We follow the path established by other specification efforts for similar languages (e.g. Java and C#), but in addition we establish a formal trace between parts of our specification and the ECMA standard, thus facilitating the proof of correctness of the specification. More specifically, we define the dynamic semantics of ECMAScript by providing (in terms of ASMs) an interpreter which executes ECMAScript programs. We use an algebra to represent the state of the ECMAScript program, the state of the ECMAScript interpreter, and the state of the host environment (typically, the browser). We assume the necessary syntactic information (namely, the annotated Abstract Syntax Tree (AST) of the given program) to be available. Using ASM we detail a visit of the AST, whose effect is to simulate the execution of the program. In the ECMA standard the behavior of ECMAScript constructs is operationally described using so-called abstract operations (AO), coming as numbered lists of steps. Therefore our interpreter is composed out of two submachines: *ECMA-Script* and *ECMA-AO* interpreters. The former invokes the *ECMA-AO* one to evaluate nodes, simulating AOs used in ECMA standard to describe the production's semantics. We organize the *ECMA-AO* interpreter rules in such a way that each rule application corresponds to an AO instruction in the ECMA standard. This helps to check the correctness of the ASM model wrt the ECMA standard as well as the correctness of implementations wrt ASM model. Therefore, the state of our interpreter can be mapped via abstraction/refinement functions onto the concrete state of any comformant implementation.

References

[1] E. Börger and R. Stärk. *Abstract State Machines A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[2] ECMA. *Standard ECMA-262, ECMAScript Language Definition*. ECMA, fifth edition, 2009.

[3] R. Farahbod, V. Gervasi, and U. Glaesser. CoreASM: An extensible ASM execution engine. *Fundamenta Informaticae*, 77:71–103, Mar./Apr. 2007.