DASAP Project Meeting, Milano Feb 2010

A formal specification of ECMAScript

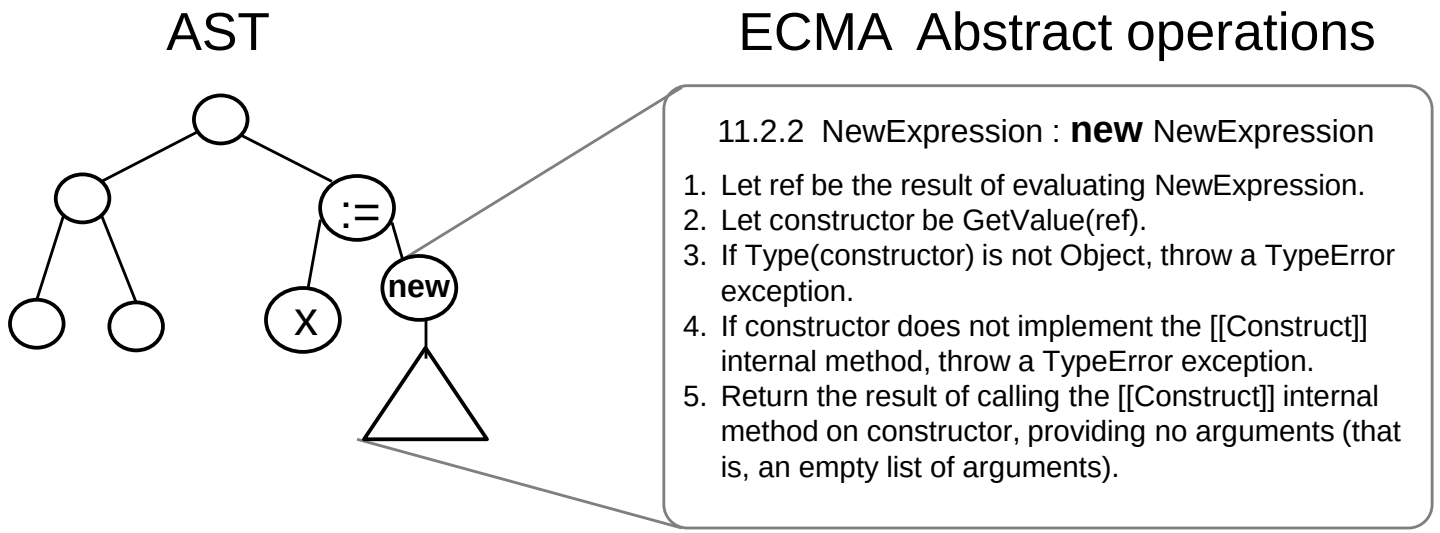
Cristian Dittamo
Dipartimento di Informatica
Università di Pisa

ECMA - 262

- ECMA Script Language Specification
- Based on several originating technologies, i.e. Javascript (Netscape), Jscript (Microsoft)
- Standard development evolution:
  - starts in November 1996,
  - 1th ed. June 1997 (appeared in Netscape Navigator 2.0 and Internet Explorer 3.0)
  - 2nd ed. June 1998 (fully aligned with ISO/IEC 16262)
  - 3rd ed. December 1999 (new control statement, exception handling)
  - **5th edition, December 2009** (accessor properties, reflective creation and inspection of objects, and support for the JSON objects)

ECMAScript Interpreter

- Collection of rules
- Traverse a parse tree while evaluating values and locations



- ECMA Abstract operation: precisely specify the required semantics of ECMAScript language constructs

ECMAScript Interpreter

1. Tree navigation

$$first: NODE \rightarrow NODE$$
$$next: NODE \rightarrow NODE$$
$$parent: NODE \rightarrow NODE$$

2. Syntactical class of a node (e.g. Literal)

$$class: NODE \rightarrow CLASS$$

3. Syntactical token represented by the node

$$token: NODE \rightarrow TOKEN$$

3. Symbolic name for the specific grammar pattern corresponding to the node (e.g. IFStatement)

$$pattern: NODE \rightarrow PATTERN$$

ECMAScript Interpreter

- Result of a node interpretation

$$[\![.]\!] : NODE \rightarrow Reference \times Completion \times Value$$

  - *Reference* is a l-value of an expression

$$ref: NODE \rightarrow REFERENCE \qquad i.e.\ ref(n) \equiv [\![n]\!] \downarrow 1$$

  - *Completion* status of the evaluation of an expression or statement

$$compl: NODE \rightarrow COMPLETION \qquad i.e.\ compl(n) \equiv [\![n]\!] \downarrow 2$$

  - *Value* is the r-value of an expression

$$value: NODE \rightarrow VALUE \qquad i.e.\ value(n) \equiv [\![n]\!] \downarrow 3$$

- Interpretation proceeds by computing, node by node, the value of $[\![.]\!]$ for each node of the AST
  - Current position on the AST

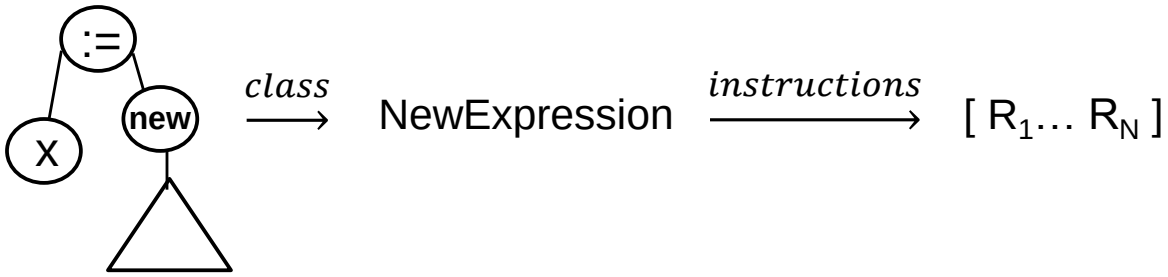
$$pos: \rightarrow NODE$$

ECMAScript to ECMA-AO

- List of steps in the ECMA abstract operations used to specify the semantics of a node class

  $instruction: CLASS \rightarrow INSTRLIST$

- $INSTRLIST$ is a list of ASM rules indexed by an hierarchical enumeration conformant to the one used in the ECMA specification
  - standard ordering and nesting structure



ECMA-AO Interpreter

- Instruction reference indicates the current instruction in the ECMA abstract operations

  $pc\colon NODE \rightarrow INSTRREF$

- Each R_i is an ASM rule
  - we use macros for control structures (including ECMA-AO exception)
- State
  - Local environment
  - Stack

Rules

$$\boldsymbol{Return}(\boldsymbol{v}) \equiv value(pos) \coloneqq v$$

$$\boldsymbol{Throw}(\boldsymbol{e}) \equiv exception(pos) \coloneqq e$$

$$\boldsymbol{Goto}(\boldsymbol{i}) \equiv pc(pos) \coloneqq i$$

$$\boldsymbol{IF}(\boldsymbol{cond}, \boldsymbol{i1}, \boldsymbol{i2}) \equiv$$

$$if\ cond\ then\ pc(pos) \coloneqq i1\ else\ pc(pos) \coloneqq i2$$

$$\boldsymbol{ELSE} \equiv nop$$

Main interpreter rule

$$
\begin{aligned}
&INTERPRETER \equiv \\
&\quad if\ pos \neq undef\ then \\
&\quad\quad if\ aomode(pos) = idle\ then \\
&\quad\quad\quad ESINTERPRET \\
&\quad\quad else \\
&\quad\quad\quad AOINTERPERT
\end{aligned}
$$

ECMAScript Interpreter

$$
\begin{aligned}
&ESINTERPRETER \equiv \\
&\quad if\ \neg evaluated(pos)\ then \\
&\qquad aomode(pos) \coloneqq initialize \\
&\quad else \\
&\qquad pos \coloneqq parent(pos)
\end{aligned}
$$

ECMA-AO Interpreter

$$
\begin{aligned}
&AOINTERPRET \equiv \\
&\quad let\ il = instructions\big(class(pos)\big)\ in \\
&\quad\quad if\ aomode(pos) = initialize\ then \\
&\quad\quad\quad pc(pos) \coloneqq first(il) \\
&\quad\quad\quad aomode(pos) \coloneqq running \\
&\quad\quad if\ aomode(pos) = running\ then \\
&\quad\quad\quad let\ R = il\big(pc(pos)\big)\ in \\
&\quad\quad\quad\quad R(\dots) \\
&\quad\quad if\ aomode(pos) = finalize\ then \\
&\quad\quad\quad aomode(pos) \coloneqq idle
\end{aligned}
$$

Example – new operator

NewExpression : new NewExpression

ECMA specification

1. Let ref be the result of evaluating NewExpression.
2. Let r be GetValue(ref).
3. If Type(constructor) is not Object, throw a TypeError exception.
4. If constructor does not implement the [[Construct]] internal method, throw a TypeError exception.
5. Return the result of calling the [[Construct]] internal method on constructor, providing no arguments (that is, an empty list of arguments).

ASM Rules

1. $r \coloneqq GetValue\big(ref(NewExpression)\big)$
2. if $Type(r) \neq Object\ then\ Throw(TypeError)$
3. $if\ \neg implements([[Construct]])\ then\ Throw(TypeError)$
4. $c \coloneqq Call(r, [[Construct]], <>)$
5. $Return(c)$

Example – GetValue(V)

ECMA specification

1. If Type(V) is not Reference, then return V
2. Let base be the result of calling GetBase(V)
3. If IsUnresolvableReference(V), throw a ReferenceError exception.
4. If IsPropertyReference(V), then
   a. If HasPrimitiveBase(V) is false, then let get be the [[Get]] internal method of base, otherwise let get be the special [[Get]] internal method
   b. Return the result of calling the get internal method using base as its this value, and passing GetReferencedName(V) for the argument
5. Else
   a. Return the result of calling the GetBindingValue concrete method of base passing GetReferenceName(V) and IsStrictReference(V) as arguments.

ASM Rules

1. $if\ Type(V) \neq Reference\ then\ Return(V)$
2. $base := GetBase(V)$
3. $if\ IsUnresolvableReference(V)\ then$
   $Throw(ReferenceErorr)$
4. $IF(IsPropertyReference(V), 4.a, 5.a)$
   a. $IF(\neg isPrimitiveBase(V), 4.a.i, 4.b.i)$
      i. $get := Call(base, [[Get]], GetReferenceName(V))$
   b. $ELSE$
      i. $get := Call(base, [[sGet]], GetReferenceName(V))$
   c. $Return(get)$
5. $ELSE$
   a. $bv := GetBindingValue(\ GetReferenceName(V),$
      $IsStrictReference(V))$
   b. $Return(bv)$

ECMA Interpreter - Object

- Collection of properties, where each property is a triple

$$\langle name, value, attributes \rangle$$

  - $name$ is a string
  - $value$ is any legal ECMAScript value or an ASM rule
  - $attributes$ is a set of values from a given set of predefined features

ECMA Interpreter – Property modeling

Attribute Name	Value Domain
[[Value]]	Any ECMAScript language type
[[Writable]]	Boolean
[[Enumerable]]	Boolean
[[Configurable]]	Boolean

- Named data property

$$ATTRIBUTE = \{Value, Writable, Enumerable, Configurable\}$$

$$propValue : OBJECT \times STRING \rightarrow VALUE$$

$$propAttrs : OBJECT \times STRING \rightarrow P(ATTRIBUTE)$$

$$isWriteble: OBJECT \times STRING \rightarrow BOOLEAN$$

$$isWritable\ (o,p) = \big(Writable \in propAttrs(o,p)\big)$$

$$isEnumerable : OBJECT \times STRING \rightarrow BOOLEAN$$

$$isEnumerable\ (o,p) = \big(Enumerable \notin propAttrs(o,p)\big)$$

$$isConfigurable: OBJECT \times STRING \rightarrow BOOLEAN$$

$$isConfigurable(o,p) = \big(Configurable \notin propAttrs(o,p)\big)$$