Minutes of the: ***Ecma TC39, ES3.1WG***

held in: ***Phone conference***

on: ***20 January 2009***

1 Roll call and logistics

1.1 Participants

Doug Crockford (Yahoo!), Pratap Lakshman (Microsoft), Mark Miller (Google) and Allen Wirfs-Brock (Microsoft)

2 Agenda

Not circulated ahead of time.

3 Minutes

[[Prototype]] on bound functions

Multiple options (from email on the discuss lists):

(1) bound functions do not have a “prototype” property, cannot be used as constructors, and cannot be used with instanceOf.

(2) bound functions do not have a “prototype”, delegate to the target function for construction and instanceOf using the target function’s “prototype” property.

(3) bound functions have a “prototype” property whose value is bidirectionally shared (in the arguments object sense) with the “prototype” property of the target function. Construction and instanceOf work using this shared prototype value.

(4) (current spec.) bound functions have a “prototype” that is initialized to the value of the target function’s “prototype” property. Construction uses the target “prototype” value, instanceOf uses the bound function’s “prototype” value.

(5) bound functions have a “prototype” property that is initialized to the value of the target function’s “prototype” property. However, both construction and instanceOf use the target function’s “prototype” value.

(6) bound functions have a “prototype” property that is initialized to the value of the target function’s “prototype” property. Both construction and instanceOf use the bound function’s “prototype” value.

(7) Initially the bound function has a “prototype” whose value appears to be unidirectionally shared with the target function and construction and instanceOf use the target function’s “prototype” value. However, if the bound function’s “prototype” is explicitly set to some value, then construction and instanceOf uses the bound function’s “prototype” value. Setting the bound function’s “prototype” property does not change the target function’s “prototype” property.

Option(1) seems the better of the options – existence of bind is motivated by the functionality provided by the Prototype.js library; hence option (1) seems better - that was the originally proposed option, but it had push back (from Waldemar) - currently no way to write a function that cannot be a constructor too - if we allow new on bound functions, we must allow for some delegation to the target function; that is why we need to parameterize [[Construct]] – can make

a distinction that functions written in JS are also constructors, but functions produced by bind are not constructors - does that mean we should allow the prototype property to be deletable? What if someone just deletes the prototype property of every function? - that would be useful from Caja's perspective.

What should toString do when called on the bound function? Indeed, what should be its 'name' property? - should be anonymous – we have several options:

(1) the empty string

(2) a specific string (that we agree to in this call) that labels all bound functions

(3) some mangling of the target's name

(4) a scheme similar to what we use for getter/setter - we could name them "bind " with the target function's name appended

(5) leave it as the target function's name

The 'name' property is practically used in debuggers - option (4) is best.

So, what should toString do on bound functions? - just let the implementations do something useful based on the existing toString specification (15.3.4.2)

Any reason why the 'name' property should not be writable? - leave it as it is.

eval

ES3.1 non-strict cannot statically tell if eval used as variable name refers to the global eval - strict mode currently prohibits assignment to the eval property, and also prohibits indirect eval - we should relax the second restriction - indirect eval should be permitted; and, it should not inherit strictness from its caller - actually, we would like to have a true eval operator - just like other operators, and not as a value - ok, consider the plus operator; if you wanted a plus function you would write one that used the plus operator; can we think of eval along the same lines? - an eval function that is using the eval operator - so, there ould be two magic eval operators, and there would be a strict eval function and a non-strict eval function; the non strict eval function would use the non-strict eval operator unless the string passed-in to eval has the use strict directive - could still cause binding issues - also, need to consider impact on implementations that might already be underway - sure, need to consider implementation burden/impact, but that cannot be the only consideration - in any case, the is headed in a complexity-direction that we don't want - need to consider this some more.

Meeting adjourned.