Ecma/TC39/2015/035
Ecma/GA/2015/067

Minutes of the:	46th meeting of Ecma TC39
in:	Santa Clara, CA, USA
on:	27-29 May 2015

1 Opening, welcome and roll call

1.1 Opening of the meeting (Mr. Neumann)

Mr. Neumann has welcomed the delegates at the Netflix office in Los Gatos, USA.

Companies / organizations in attendance:

Mozilla, Google, Microsoft, IBM, Intel, jQuery, Facebook, Netflix, Indiana University PayPal, Yahoo!

1.2 Introduction of attendees

Jordan Harband (invited expert – Airbnb)

John Neumann – Chair

Allen Wirfs-Brock – Mozilla

Yehuda Katz – jQuery

Adam Klein – Google

Brian Terlson – Microsoft

Sebastian Markbage – Facebook

Jeff Morrison – Facebook

Lee Byron

Dave Herman – Mozilla

Kevin Smith

Brendan Eich

Mark Miller – Google

Jafar Husain – Netflix

Istvan Sebestyen – part time on phone

Sam Tobin-Hochstadt – Indiana University

Daniel Ehrenberg

John McCutchan

Simon Kaegi – IBM

Peter Jensen – Intel

Dan Gohman

Michael Ficarra (invited expert – Shape Security

Waldemar Horwat – Google

Chip Morningstar – PayPal

Eric Ferriauolo – Yahoo!

Stefan Penner

Paul Leathers

Jonathan Turner – Microsoft

Matt Sweeney – Netflix

Miško Hevery – Google

Alex Russell – Google

May 27 2015:

Brian Terlson (BT), Allen Wirfs-Brock (AWB), John Neumann (JN), Jeff Morrison (JM), Sebastian Markbage (SM), Yehuda Katz (YK), Dave Herman (DH), Sam Tobin-Hochstadt (STH), Lee Byron (LB), Kevin Smith (KS), Daniel Ehrenberg (DE), John McCutchan (JM), Dan Gohman (DG), Brendan Eich (BE), Adam Klein (AK), Jordan Harband (JHD), Mark Miller (MM), Michael Ficarra (MF), Waldemar Horwat (WH), Chip Morningstar (CM), Simon Kaegi (SK), Peter Jensen (PJ), Eric Farriauolo (EF), Stefan Penner (SP), Paul Leathers (PL), Jonathan Turner (JT), Matt Sweeney (MS)

May 28 2015:

Brian Terlson (BT), Allen Wirfs-Brock (AWB), John Neumann (JN), Jeff Morrison (JM), Sebastian Markbage (SM), Yehuda Katz (YK), Dave Herman (DH), Sam Tobin-Hochstadt (STH), Kevin Smith (KS), Daniel Ehrenberg (DE), Adam Klein (AK), Jordan Harband (JHD), Jafar Husain (JH), Mark Miller (MM), Michael Ficarra (MF), Chip Morningstar (CM), Simon Kaegi (SK), Peter Jensen (PJ), Eric Farriauolo (EF), Stefan Penner (SP), Paul Leathers (PL), Jonathan Turner (JT), Brendan Eich (BE), Dan Gohman (DG), Miško Hevery (MH), Matt Sweeney (MS), Istvan Sebestyen (IS)

May 29 2015:

Allen Wirfs-Brock (AWB), John Neumann (JN), Jeff Morrison (JM), Sebastian Markbage (SM), Yehuda Katz (YK), Dave Herman (DH), Sam Tobin-Hochstadt (STH), Kevin Smith (KS), Daniel Ehrenberg (DE), Adam Klein (AK), Jordan Harband (JHD), Jafar Husain (JH), Mark Miller (MM), Michael Ficarra (MF), Chip Morningstar (CM), Simon Kaegi (SK), Peter Jensen (PJ), Eric Farriauolo (EF), Stefan Penner (SP), Paul Leathers (PL), Jonathan Turner (JT), Brendan Eich (BE), Dan Gohman (DG), Miško Hevery (MH), Matt Sweeney (MS)

1.3 Host facilities, local logistics

On behalf of Netflix **Jafar Husain** and **Matt Sweeney** welcomed the delegates and explained the logistics.

2 Adoption of the agenda (2015/033-Rev2)

The agenda was approved as posted on the github:

Agenda for the: 46th meeting of Ecma TC39

1. Opening, welcome and roll call
   i. Opening of the meeting (Mr. Neumann)
   ii. Introduction of attendees
   iii. Host facilities, local logistics
2. Adoption of the agenda (TODO: Ref document name)

3. Approval of the minutes from March 2015 (TODO: Ref document name)
4. ECMA-262 6th Edition
    1. Editor's status report
    2. post GA approval PR??
5. ECMA-402 2nd Edition
6. Post ES6 process, roles. and targets
7. ECMA-262 7th Edition and beyond
    1. Module export-from additions - Move to stage 3? Draft Spec (Lee Byron - Facebook)
    2. SIMD.js - Move to stage 2? Docs: Draft Spec, Polyfill, Presentation. (Peter Jensen - Intel, John Mccutchan - Google, Dan Gohman - Mozilla)
    3. function.next meta-property - Move to Stage 2? Docs: Proposal including spec. language (Allen)
    4. Observable Nominal Type Strawman, Polyfill. (Jafar Husain - Netflix, Kevin Smith)
    5. Relaxed semantics for Promise.resolve nominal check (https://esdiscuss.org/topic/subclassing-es6-objects-with-es5-syntax#content-50)
    6. Function.prototype.toString revision (Michael Ficarra)
    7. Decorators (Yehuda Katz)
8. Test 262 Status
    1. Accuracy tests for Math methods? see also
9. Report from the Ecma Secretariat
10. Date and place of the next meeting(s)
    1. July, 28 - 30, 2015 (Redmond, WA - Microsoft)
    2. September 22 - 24, 2015 (Portland, OR - jQuery)
    3. November 17 - 19, 2015 (San Jose - Paypal)
11. Group Work Sessions
    1. Value types (Daniel, many)
    2. Extensible operators and literals (Brendan)
12. Closure

3 Approval of minutes from March 2015 (2015/031)

The minutes were approved without modification.

4 Status of "ES6 Suite" approval at the June 17, 2015 Ecma GA

4.1 RF "Opt out" for ES6 Release Candidate #1

Mr. Sebestyen reported that the "opt-out" for ES6 has ended on April 25, 2015. No-one has requested to "opt-out". So the RF goal is intact.

4.2 RF "Opt out" for ECMA-402 2nd Edition

Mr. Sebestyen reported that the "opt-out" for ECMA-402 2nd Edition has ended on May 26, 2015. No-one has requested to "opt-out". So the RF goal is intact.

4.3 RF "Opt out" for ECMA-404 1st Edition

There is no change on the in 2013 October approved standard. However, that was approved under the RAND Ecma patent policy regime. Since the JSON Syntax originally in ECMA-262 Ed. 5 has been moved from ES6 into ECMA-404 in order to assure the RF status of the entire ES6 Suite TC39 decided to launch an opt out on ECMA-404 immediately, ending on May 26, 2015. **Mr. Sebestyen** reported that the "opt-out" for ECMA-404 has ended on May 256 2015. No-one has requested to "opt-out". So the RF goal is intact.

4.4 ECMA-327 (Compact profile) and ECMA-357 (E4X) matters

Question in the March TC30 meeting: Should they be withdrawn?

ECMA-357 and ES6 will not work together.

TC39 is on the opinion that ECMA-327 and ECMA-357 should be withdrawn.

Mr. Sebestyen has asked the opinion of all Ecma members, in order that final TC39 decision can be made at the current meeting. He reported that no Ecma member has requested not to withdraw the two standards. So TC39 confirmed that the June 17, 2015 GA should withdraw these two standards. The two standards will still be accessible on the Ecma website, but among the withdrawn specifications.

5 ES7 and Test262 Discussions

Most time was spent to progress ES7 related topics.

For details please see Annex 1 in the Technical Notes.

6 Report from the Secretariat

Mr. Sebestyen reported that the GA will also vote if the "experimental" TC39 IPR policy options (RF Patent and Software Copyright Policy) should become "final" and "Ecma-wide" policies.

7 Date and place of the next meeting

I. July, 28 - 30, 2015 (Redmond, WA - Microsoft)

II. September 22 - 24, 2015 (Portland, OR - jQuery)

III. November 17 - 19, 2015 (San Jose - PayPal)

8 Closure

Mr. Neumann thanked the TC39 meeting participants for their hard work. TC39 has reached an important new mail stone by finishing and approving ES6. Many thanks to **Mr. Wirfs-Brock**, the editor of ES6 for his hard work. Also many thanks to **Mr. Waldron**, the Editor of ECMA-402 2nd Edition.

Many thanks for the technical note takers in Annex 1.

Many thanks to the host, **Netflix** for the organization of the meeting and the excellent meeting facilities and dinner. Many thanks in particular to **Mr. Husain** and **Mr. Sweeney**. Many thanks also to **Ecma International** for the social event.

Annex 1

Technical Notes

May 27 2015 Meeting Notes

Brian Terlson (BT), Allen Wirfs-Brock (AWB), John Neumann (JN), Jeff Morrison (JM), Sebastian Markbage (SM), Yehuda Katz (YK), Dave Herman (DH), Sam Tobin-Hochstadt (STH), Lee Byron (LB), Kevin Smith (KS), Daniel Ehrenberg (DE), John McCutchan (JM), Dan Gohman (DG), Brendan Eich (BE), Adam Klein (AK), Jordan Harband (JHD), Mark Miller (MM), Michael Ficarra (MF), Waldemar Horwat (WH), Chip Morningstar (CM), Simon Kaegi (SK), Peter Jensen (PJ), Eric Farriauolo (EF), Stefan Penner (SP), Paul Leathers (PL), Jonathan Turner (JT), Matt Sweeney

Agenda approval.

Consensus / Resolution

So say we all.

Approve minutes from previous meeting

Consensus / Resolution

So say we all.

ES6 Updates (AWB presenting)

AWB: Doing a few cleanups and tweaks to final draft to make it ready to publish. Have final PDF. AWB: Working on updating Jason's tool to produce HTML version. AWB: Deck on significant bug fixes (Share slides) AWB: Bug #1: super prop assignment can silently overwrite non-writable properties (now fixed in spec) AWB: Bug #2: Unintended for-in eval order change (now fixed in spec) AWB: Bug #3: GeneratorParameter grammar parameter now eliminated (now fixed in spec -- replaced with early error rules) WH: Wants a copy of the modified grammar to re-verify the changes in his ES grammar validator BE: can you put your Common Lisp based validator on Github? WH: I want to. Stay tuned. [May need to get rid of confusing irrelevant extra fluff that accumulated over the years.] AWB: Spec going to ECMA in 2-3 weeks, will likely be approved by general assembly AWB: Then going to ISO, changes mostly handled by ECMA secretariat. They will find some real spec issues. But it takes a year, so it'll be confusing because we'll come out with ECMAScript 2016 by then... AWB: Name is changed to "ECMAScript 2015 Language Specification"

AWB: ES6 is done. AWB: Reminder that a year from now we will be at the same point for ES2016. At end of Jan next year we have to have a complete ES2016. YK: We expect it to be light... AWB: Yes, fundamental issue is what's in the spec and when it can come in. AWB: We don't have a lot of breathing room to get stuff ready for inclusion. AWB: Not going to be editor anymore. Who will be editor? AWB: Editor is a full-time job.

[debate about form of the standard document] WH: The format of the document plays only a minor role in what an editor does. The other duties of an editor are still a full-time job. AWB, BE: Not feasible to switch away from Word for 2016. BT: "Ecmarkup" is being used for 4 or so ES7 specs, we could rapidly switch over if we chose tohttp://bterlson.github.io/ecmarkup/ BE: can we generate Word that's good enough for Ecma and ISO from Ecmarkup? BE: easier to recruit new editor if we have better tooling and github PR-based helpers -- really want that modern workflow, it'll help productivity while not eliminating single-final editor role YK, BT: Doing editing as a group would allow more people to get involved and reduce the amount of work that editors would have to do, allowing more people to help AWB: Actual document editing took <20% of editor time. The 80% was spent on integration of proposals across the entire spec.

AWB: Jason no longer wants to maintain HTML spec. MF: If we move to EMU, how will links persist across revisions? BT: EMU requires assigning unique ID to each section

AWB: I'm leaving Mozilla too, but happy to help the new editor out.

?: What needs to live on the ECMA GitHub? WH, AWB: Any contributions need to be in some format that ECMA can archive, for legal, librarian, and historical needs — imagine someone needing to track down the history of some contributions 15 years from now. For individual documents (pdfs, etc.) the simplest way is to send them via the ECMA reflector. ECMA keeps an archive of those forever. For ongoing things use an ECMA-sanctioned repository such as ECMA's GitHub.

Module export-from additions (LB presenting)

LB: This proposal fills in a gap and makes things more consistent. Currently, it's unnecessarily verbose to put together a module. YK: "export v from 'mod';" is confusing; does it re-export "default", or export a name "v"? YK: Existing syntax for this seems clearer: "export {default as v} from 'mod';" LB: How about we keep "export default from 'mod';' separately? Everyone seems happy with that.

Resolution: move to stage 2 with two accepted forms: "export default from 'mod';" and "export * as ns from 'mod'"

SIMD.js Stage 2 (PJ, JM presenting)

PJ: (shows demo of graphics performance with and without SIMD.) BT: (shows similar demo.) MM: If you do a typeof, what do you get? DE: Will talk about typeof semantics after presentation. WH: Why is max only on float? That seems gratuitously inconsistent. max is just as useful on integers, even if you can roll it on your own. MM: From a POLA perspective, it seems like there should be max on integers as well. JM: Rolling your own max is simple. BT: Talked about SIMD and ES6 at recent talk, and most questions where about SIMD. BE: I've gotten the opposite reaction. BT: Developers were interested in games. SK: Why 128 bits DG: 128bits is a natural size for many operations and was the largest common size across SIMD architectures we considered CM: Makes sense for performance use cases, but it seems weird to have the implemention detial bubble up. JM: Larger can be written on top of 128. MM: Minor API issue: the name swizzle historically used simillarly to marshal and serialize JM: In graphics programming the name swizzle is the right choice WH: How does endianness become visible? DG: First lane is always at offset 0 of the typed array, second follows first lane, etc. The endianness within a lane is implementation-dependent. (discussion about little endian/big endian) DG: Contents of the lane are byte order dependent on platform WH: SIMD reinterpret cast will do different things on different systems? DG: Yes. AWB: You see the same thing with TypedArray. WH: You have load, load1, load2, load3, but no load6, for instance? DG: You could imagine what a load6 might be, but not particularly useful. BE: It could be added in the future. MM: What does xmmintrin.h mean? WH: XMM (a register type in the x86 architecture) intrinsics R?: It's a header file that defines an interface that's widely used for 128 operations. In emscripten we have an emulation of it in terms of SIMD.js. DE: Intention is to represent with value types with wrappers. Typeof returns a string representing the type for the value object. MM: This is a proposed system defined value types. How does this work with user-defined value types? BE: If we're going to integrate with main spec, we might want to roll into user-defined value types. DE: What if we move to stage 2 and we can still integrate it into user-define value types. AWB: It would be good to not introduce 6 new system value types. WH: Interpreting <, <=, >, >= as always being string comparisons in current spec text is hostile to value types and other numeric comparisons. (we don't want 200 < 9). DE: With less-than, I just added a note that it compares it as strings. BE: We should work on the operators which should be overridable. Don't define future-hostile semantics -- those that we'll need to change later. DE: Like to make operators which are eventually overloadable, throw now. WH: The spec has the Int64x2 case but can't find its definiton. Curious what int64's turn into when extracted. How does that work? DE: Not defined. There is no way defined to load and store them yet. WH: Defined multiplication by element-wise ECMAscript multiplication with its rounding of intermediate results > 53 bits. This is not what SIMD implementation do — they all do strict modular integer arithmetic. DE: I've added a note asking if this is the direction we want to go. The polyfill uses Math.imul which is the right solution. (DE will change the spec.) JHD: Are properties of SIMD supposed to be nonwritable, nonconfigurable? MM: The general style we've agreed upon is that primordial properties are either configurable or writable. This is important for initialization of realms, to make it seem like a different kind of realm. DG: I believe that Firefox JIT can handle that. JM: Should we think about standard modules? DH: While the module imports are not writable, the module table is. Realm initialization is just as possible with modules as with configurable/writable properties. You just have to update the table. DH: It's fine to not have a blocking dependency on modules. We don't have convensions for standard modules yet. I think it's fine to put SIMD on the global object. JM: It seems like someone needs to be the first to add a standard module. DH: It's OK if we wait for userland module convensions to emerge. MM: We can let this proceed to stage 2 without having to specify in terms of value types or standard modules, with the idea that we will eventually get there. DG: If we want to get this done in 2015, can we agree to have a SIMD global? DH: There's nothing wrong with having a global named SIMD and a standard module for the same thing. MM: With modules standard and loaders not standard, is it the case that it doesn't give TC39 the option of specifying new things in terms of modules? STH: We want to ship SIMD on a timeline which is not constrained by the loader spec. MM: I'm convinced, I was just taken by surprise by the implication (that we couldn't add system modules). AWB: With globals, each realm now has to have all of these duplications, depending on optimizations. MM: SES has to make sure that none of the primordials expose mutable state. They have to freeze them, and the only way to do that is to walk eagerly. DG: What about the large number of SVG bindings? MM: The SVG bindings are provided through a membrane. DE: Can we provide SIMD through a membrane? AWB: Not if it's in the ES spec. DH: Can't SES just delete SIMD? (room laughs) MM: Clarifies the cost in question: traversing the objects and freezing everything. STH: So generally, adding "n" functions will present this problem for you. MM: Yes, when "n" is large enough. The way to avoid this overhead is to have a platform-provided way of creating a new realm that looks like this realm but is frozen. DH: (Notes that such a capability would have benefits beyond just security.) CM: (To MM) Are you interested in creating a realm with specific constraints, or general realm initialization? MM: The realm API is for creating a new realm according to the wishes of the creator of the realm. What I'm interested in is not the Realm API itself, but have a way to ask the platform for an SES realm. MM: "SESsiness" BE: "Sessility" (real word)

Resolution: move to stage 2

R?: If you put a NaN into a TypedArray, the spec requries it to be a non-signaling NaN. Why? AWB: It was copied out of the WebIDL spec. There are only two references, we could remove those. BE: spec link:http://people.mozilla.org/~jorendorff/es6-draft.html#sec-setvalueinbuffer BE: proposal is to remove the "if value is NaN..." language and let different (bit-patterns observable via typed array views as well as SIMD) NaNs be stored WH: Any thought to UInt SIMD types? uint8 and uint16 are much more useful for pixel values than int8 and int16. DE: Use a different set of primitives. +, -, * don't care about signed/unsigned. Use differently named functions to load them or do lane-wise comparisons. WH: That's awkward, particularly for uint8. Also, some operations can't easily take different names: <, >=, value-to-string conversions, etc. If I have a pixel color value of 255, I don't want it to print as -1. DE: Using < on SIMD values makes no sense. What should it do? WH: The obvious dictionary order comparison of the numeric values. DE: People won't use it. AWB: Oh yes they will. If it can be done, people will do it. BE: group resolved already to avoid shipping non-starters...

Generator function.next Meta Property (AWB presenting)

MM: If you allowed function.next within an arrow function inside of a generator, what would it mean? AWB: It would just mean the same thing that it does outside of the arrow function. DH: What are the use cases other than getting the first one? AWB: (References example in repo) DH: But you could do the subsequent capturings with a local variable. AWB: It allows you to not special-case the first one. DH: This feels like has a little of the ".current" api from C# iterators. Not sure if it matters. Does this create any GC pressure? AWB: Don't think so. It's specced so that it's cleared out when the next value comes in. DH: Think you might just need "function.first" or something, but your (AWB) argument for not special casing is good. DH: Think it's odd to call it next when you mean previous. BE: writes "function* gen() [first] {}" as an option DH: No... DH: Could be "function.yield" MM: Writes:

```
function* addr() {
    try {
        yield;
} finally {
   return function.next;
}
}
let tally = addr();
tally.next(17);
tally.return(5);
```

AWB: Returns the last value passed to next: 17 MM: Is it the value coming in to "return" and "throw" or just "next"? AWB: Just "next". DH: That's why you like "next", because it's only the value passed in with "next". MM: We're agreed that this returns 17. If you resume with "return" then "function.next" retains the value of the previous resumuption. AWB: Yes. I entertained this. WH: Any weird interactions with yield*? KS: yield* primes its generator with the value undefined. If you write using function.next, you get problems because it will be primed with an unwanted "undefined" DH: Could "yield *" prime the generator with "function.next"? (discussion about incompatibility with current spec) DH: The code using yield * is probably pretty low, so we could probably do an errata. DH: The question is whether the community transpiling and using function* can MM: There is a capability leak concern here, where it's passing to the subgenerator something that's not implied by the yield * itself. DH: Is that circular? MM: But it's passing something from before when yield * is evaluated. DH: I'm beginning to think that this is the wrong path (passing in function.next via yield *). AWB: You could create some kind of wrapper if you wanted to pass in the first value to the subgenerator. MM: Libraries could do this.

```
yield * wrap(g, function.next);
```

MM: "wrap" returns an iterator which wraps the generator which primes the subgenerator with the supplied value. ?: Collect usage data of "wrap" to see if we should change yield*. WH: By the time we get the usage data, it will be too late to fix yield*. AWB: We'd end up with a second form: yield** (general agreement on current yield * behavior) AWB: Proposing that we move this to stage 2. DH: Don't think that "next" is the right name. YK: Not sure we can advance a small syntactic thing to stage 2 if we're not sure about the final syntax because of transpilers. MM: Is it worth taking a few minutes to brainstorm on a name?

List of _ options for function._: current, previous, last, next, nextarg, pre, step, lastNext, n, in, lastYield, lastInput, yielding, input, genin, previousValue, gin, gr, now, sent, generation DH: Want some indication that it's the resumption value. WH: If I were seeing function.current for the first time, I'd think that it was arguments.callee. BE: What about "yield.something" AWB, MM: (agree that it would work) DH: There's human ambiguity. Adding parens around "yield" should not change the meaning. JHD: what do we call the value passed into Generator#next? we should name that and call this the same thing LB: Propose "sent" MM: Does anyone object to "sent" CM: Would that be "dissent"? (general agreement on "sent")

Resolution: advance to stage 2, update proposal to use "sent"

Test 262 Update (BT presenting)

BT: Test262 is super active. Much es6 coverage now. Strict clean.

DG speaks to Math method test result variation, proposes fdlibm (+/- 1 ulp accuracy for most methods including sin) [debate about whether to go for precision-based limits or mandate one particular implementation for reproducibilty]

WH: The state of the art advances. Had we mandated one implementation for reproducibility in ES1, we'd now be stuck with numeric functions that are inferior in both accuracy and speed than the current state of the art. WH: For some functions, such as sqrt, it is practical to require correctly rounded exact results. For others there are no known efficient implementations yet without a bit of tolerance.

DG: In the context that math libraries have the power in ECMAScript to deliver high-quality results at a variety of performance/precision tradeoffs and can evolve over time, the committee has three main approaches for the builtin math functions: - Stay with the status quo. Math functions in the spec are entirely ungoverened. This has been the reality for a long time and it's not necessarily problematic. - Specify particular implementations for each function, possibly including algorithms from fdlibm, crlibm, or other places. The main advantage of this would be that floating point in the spec bit-for-bit reproducible, which is an interesting property. - Empirically discover maximum error bounds for existing ECMAScript implementations and specify something around that. What do you prefer? WH: Efficient and precise standard math libraries do a large amount of bit-banging. They can't be implemented efficiently in userland ECMAScript code using just +, -, /, etc. They need additional primitives. DG: We can do everything we need with the existing math primitives in ECMAScript. It'd be nice to add a few things, like reinterpret cast, but we can do that with typed arrays if needed.

May 28 2015 Meeting Notes

Brian Terlson (BT), Allen Wirfs-Brock (AWB), John Neumann (JN), Jeff Morrison (JM), Sebastian Markbage (SM), Yehuda Katz (YK), Dave Herman (DH), Sam Tobin-Hochstadt (STH), Kevin Smith (KS), Daniel Ehrenberg (DE), Adam Klein (AK), Jordan Harband (JHD), Jafar Husain (JH), Mark Miller (MM), Michael Ficarra (MF), Chip Morningstar (CM), Simon Kaegi (SK), Peter Jensen (PJ), Eric Farriauolo (EF), Stefan Penner (SP), Paul Leathers (PL), Jonathan Turner (JT), Brendan Eich (BE), Dan Gohman (DG), Miško Hevery (MH, Matt Sweeney

Ecma Update (Istvan)

Do we still withdraw E4X and etc?

Consensus / Resolution

Withdraw.

Function.prototype.toString revision (Michael Ficarra presenting)

MF: presenting https://github.com/michaelficarra/Function-prototype-toString-revision MF: Let's review the spec for toString YK: How does it deal with default arguments? MF: The spec currently doesn't say anything about parameters MM: The requirement is that the returned function string has the same behavior when called YK: Worried about what implementations do BE: All the known implementations do source recovery

MF: presenting...

Open issues: * function name property * definition on MethodDefinition and GeneratorMethod is left undefined because how would you define this, new.target and super--eval'ing it couldn't let them bind properly. However, it's important to leave things open to implementations to just hold the source code as they do right now. * It would be better if the spec talked about the result of evaluating [[Call]], rather than the internal steps of [[Call]], so it doesn't overspecify. This text seems to use the word 'indistinguishable' in a way that's specific to that particular paragraph; maybe it should leave the word 'indistinguishable' for what it means in the rest of the spec and use another word here or refer explicitly to the outcome of the evaluation. * Add an optional FunctionBody after '[native code]' for the native case (suggested by MM) * Require (in chapter 16) that implementations generate a SyntaxError for '[native code]'

MF: Discussing the "Else, if func has an [[ECMAScriptCode]] internal slot" clause of the new spec text AWB: The statements "func was defined using ECMAScript code" and "has an [[ECMAScriptCode]] internal slot" have the same meaning MF: How do we handle things that were not created using the Function constructor or written in ECMAScript code? What if a host-provided exotic object has an [[ECMAScriptCode]] internal slot? AWB: Then it is, by definition, created by the Function constructor. MM: The first if should just be "if func has an [[ECMAScriptCode]] internal slot and is callable..." MF: But then you need a way to distinguish the case where the host environment provides some object that has an [[ECMAScriptCode]] internal slot? BE: You don't need to, it can be the "else" clause, along with an Assertion

YK: Back to the goals: what rubric is being used to decide which cases should be defined to throw a SyntaxError when toStringed? MM: The goal is to avoid the case where toString doesn't generate an error, but evaling the result doesn't produce equivalent behavior to calling the function. YK: An alternative design is to provide source recovery. In that case, the design should be to return whatever the user typed. That obviously violates the evaluatable requirement.

MF: The proposed change also defines the 'function() { [native code] }' string that must be returned. YK: Does that mean that the PS4 returns that string for all functions? Or is it non-conforming. MM: This definition means we're committing to '[native code]' being a syntax error. CM: Seems brittle, could we prefix the string with something instead? BE: Web code depends on that specific "[native code]" string -- de-facto standard MF: And the point is to be easy to parse it (paired braces, brackets, etc)

MF: Proposing moving to stage 1. YK: Don't think we should move to stage 1 as this is the first time the committee has seen it. BE: This has been discussed on es-discuss, there are open bugs AWB: The problem goes back to ES6 discussions STH: Yehuda clearly disagrees with the underlying goals, which is why his complaint should be relevant to whether we move to stage 1 BE: That is a very valid procedural objection; separately, I don't feel like source recovery is doable

AWB: If the interest is in providing moving functions between address spaces, then maybe we should do it somewhere other than Function.prototype.toString, which doesn't work for all cases right now anyway. MM: Why not make toString do that job? AWB: Because it's at odds with source recovery. YK: As we add more things like 'super' to the language, there are going to be more and more cases where toString is not going to provide portability. MM: The cases I care most about are FunctionExpression, ClassExpression, GeneratorExpression; am open to producing guaranteed SyntaxErrors for things like GeneratorMethod. CM: But that's still going to put pressure to try to generate things that work for round-tripping behavior and are further from the source recovery usecase MM: The de facto standard in ES6 was already to generate an error at eval time for methods, I'm only trying to make that explicit in the spec YK: I don't want the addition of 'super' in a method to change the output of a test suite from the source text to a syntax error MM: What if we inject a guaranteed syntax error [like "[native code]"] in to the toString? Or alternatively guaranteed that method syntax is a syntax error in an expression. YK: I would be open to that if it's easy to strip out. AWB: This whole thing bothers me, toString seems like a debugging/recovery thing, even if it's been historically used with eval. I don't like that, in a debugging session, toString will not return the source that I typed. CM: It really sounds like these goals are at odds

(function() { /* hello / *}).toString() "function () { / hello / }" (function() { / hello / }).toString() "function () { / hello / }" (function(foo) { / hello / }).toString() "function (foo) { /* hello */ }"

JHD: `Function.prototype.toString.call({foo() { return foo }}.foo)` in firefox returns "function foo() { return foo }" but in v8/chrome returns "foo() { return foo }" which are not functionally equivalent JHD: also,`Function.prototype.toString.call(Object.getOwnPropertyDescriptor({get a() { return 3 }}, 'a').get)` in FF returns "function () { return 3 }" but v8/chrome returns "
YK: Can we agree that source recovery should be a goal of this proposal? MF: It wasn't originally my goal in putting together this presentation. MM: I agree that implementations seem to be aiming for that goal, though it's not a goal of mine. SP: Another use case is detecting what features a function is using (say, 'super') MM: The injected syntax error solution supports that use case

MM: I only care about a certain set of cases YK: What about arrow functions?

[...lunchtime discussion...]

YK: There are a set of things that do not have unserializable state (FunctionExpression, ClassExpression, GeneratorFunctionExpression, and some declaration forms of those). There a whole other set of forms (arrow functions, concise methods) that may have unserializable state. YK: The proposal is to add a new predicate (strawman: "Reflect.isPortable") that can be used to determine which of these forms a given function falls into. YK: The predicate could even return more information, such as a list of free variables in the function. YK: Given the above predicate, I am satisfied that moving MF's toString proposal to stage 1 will satisfy both the source recovery use cases and MM's portability uses cases.

Resolution

Move MF's Function.prototype.toString proposal to stage 1, with a dependency on the Reflect.isPortable predicate (which is effectively a stage 0 proposal) and an added goal of supporting the source recovery use cases.

Decorators (Yehuda Katz, Jonathan Turner) (Need slides)

YK: Used to be against decorating function decls which hoist. Also, having decorators on exprs and decls was bad. However, I've come to peace with hoisting the execution of the decorator expression. Seems plausible. AWB: Function declarations are created before there's any environment. If you introduce something that can execute at that time it changes all of the semantics. YK: This is hard. We need to think hard. What Allen said was true. DH: We shouldn't rathole for too long on what the answer is because we don't have one yet. There is no obvious right answer. Easiest not to support this at all, but this is a mistake - people want to use function declarations, and if decorators don't work on function declarations, people won't use function decls. We have a few tools - 1 is imports. YK: The semantics will be rough no matter what. We can prototype and see if in real world code this is a problem. JM: Prototyping is great but it doesn't find footguns well because it's a small module. JT: We can put it in typescript. We get good feedback. If it's just in experimental, it's possible to remove later. AWB: Clarifying module initialization: when a module is instantiated, one of the first thing that happens before it finds any imports, it instantiates any function declarations. DH: You could observe that a binding isn't initialized yet. Could introduce TDZ? [Problem: If you make a let binding and refer to that in a decorator parameter, you will hit a TDZ] AWB: Creates opportunity for fatal circularities that weren't a problem before. YK: We should get our transipler friends to try it out and see. DH: There needs to be work on this... need a plausible design. YK: I agree. AWB: Would it be an acceptable semantics if we can't figure it out we say that decorated decls don't hoist? DH: Sure. There will be inconsistencies any way. YK: People depend on hoisting. What it would mean is that occasionally their code wouldn't work and they wouldn't use decorators. Maybe that's ok. AWB: Maybe that's ok! Could be same as classes. Could have good error message.

[Presents on parameter decorators] YK: Parameter decorators work on Parameter descriptors. JM: Couldn't I just decorate the entire function and use the reflective API to touch the parameters? JT: Yes. AWB: What do we statically know and what is knowable dynamically. Do we statically know there is a formal parameter named "f"? YK: You can't change type or name of the parameter descriptor. The main thing is metadata. Possibly wrapping the default expression. AWB: Root of the question is about the fact that a function definition starting with name and parameers through the body is something that is statically analyzed as a unit independent of evaluation. This injects in essence evaluation semantics into the middle of the static analysis... YK: I Wouldn't think of it that way. I would think about it that you create a function in the first step and then you go through the formal parameters and get their descriptors and you could modify them, but the modifications you can make are limited to things we can accept. AWB: Ok, different question. Last question assumed the decorator was evaluated at func definition time. But another way is that they are evaluated on each invocation. YK: One goal of decorators is that they don't introduce call-time overhead. BE: Need to clearly define the evaluation model. YK: Started with the reflection API as I'd like to desugar to two reflection APIs. AWB: Here's the trap: If I wanted to write a static compiler for ECMAScript, how does this impact those uses? YK: The simplest thing this is doing is adding metadata which seems equivalent to adding to weakmap. AWB: If evaluation happens after the class... if it doesn't require anything at compilation time... YK: I think that's a sticky question.. AWB: It looks like it's inside the function. YK: It's outside. AWB: What's the scope of param decorators? Are params in scope like they are with defaults. YK: All decorators, no matter where they are placed inside the class body, have the scope of the outer scope. It's possibly confusing. AWB: I think it's totally confusing. BE: It seems confusing. BE: ARB sees wanting decorators on static constructs like modules and other decls. There is a tension between static and dynamic. YK: In practice the case that Allen mentions won't happen. YK: Originally thought that param decorators didn't fit into this, but everyone wants this so... AWB: What about destructuring? JT: (Answering what is done in TS) Param decorators have outer scope. AWB: Violates rule we had in ES6. YK: Need to work out the semantics. AWB: What about patterns [destructuring bind]? JT: I think TS doesn't allow now. DE: Must be hard to even reflect on destructuring bind AWB: It would be bad if the decorators worked only some of the time, and then not on destructuring bind. Good to capture all the hard problems. [Discussion between AWB/YK Regarding when evaluation occurs for various decorator constructs] JT: Imagine we had a reflect.decorate API that was capable of composing for you. The other way would be to form a decorator pipeline of sorts. YK: Completely replacing a class with a new class seems bad. JT: We want to create a reflect API for self-hosting decorators with a step-by-step thing. [More presentation and discussion missing from here]

Observable Nominal Type (JH, KS)

JH: Presenting slides TODO(JH): add link Issues with async generator proposal [expanding on bits where slides are terse]:

- General agreement that async function* should return an "async iterator" instead of an Observable, as observabe's push model is not necessarily asynchronous, example being sync DOM events

Questions on Array.prototype[Symbol.observer] slide: DH: Why the check for falsiness of iterResult? JH: Just being safe, agree that if |generator| is a real generator it's not possible for iterResult to be falsy. MM: Why are you calling generator.return()? That's normally meant for early exit. JH: In this case the generator is being used as a sink, rather than a source....no, sorry, the slide is wrong [live coding].

Questions on WebSocket slide MF: Why arrow functions? JH: No particular reason, other than those that refer to 'this'.

JH: [...continues...]

SP: What happens when errors occur? Error propogation? JH: When an error occurs in a Promise, the Promise is "dead". That's not the case with Observables, since other observers could still be added by subscribing. But an error signals the end of a single subscription. Every observation ends with either "done" or "error". SP: I think that subscribe() seems like it conceptually should return a Promise, with the only callback passed into subscribe() is "next". That would make this compose better with other Promise code. JH: That's exactly what Observable.prototype.forEach does. But there you don't get the subscription back, so you can't unsubscribe. Unless you have cancelable Promises. MM: Alternatively you could return a pair of [subscription, promise] from subscribe(). But that has the problem that a single subscriber could end the whole observation. SP: OK, now I'm seeing that this is the same issue as cancelable promises.

side bar: [discussion among YK and MM about having then() return a subscription, and why we didn't do that] MM: You could have a lower-level operation than then() that returns a pair of [promise, subscription] where unsubscribing only cancels the particular callback passed in, not the whole promise. And you could have async functions make use of this, which makes it not so bad that you have a pair returned. This might provide an answer for cancelable promises. [TODO(YK, MM): More detail here if you want it pulled out of this presentation]

...back to Observables...

MM: Does calling unsubscribe() cause return() to be called on the argument to subscribe()? JH: No, I don't think so...KS? KS: Yes, a well-behaved Observable should call return(). CM: I don't think that makes sense, return() should only be called when the Observable's stream is complete. MM: Doesn't that break the compositional cleanup semantics? KS: The way I designed the polyfill was more in line with that thinking [that return should be called on unsubscribe], will sync up with JH to sort that out.

YK: I think SP was getting at this: we shouild make sure that we [learn the lessons from Promises] and have error propagation work well.

DH: Comments on Event Composition slide. Trying to describe Hot/Cold language. For mouse moves, it seems like once you have no more subscribers, you want to stop receiving mouse events. DH: trying to understand hot vs cold observables: mousemove is an example where once you reach zero subscribers there's no point in continuing to receive events, so the data source cancels JH: Yes, that's "hot" DH: whereas a cold one might be like a network fetch where when the subscriber count reaches zero that doesn't mean you won't have new subscribers and you don't necessarily want to cancel the underlying request DH: but the decision to be hot vs cold is at the data source, and Observable combinators are about subscription, so one set of combinators works for both hot and cold? JH: well yes but there are plenty of combinators and inevitably some only make sense for hot or cold, and it's IMO more reasonable just to have one set of combinators and have some that simply don't do anything reasonable when called on the wrong type of data source; so that's a leaky abstraction but more practical

MM: Proposes Observable.prototype.then(). KS: Considered that in the polyfill, but ran into possible other things that .then() should do. Will continue consideration.

JH: Continuing Event Composition use cases... MF: These new methods, are you planning to put them on Observable.prototype? Won't that cause problems if people start monkeypatching? JH: Yes, something to be considered.

YK: Promises got fast-tracked because of use-cases in ES6 and the DOM. Have you gotten a lot of feedback from DOM folks that want this? I'd imagine that there would be people chomping at the bit to use Observables for events. JH: Have only heard a bit from DOM, regarding filesystem APIs. Not a lot of feedback from that side. YK: In the meantime, I encourage JH to work on "Observables A+", with a test suite MM: With the goal of working with other Observable libraries and getting them all on the same page JH: I've been in discussion with those libraries YK: You don't even have to write the reference library, but it would help to have lots of examples (IndexedDB, other web APIs) showing the value, as was done for Promises.

JH: I think it would be nice to explain DOM events in terms of Observables; at the least we should be able to adapt them.

MM: Naming concern: this name seems close to "Object.observe". If it didn't exist, "Observable" would be the right name. DE: We could defer this question until one of the two proposals makes its way further along. JH: Could Object.observe be in terms of Observables? MM: Would Object.observe folks object to that? AK: I'm probably the best person to speak to that, and it seems like a reasonable thing for Object.observe to be in terms of Observable JH: Also, ideally "subscribe" would be called "observe" CM: I think "subscribe" actually has something to recommend it -- gives rise to Subscription as name for thing you get back (Observation doesn't work, singular) AK: No one is currently working on pushing Object.observe to stage 3, but I'd be happy to have contributions or feedback. BE: There are objections to it; some people think it shouldn't be there (see Nov 2014 meeting notes) JH: Including me DE: Let's say neither Object.observe nor Observable can get to stage 3 until we get some resolution on the naming conflict [General agreement, moving on]

[lots of discussion about hazards of sync Observables; Observable.prototype.subscribe() only actually subscribes at the end of the turn to avoid one such hazard (see slide)]

MM: Basically, in Promises, there's a guarantee that the callback is called from a clean stack. For Observables, the only guarantee is that the callback is not called from the callback provider's stack. Which is still a pretty good guarantee. JH: Right. We put the burden on Observables that, if they act synchronously, they be careful that they don't depend on state that might change while they act. MH: We [Angular] can confirm that this is a good tradeoff. JH: The motivation for not forcing next() to be scheduled in a new job is to be maximally efficient.

MM: Something I've expressed before, and I'll reiterate, is that I worry about all this new syntax, and would like to see some way to use composition to avoid adding new syntax for each combination of these things. DH: I think the exploration is great, but I also have concerns about proliferating syntax.

Resolution

Move to stage 1, keeping in mind DOM events especially

May 29 2015 Meeting Notes

Allen Wirfs-Brock (AWB), John Neumann (JN), Jeff Morrison (JM), Sebastian Markbage (SM), Yehuda Katz (YK), Dave Herman (DH), Sam Tobin-Hochstadt (STH), Kevin Smith (KS), Daniel Ehrenberg (DE), Adam Klein (AK), Jordan Harband (JHD), Jafar Husain (JH), Mark Miller (MM), Michael Ficarra (MF), Chip Morningstar (CM), Simon Kaegi (SK), Peter Jensen (PJ), Eric Farriauolo (EF), Stefan Penner (SP), Paul Leathers (PL), Jonathan Turner (JT), Brendan Eich (BE), Dan Gohman (DG), Miško Hevery (MH, Matt Sweeney

Relaxed semantics for Promise.resolve nominal check (MM)

MM: presenting

```
Promise.resolve(arb1).then(arb2, arb3);
```

MM: The invariant that we are trying to maintain is that in a realm where the primodials are frozen and arb1, arb2, and arb3 are from an untrusted party, then any code associated with those objects will be executed in a later turn. Since promises are not frozen, the invariant can be broken if "then" is overridden on the instance. The invariants can be maintained by a subclass of Promise.

```
DefensiblePromise.resolve(arb1).then(arb2, arb3);
```

The other way that the invariant was broken was with Promise.resolve. Once we added the newTarget parameter to the Reflect.construct method, that meant that someone could invoke the Promise constructor with an arbitrary newTarget.

AWB: That could be checked in the Promise constructor code. The constructor could traverse the prototype chain of the constructor. MM: Because we have the mutability issue we have to protect the invariants in userland anyway, so I like the proposal from C. Scott Ananian. Just perform a Get on the "constructor" property of the argument supplied to "resolve". MM: Do we have species on Promise AWB: Yes MM: Don't think @@species buys you anything here AWB: But there's a consistency AK: This is a breaking change for shipping browsers. YK: I would be suprised if there are programs which rely on this edge case. MM: We should take this into account. Even if there's code subclassing promises, they would probably not be affected by this change. I would like AWB's opinion on whether we use @@species or constructor. AWB: NewPromiseCapability might use @@species anyway. KS: Can we confirm? AWB: No, it doesn't use @@species. MM: In that case I say we use "constructor". AWB: This is a class-side method, @@species is really for instance chaining. MM:

```
FooCancellable.resolve(arb1).then(arb2).then(arb3);
```

@@species of FooCancellable is Cancellable. Using "constructor", the first then is called on a FooCancellable and the second is called on a Cancellable. That looks correct. AK: We'll have to look and see if this change breaks anything. (Not asking to postpone.) SP: Chrome canary is already broken here: class Foo extends Promise {} Foo.resolve(Promise.resolve()).constructor !== Foo;

Resolution: Change Promise.resolve in ES6 specification to use "constructor" property.

Operator overloading breakout

Slides: http://www.slideshare.net/BrendanEich/extensible-operators-and-literals-for-javascript

DE: Why not use an implicitly named, lexically scoped object for literals (literalSuffixTable)? No staging, just runtime lookup. BE: Don't overload ===, instanceof, in DG: Most operators could be useful for SIMD, except == < <= is probably not a good idea since it'll return a SIMD vector which is truthy BE: Strict equality is still via a structural recursive strict equality check not overloadable, or do we want to change that? BE: Multimethods for dyadic operaors, not double dispatch; see Christian Hansen's work and Cecil SM: Maybe mangle the name for literals somehow else? [Discussion about how to handle suffixes with module imports] BE: The hope is that the spec just has to define operators and literals in a general way. We can have an intermediate step which is value types.

Value types breakout

DE reviews Niko Matsakis's proposal https://github.com/nikomatsakis/typed-objects-explainer/blob/master/valuetypes.md

- (I missed first implicit bit that DE identified -- /be)

- ValueType per-realm registry can only grow, never shrink -- is this ok?
- what if a value from another realm lacks a registry entry for its type? type error or implicit registry (MM objected?) SM and DE discussion of trade-offs imposed by registry key as defined by NM SM concerned about prototype sharing among several value types (immutable array and Array) DE would rather stick with NM's proposal and leave out prototype-sharing and other such features

DE raises intermediate value representation problem.

```
  let Point = ValueType(Symbol('Point'), {x: Float32, y: Float32});
  let p = Point(1, 2);
  // is p.x a Float32 or a JS number?
  assert Float32(0) !== 0;
```

Int64 hard case vs. number as well.

```
x = Float32.[[Cast]](1);
y = Float32.[[Cast]](2);
```

DG: how about Complex? DE: `3+2i` is a "literal" that can be partially evaluated by smart implementations; the `2i` uses literal suffix `i` to make imaginary-2, and `+` operator does rest. In general if number is the intermediate value type, lose precision when demoting (from 64-bits to number) and performance when promoting. Lose-lose!

Could add `Float32.add(a, b)` and so forth -- and these could be operator multimethods -- to help people avoid promotion to number from 32-bit value types
DE: Also thinking about [[Serialize]] and [[Deserialize]] internal methods (maybe Symbol.serialize and Symbol.deserialize one day) for persistence and structured clone
SM: thinking of separate faster-GC heap for value types since acyclic

DE: thinking about discriminated unions as well, which is why symbols might want to be embedded in value types. SM: serialization raises question how this value types thinking relates to typed objects BE: typed objects wanted for their reference identity, heap allocation, mutability

DE: mentions Swift inout handling of structs: `p.x = 1` => `p = p.replaceX(1)` updates whole struct

Fresh realms breakout

MF: Fresh Realms

Programmer want guarantees about how their program will run without worry about what past scripts have done

- in particular referring to scripts like prototype that monkey patch

original proposal....

global-f-global | var a=0; function f() { "realm"; a --> refers to lexical scope }

new proposal is module based -- want to declare dependencies should be run in a "fresh" realm

discusssion around how modules might construct graphs and a fine critique of the npm approach. discussion around the process for splitting an existing module in two in the face of a "fresh" realm and looking at the problems associated with having those newly split modules sharing state

DH: Belief is that the dynamic api is sufficient and we need experience before creating declarative syntax DH: An approach using manifests like System.js to construct the appropriate realm graph

JHD: Want the abilty to run a module in a fresh global especially in the context of using shims

MM: Fine but we should go ahead in parallel e.g. keep the discussion going as we gain experience MM: SES Provides defensability but not defense. Enables use of multiple co-operative realms while protedting them from one another

Brendan Break-in about literals and operator overloading

DH, YK: Do these invariants actually hold? Even if they do, do we really need all of them? Some make sense, but maybe not all. Christian Plesner Hansen's multimethod post: https://mail.mozilla.org/pipermail/es-discuss/2009-June/009603.html Christian's language? http://h14s.p5r.org/2006/05/neptune.html YK: For operator overloading, instanceof won't work in Node because Node agressively duplicates prototypes. instanceof is an antipattern. DH: npm will give you multiple instantiations of the same module.

async await extensibility

sync Iterator:

iterator function* for (x of xs)

async iterator

AsyncIterator async function* async for(x of xs)

sync observable

observable function*> for (x on xs)

async observable

Async Observable async function*> async for (x on xs)

---

function*>() { var stream = await someObservable; for (let price on stream) { yield CAN(price); } console.log("done"); } }

vs. (new for* syntax with sugar -- e.g. on

push(function*() { var stream = await someObservable; for* (let price on stream) { yield CAN(price); }; console.log("done"); }

vs. (new for* syntax - desugared)

push(function*() { var stream = await someObservable; await on(for* (let price of stream) { yield CAN(price); }); console.log("done"); }

```
function* d(xs) { yield xs[0]; }

function d ([xs, gen]) { gen.next(xs[0]); }
```