***Minutes of the:*** ***Ecma TC39-TG1***

***held in:*** ***Mountain View (Mozilla)***

***on:*** ***24th – 26th January 2007***

## Attendees

- Jeff Dyer, Adobe Systems
- Michael Daumling, Adobe Systems
- Brendan Eich, Mozilla Foundation
- Graydon Hoare, Mozilla Foundation
- Lars Thomas Hansen, unaffliliated (invited)
- Cormac Flanagan, UC Santa Cruz
- Chris Pine, Opera Software
- Dave Herman, Northastern University
- Douglas Crockford, Yahoo!
- Dick Sweet, Adobe Systems
- Francis Cheng, Adobe Systems (Wed & Fri only)

## Meeting goals

- Tentative schedule
- Tpec - prototype of code merge (definer, verifier, evaluator and machine code)
- AST - resolve fixmes
- Definer state - finalise semantics
- Verifier state?
- Evaluator state - run Function.es; figure out how to use typechk.sml
- Built-ins - complete first pass over all (modulo Unicode, decimal)
- Running Function.es (or subset) in the r.i.
- Nail down known open issues
- Other goals?

## Agenda

- (from Dick) As I get into the nuts and bolts of adding the “use <numbertype>” pragma to the AS compiler, I have a number of things I’d like to discuss with the committee.
- Primitive types not backward compatible.
- Non-nullable strings.
- Final form of r.i. code.
- AST FIXMEs
- ByteArray.
- Design principles for the builtins: how does subclassing + overriding interact with behavior of builtins?

- for-in semantics: is it Right that for-in iterates over values with an iterator for the object expression and over property names with an object for the object expression, esp when we have for-each for the former case?

- Versioning.

- Prose vs code vs testsuite
  - (from Pratap)
    - What is the guidance on the following: what should be taken as authoritative in the case where the standards prose is at variance with the reference implementation?
    - I recall bringing this up when we met last met at Adobe, and I request the committee to think about this once more: with the goal of coming up with compatible implementations of a common language, I feel that an implementor would be more interested in the availability of the conformance testsuite than an ML-based reference implementation (which would require the implementor to have a strong knowledge of ML). Should we not work on, and publish, such a testsuite too?
  - We should talk next week, but I will write some guidance here:
    - There won't be prose specifying every detail in the reference implementation, by a long shot; nor should there be.
    - The reference implementation will have bugs, just as non-executable specs have in their prose and pseudo-code (but with the refimpl, we can test and fix more readily), so it can't be considered the last word until it is very mature. In the mean time, testing and certainly human review will be needed. Even a published spec has unknown (and sometimes known) errata, and so must be subject to revision. Too many specs fail to reflect reality (e.g., see Mark Pilgrim on HTTP spec vs. reality). So it would be an imposture to say "the prose always wins" or "the ML code always wins". But as noted above, the ML refimpl is where the details are specified; there will not be prose for every corner case or logic level.
    - Test coverage is exponentially complex, and we will never write anywhere near a full-coverage suite by hand. We have agreed to build a common testsuite, although the details for how we'll do this are still sketchy. But in my opinion, we should not produce a normative testsuite along with the ES4 spec, since debugging and completing it to have the necessary correctness and coverage will take many years, even if we use the appropriate concrete and symbolic execution techniques (the subject of ongoing academic research, e.g. CUTE and EXE) to generate tests.
    - Implementors need to use multiple techniques to achieve interoperable correctness: reading the prose; reading, running, and testing against the ML refimpl **and other implementations**; running tests from various suites; testing against real-world content. There is no silver bullet or easy answer. We should not pretend to supply one.
    - **Updated:** With the choice of Standard ML for the refimpl, it is possible (and some think likely) that the refimpl will be translated to a Monadic style and then transported into a metalogical framework such as Twelf, where it can be proven correct. This does not relieve us of the need for debugging, reviewing, testing, and interoperating, of course, since the refimpl may provably do something other than what we intended! Still, another reason favoring the ML refimpl over prose alone, or prose and an incomplete testsuite.
    - **Updated:** Implementors reading the ES4 spec and refimpl may need to learn ML, as you note, but ES1-3 readers had to grapple with its less sound meta-language. At the Newton face to face last October, we agreed not to reinvent ML with our own syntax and bugs, but use the real thing. That decision still looks right to me, even if it means the expository value of the spec is diminished by unfamiliarity with ML on the part of implementors. Prose specs with or without any testsuite often give implementors the feeling that they know what is meant, and lead to wildly varying implementations. Formal methods can be taken to the opposite extreme, of course. I think we are striking the right balance, provided we do work on tests together. But

again i do not believe any tests we write or gather should be normative along with the ES4 spec. Not for many years, at any rate.
    - **Last update:** We are using ES4 itself to specify as much as we can, including almost all of the standard library. So the ML "cognitive load" is less than it might seem at first, and again: ML better defines the parts of ES1-3 that were written in an *ad hoc* meta-language (inside [[double brackets]], sometimes), which were defined informally and sometimes inadequately. — *Brendan Eich 2007/01/19 15:02*

- Non-reserved keywords
    - (from Pratap) Is there a way to treat keywords as identifiers? In C# you can use the prefix '@' for this purpose. For example, @int means use the keyword 'int' as an identifier. From a language interoperability perspective (as possible on the CLR) this is a useful feature.
    - See reserved words, implemented in Firefox 2 (JS1.7), and note that it has always been possible since JS1 in 1995 to use reserved identifiers as property names (e.g., `javaIOFile['delete']()` – in ES4 one may simply write `javaIOFile.delete()`). — *Brendan Eich 2007/01/23 09:56*

- Mixing static and dynamic types in the presence of structural typing (duck typing)
    - We are not all (or at all) in agreement. For example, Dave and I have completely different ideas about what we have decided so far. I move that we cover this Thursday or Friday to give everyone interested an opportunity to read all the materials on the wiki. (It's possible we should do this in a subset group.) — *Lars T Hansen 2007/01/24 08:34*
    - behavior of type checker at `with`, unbound variables, `eval`
    - reject unbound variables? therefore reject `with`? or type-check `with` body at runtime?
    - `eval`: for type-checked code, dynamic error to introduce new bindings into parent function? — *Dave Herman 2007/01/24 21:42*
    - carrot for that stick: make `eval ''...''` (`eval` as unary operator) contain its bindings and otherwise behave better? — *Brendan Eich 2007/01/24 23:24*

- Should U+FEFF (zero-width non-breaking space) in source be treated as whitespace? (We all do it, so let's say "yes" and add it to the spec.) — *Chris Pine 2007/01/25 16:53*

# Notes

## Numbers

### ISSUE 1

There are a number of operators that are binary-centric. Dick proposes that use pragma should only refer to float. E.g. what's use of "use int"? Response from committee, it is useful for many applications. What about support for "big int"? Response: decimal won't give you big int for apps like crypto (only 53 bits). - Committee: we can't change these ops for double for backward compatibility reasons. - Doug points out that the community is comfortable with the internal conversion to int. - Dick is comfortable with following that precedent for decimal. - Example:

```
function f(v) {}

use int ;

f(1.6);
```

- Lars says that you should get 1.6 passed to f() because use pragma affects operators, such as:

```
1.6 + 3; // 1 + 3 because of use int pragma.
```

Lars: history of use pragma. It was brought in to help with decimal, but maybe it is a mistake to give it a broader effect.

Graydon: new proposal. What if use int is a special case. In a use int block just prevent any use of floating point literals at all.

Lars: this has practical appeal, but is inconsistent with ‘use decimal’ and ‘use float’ and it follows that we would just get rid of use decimal. I propose we don’t make this change.

Michael: what about having a pragma that throws on overflow.

Brendan: Requests for this are rare. So what is the meaning of use pragma? Going through the wiki discussion. Question 4 applies to this discussion.

Dave: What does the type checker say when it sees two ints added that could be int or double.

Lars: It would have to call it numeric.

Recap problems:

1. Annotations on values don’t completely control how the numbers are converted.

Current Spec draft: The use pragma affects:

1. operators
2. literals that are members of that data type. i.e. use int will not affect the literal 1.6, but use decimal will turn the literal 1.6 into 1.6m and use int will turn the literal 1 from int to uint.

### ISSUE 2: Number class.

Question 1: what does use Number mean?

Question 2: var v:Number

```
X = new Number (3)  // 3.0d
```

Problem is that Number is an alias, so you can’t really construct a number.

One idea is to say “class Number is Double”.

Question 3: Should use decimal affect use Number? Proposal that “class Number is Double” creates a bw compat problem because Number is a wrapper in es3.

Second idea is:

```
dynamic class Number {
        var d:Double;
}
```

Jeff: The second idea would be incompatible with the AS3 implementation.

**Resolved** that Number should be a wrapper class (second idea).

### ISSUE 3: What about 'use Number'? is it useful?

Lars: could be useful if you are in use decimal and want to drop back into default.

Brendan: what about 'use default numeric'

### Primitive Types Not Backward Compatible

The issue is whether we should add lower case versions of primitives that are unboxed and very fast, and leave upper case versions that are backward compatible.

The only primitive that is at issue here is String. All other primitives can be dynamic and not final.

Lars: proposes that we make String dynamic and nonfinal.

Dave: In Java, there are security concerns about extendable String, so String is final in Java

Lars: what about making String like Number. Make it a bw compat class, and add a new lower case string class that is final and not dynamic.

What about lower case Boolean?

Lars: For Boolean, maybe we should just seal it and break bw compat.

Brendan: I'd prefer to add lower case versions for Number, Boolean, String. Jeff agrees.

**Resolved**: Further discussion lead to a decision to drop lower case versions of Boolean and String because the addition of the two do not provide any efficiency benefit because intrinsic provides early binding (recall that intrinsic implies final).

(This leaves us with `Number`, `String`, and `Boolean` backward compatible but fairly optimizable. The differences with AS3 are:

- Mutable instances, so implementations have to support both `''hi''.foo = 42` and the more common `s = new String(''hi''); s.foo = 42; ....`
- Equality and strict equality return false for two different instances: new Boolean(true) != new Boolean(true) && new Number(42) !== new Number(42).

It would be good to get Ed's comments here. — *Brendan Eich 2007/01/24 14:41*)

LUNCH

### Semicolon insertion in function expression definitions

```
Class C {
  function f()
        37

   -1;
}
```

Lars: what does this mean? I think we should have auto semicolon insertion after 37.

## Various sorts of magical function names

We discussed all the awful sorts of magical function names again. Some revised opinions on this since the last meeting. Current elaboration follows.

| In Source | In Impl | Handles |
|---|---|---|
| `function get y() {}` | "virtual" prop-state | `x.y` |
| `function set y(z) {}` | "virtual" prop-state | `x.y = z` |
| `class X {`<br>`  function X(y) {}`<br>`}` | ctor function plus<br>static X.construct(y):X!<br>that calls new X(y) | `new X(y);` |
| `class X {`<br>`  intrinsic function invoke() {}`<br>`}` | normal function | `z = new X(y);`<br>`z();`<br>when no function closure found in magic slot |
| `class X {`<br>`  static intrinsic function invoke(y) {}`<br>`}` | normal function | `z = X(y);`<br>when no function closure found in magic slot |
| `class X {`<br>`  static intrinsic function to() {}`<br>`}` | normal function | `y to X;` |
| `function get*(n:string) {}` | magic part of object | `x.y` |
| `function set*(n:string) {}` | magic part of object | `x.y` |
| `function call*(n:string) {}` | magic part of object<br>(special name lookup stage,<br>after fixtures, before<br>dynamic props) | `x.y` |

## Better eval

- Much discussion of `eval` vs. `with`, how to reform `eval`, alternative syntax or keyword ideas, etc.
- Proposal: new pragmas, `use safe eval` and `use unsafe eval`, orthogonal to strict/standard and in addition to resurrected eval.
- Under `use safe eval`, `eval` cannot create bindings in its caller's lexical or dynamic scope because it gets its own variable object and implicit block
  - A safe `eval` can still see its caller's bindings and mutate the values of any mutable bindings, subject to type constraints.
- This allows type-checked `eval` callers and `eval`'ed programs (strict is propagated to `eval`, as are other pragmas such as `use decimal`).
- It is an error to call `eval` in strict mode without a `use safe eval` pragma being in effect.
- Take `safe` and `unsafe` as strawman proposals; we don't want false advertising. Suggestions welcome.

## Primitive Types, Day 2

- Yesterday's proposal plus non-nullable String (whose default value would be an, if not **the**, empty string) means every `''`is a different object.
  - `var s1 = '', s2 = ''; s1.foo = 42; assert(s1.foo === 42); assert(s2.foo === undefined).`
  - Neither backward compatible nor as easy to optimize as either ES1-3 or AS3.
    - Utmost backward compatibility (perhaps breakable without pain) would want `s1.foo = 42; assert(s1.foo === undefined)` – each `.` operator creates a new wrapper.

        - In the minutes_feb_21_2007 meeting on Friday, some of us sort of talked ourselves out of “Day 2” and back into “Yesterday”‘s proposal. — *Brendan Eich 2007/02/23 12:07*
- Jeff suggested catchalls on prototypes to handle “peeks and pokes”; we know of real-world code that peeks and wants to get `undefined`.
- Lars points to `s = new String(...)` uses on the web that want a mutable `String` object.
- So we are back to having a sealed `string` type for literals, concatenation results, and a backward-compatible `String` that can box `string`.
    - Same for `boolean` and `Boolean` – need the former because in `b = new Boolean(false); if (b) ...`, `b` always tests as true.
    - Jeff’s prototype catchalls suggestion for `string` and `boolean` help avoid boxing costs.
    - Note that `string` and `boolean` are non-nullable and final.
    - Open issue: Should `String` and `Boolean` be final?
        - My recollection is that we said “no”, but `string` and `boolean` are `final`.
        - Further issue, I’ll mirror it in the minutes_feb_21_2007 minutes: web script extends `String.prototype` with new methods, expects them to apply to string primitive values. Suggest `string <: String`. Details to be worked out in the February minutes, or beyond. — *Brendan Eich 2007/02/23 10:05*

## ByteArray

- See bytearray.
- Need an up-down vote or an alternative proposal, quickly.
- Lars notes that this class makes sense only if native (for performance, since people make do today with string or array hacks but complain about space and time costs).

## Type System

One change to yesterday’s discussion: introduce a new type `{*}` to represent the type of untyped objects (”star for objects”), rather than making `{}` a special case. So `{}` still means the structural object type with no fields.

### Equality

Some types are considered to be identical to each other (like a shorthand) including:

- `[]` and `[*]` and `Array`
- `{}` and `Object`
- reorderings of structural object type fields
- no `this` type in a function is `this:*`
- `this` type in a method of `T` is `this:T`
- unions are flattened
- nullable `T` is `(T,Null)`

### Subtyping

- written `T <: U`
- Same as nominal/structural subtyping we’ve discussed in the past
- transitive
- anti-symmetric (if `T != U`, they can’t both be subtypes of each other)

### Compatibility

- written `T ~: U`
- means `T` may not be a subtype of `U` but can be used compatibly with `U`
- such uses require the “unsafe bit” to be enabled and runtime checks to be performed
- but does not involve conversions, so object identity is preserved
- not transitive (because it has cycles)
- not symmetric (because it includes subtyping)
- write `T ~ U` for `T ~: U` and `U ~: T`
- includes subtyping and universal compatibility with `*`:

```
T <: U
------     ------     ------
T ~: U     T ~: *     * ~: T
```

- includes compatibility of structural objects with the “untyped object” type `{*}`:

```
------------------------      ------------------------
{x1:S1 ... xn:Sn} ~: {*}      {*} ~: {x1:S1 ... xn:Sn}
```

- includes compatibility with array types:

```
--------------------     --------------------
Array ~: [T1,...,Tn]     [T1,...,Tn] ~: Array
```

- lifts through function types:

```
U' ~: U
T ~: T'
forall i <= n. Si' ~: Si
-------------------------------------------------------------
function(this:U,S1 .. Sn):T ~: function(this:U',S1' .. Sn'):T'

U' ~: U
T ~: T'
forall i <= m. Si' ~: Si
forall m < i <= n. Si' ~: V
-------------------------------------------------------------------
function(this:U,S1 .. Sm,...V):T ~: function(this:U',S1' .. Sn'):T'

U' ~: U
T ~: T'
forall m < i <= n. Si' ~: V
forall n < i <= m. V' ~: Si V' ~: V
-----------------------------------------------------------------------
function(this:U,S1 .. Sm,...V):T ~: function(this:U',S1' .. Sn',...V'):T'
```

- lifts through union types:

```
exists i. S ~: Ti
-----------------
S ~: (T1,...,Tn)
```

- invariant through object types:

```
forall i <= m. Si ~ Ti
-------------------------------------
(m <= n) {x1:S1 ... xm:Sm} ~: {x1:T1 ... xn:Tn}
```

- invariant through array types:

```
forall i < m. Si ~ Ti
forall m <= i <= n. Sm ~ Ti
--------------------------  (0 < m <= n)
[S1,...,Sm] ~: [T1,...,Tn]

      example: [int,*] ~: [int,bool,string]

forall i < m. Si ~ Ti
forall m <= i <= n. Si ~ Tm  (m > n > 0)
--------------------------
[S1,...,Sm] ~: [T1,...,Tn]

      example: [int,bool,string] ~: [int,*]
```

### Convertibility

- written `T ~~> U`
- means `T` may not be compatible with `U`, but can be used as a `U` via a conversion
- such conversions do not preserve object identity
- includes convertibility (and therefore subtyping):

```
T ~: U
-------
T ~~> U
```

- includes `to` methods:

```
T is a class
static T.intrinsic::to : S' -> T
S ~~> S'
--------------------------------
S ~~> T
```

### Questions

- what namespace do the `to` methods live in? – `intrinsic`
- are arrays invariant or covariant? – invariant
- are people okay with the `{*}` type (for “untyped object”)? – yep

### Moved to spec page

I’m folding these rules back into the type_system page.

I’m not deleting the rules from here, since they’re a record of the meeting, but they’re not authoritative. Future changes will go in the spec page.

— Dave Herman *2007/01/26 13:36*

## Zero-width non-breaking spaces

- Should U+FEFF (zero-width non-breaking space) in source be treated as whitespace? (We all do it, so let's say "yes" and add it to the spec.)
- Background: Firefox 2 strips it, but the plan is for Firefox 3 to not strip it and treat it as a format control character.
- Resolved: after examining the Opera bug report, Chris revised the proposal to say that the ES4 will treat U+FEFF as it does any other format control character. That is, it won't be stripped and cannot be used in identifiers. This is consistent with the existing proposal to update Unicode (Update Unicode).

## Dotted name ambiguity

Ambiguity between package reference and dotted object reference. Resolved:

- no feedback of scope back to lexer
- spaces allowed between dots even in package names
- lexer produces sequence of identifier tokens separated by dot tokens
- parser produces an AST node representing an unresolved path
- definer resolves the path `I1 . .. . In` to either a package dereference or an object dereference path:
  - if the first name in the path is statically in scope, then it's resolved as a dereference path, regardless of type information etc.
  - otherwise find the longest prefix `I1 . .. . Im` that is either an imported path or the name of a package in this compilation unit
    - if this prefix is an imported path and the identifier on the right of the dot following that path matches a name imported from that package, then resolve the expression to the expression `(I1 . .. . Im-1)::Im`; then resolve the rest of the path as an object dereference path from this expression
    - otherwise if it's a package name and m < n, resolve the prefix with the next identifier to `(I1 . .. . Im)::Im+1`, then recursively resolve `Im+2 . .. . In`
  - otherwise if there is no such prefix, then it's an object dereference path with an unbound head variable (to be resolved dynamically)
- notice that introducing bindings dynamically (via global object or `with`) have no effect on this resolution process, since it happens in the definer long before runtime

— Dave Herman *2007/01/26 16:24*

## Nullability conversions

- no implicit conversions between nullable and non-nullable types
- can convert via `cast` if you just want to assert it's not `null` (get an error if it is)
- if you want to do something more complicated, you can use `switch type` and it's well-typed (yay Modula-3)

— Dave Herman *2007/01/26 17:33*

## Interface subtyping

- interfaces allow method redeclaration with appropriate subtyping (covariant return, contravariant arg)
- this might result in unimplementable interfaces, e.g. if two superinterfaces have versions of a method with unrelated return types
- but sometimes these types might have a common subtype introduced later, e.g. by a new class definition in `eval`
- the algorithm for detecting these impossible-to-implement interfaces is either complicated or impossible
- we could just make method redeclaration invariant or illegal (in AS3 it's illegal)
- resolved: just make it legal
    - on interface extension or implementation, every method must have correct types w.r.t. its (super)interfaces
    - so if it's impossible to implement your interface, you'll find out when you try

— Dave Herman *2007/01/26 17:33*

## Hashes

- Brendan: should we make typed object literals not have a prototype?
- Douglas: prototype pollution is a very common source of problems for our coders
- problematic because it's a little harsh
    - can't call `hasOwnProperty` or `toJSONString`
    - although could perhaps convert to an object via `Object.wrap` or something
- other options:
    - `Dict` class like AS3, with `to`-conversions defined from other objects
    - `#{...}` syntax for special hash objects (would then need the type system to know the difference, because you can't call `Object` prototype methods on these)
- Jeff: no new features
- Brendan: gentlemen's agreement to match extensions via deferred proposals for ES4.1 or whatever

— Dave Herman *2007/01/26 17:33*