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

held in: ***Oslo, Norway (Opera)***

on: ***27th-28th July 2006***

Attendees

- Doug Crockford, Yahoo!
- Cormac Flanagan, UC Santa Cruz
- Brendan Eich, Mozilla Foundation
- Jeff Dyer, Adobe Systems
- Dave Herman, Northeastern University
- Graydon Hoare, Mozilla Foundation
- Pratap Lakshman, Microsoft
- Lars Hansen, Opera Software
- Michael Daumling, Adobe Systems
- Charles McMathie-Neville, Opera Software
- Chris Pine, Opera Software

Agenda

- Introduction from Yahoo! and Microsoft
- (Auto-)conversion of structured types
  - the perceived benefit of mixed typed/untyped code
  - problems that might occur, esp covariance on mutation
- Date literal
  - good enough?
  - ambiguities?
- Multiple compilation units – go over it again
- Go through the proposals again, make sure we're still in agreement
- Non-nullability by default
- "Modules"
  - Package loading
  - Advanced visibility control and optimization of imports
- Next f2f date?
  - We talked previously about meeting at Adobe in Newton, MA (Boston area)
  - ICFP is in Portland, OR on 18-20 September
  - TC39 meeting is in Portland in October

Summary 27 July

Yahoo!s position

Yahoo very big user of ECMAScript.

Various ideas surveyed in company:

- Flash group: “Whatever Adobe is doing is right.”
- PHP group: “why not $ on every var?”
- Java group: “may they burn forever”
- JS group: “please don’t mess it up”

Backwards compatibility hard & important, some 3rd Ed features just now becoming usable (because of dropping support for very old browsers). Using new features is very hard (e4x as example), and some features don’t add anything (e4x again). These kinds of features of no/little value to Yahoo!.

Security the biggest and hardest problem they’re facing, and would tolerate backwards incompatibilities / syntax errors for features of this magnitude. (Mashups, ads, eval.)

Yahoo!s position is that creating a new language would be of value (and supporting it in the SCRIPT tag).

Discussion on version parameter on type attribute of SCRIPT: IE6 ignores it, hard to use, but there is a standard for it now. The problem is browsers already deployed. Do we want to / need to fix this?

Doug: unclear that features like classes are a benefit even long-term, JS-like-Java programs tend to be larger and hairier than idiomatic JS.

Jeff: would be positive to have feedback on the proposed features from eg Yahoo!.

Brendan: TG1 is in fairly good agreement on what goes in, we just need to get it sound. We have to worry about finding incompatibilities in the spec.

Microsoft's position

Two scripting engines: a C++ engine for JScript used by Windows and MSIE, and a C# engine used by JScript.NET. *Any* JScript program is a legal JScript.NET program and they are compatible (by the definition of passing a compatibility test suite). Currently not possible to use only one, because the CLR is not available everywhere, and MSIE can’t depend on that (yet). If they can do a stripped-down CLR for MSIE they will ditch the C++ engine.

A lot of script code is being written at Microsoft, with Atlas. live.com is massively scripted.

Priority 0 is backwards compatibility. *Only* for security fixes will backwards compatibility be broken.

ECMAScript 4 will not make it into MSIE 7 or Windows Vista. But they are keen to get it into products shipping after that.

Conversion of structured types

It’s possible we need some sort of conversion of structured types to support mixed typed and untyped code.

It’s possible to annotate structure literals with types so that they can be passed to typed code. But it’s unclear how tractable this is, since literals may be created in many places, in legacy code, or in code with other constraints.

Lars: my main concern is that we have no data, that we’re arguing for a use case that has not been demonstrated to exist.

Brendan: my feeling is that there will be some conversion trouble, there will be an interesting class of programs that’s excluded by “rule 4”. Dyer: disagree that typed literals is sufficient to solve real problems.

The consequence of "rule 8" is that *every* field access (not just write) in typed code can throw an exception, since the access through a proxy must fetch a value of the type of that field in the proxy.

Having an explicit conversion is "nice", but what is the meaning of the conversion? Copying is also "nice", but deep copy has problems. Graydon: anything but copy-on-write is problematic (?).

Cormac: does getters and setters get us into problem anyway? Brendan: they tend not to throw.

Jeff: Concerned about explicit conversion – you still have to change code that you do not have access to.

Dave: applying typing to eg the DOM hierarchy will break all existing (untyped) code that works with the DOM code without some sort of workaround. Lars: do we get one wrapper/proxy per DOM method per tuple of actual parameters, ie, a big space issue?

Pratap: concerned about performance implications of mixing untyped/typed because a lot of conversion. Brendan: yes, if performance is poor then people will just not use it. Dave: a tight loop across a proxied object could be bad.

"Rule 8", which adds runtime checks "whereever necessary" may be efficiently implementable through storing a bit in every slot (or in the pointer from that slot) that says that the type on the slot is different from the type of the thing it references, and a read barrier is needed to see if the type in the slot in the object is of the correct type for the variable / context.

Dave fixes solution 4 and Graydon writes up solution 8 in more detail with discussion of implementation choices; there is general consensus toward solution 8 because it appears to be efficiently implementable.

Clarifying example:

```
var x : { i : int } = { i : 10 }  // conversion from untyped to typed
var y = x
delete y.i                        // not an error, y is untyped
var z = x                         // no error here
print(x.i)                        // error here
```

Cormac's matrix:

```
                        untyped r/w      typed r/w
  obj from untyped                       read check
    code, not type      "no" checks      that slot gives
    on slot                              value of right type

  obj from typed        write check
    code, type on       that value is    "no" checks
    slot                of right type
```

There is initially a run-time test to find out if your object came from typed or untyped code, so that is implied. Brendan: you want a single fast check to get to the fourth quadrant.

Object field covariance

Are object fields covariant?

Suppose FastTCPSocket <: TCPSocket.

```
type T = { x : TCPSocket }
type U = { x : FastTCPSocket }
```

Is U <: T? If so (says Cormac), we need other kinds of barriers or checks: the x field of a U can only hold a FastTCPSocket, so a function taking a T needs to check that stores into its parameter actually store FastTCPSocket or its subtypes. The check is there already, though it needs to be a subtype check, which may not be very fast.

Brendan: how does this fit with JSON? Any other use cases for this?

Graydon: there is the versioning problem. Consider:

```
{ x: { a: 10 }, y: { p: 10 } }
{ x: { a: 10, b: 20 }, y: { p: 10 } }
```

Here the only thing that's happened is that a field has been added (protocol upgrade, say). So without covariant structural subtyping we'll have problems here.

We agree we need this.

(Another solution is const fields in structures, but there's no agreement on this.)

Date literals

There seem to be very little justification for date literals per se, so there's tentative(?) agreement to drop that proposal.

But there also seems to be agreement about fixing Date.parse() by defining a literal format it is known to understand, and for defining eg Date.toISOString() to produce a string on this format always. The format would be an ISO time string, probably not unlike what the date literal proposal has currently.

Graydon: what resolution? Brendan: milliseconds.

But we could do a separate proposal for nanotime (or better).

Doug: could have a variation on the Date constructor that takes a structure with fields like "month", "day", etc with defaults taken from the current date. This would fix the problem with `Date.parse(''9/11'')`, for example. Brendan: It's not backwards compatible, though.

Doug elaborated his proposal by adding a test for `Object.prototype.toString` being overridden, either replaced in `Object.prototype` or shadowed in the object or a nearer prototype: if `toString` is overridden on the single object argument to the Date constructor, then it should be called per ECMA-262 Edition 3 15.9.3.2 `new Date(value)`; otherwise, we know the default conversion to primitive will call `Object.prototype.toString`, which will produce a string guaranteed not to parse as a date, and we should instead look for `month`, etc. properties in the object.

— Brendan Eich *2006/08/03 18:27*

Brendan: let's see if we can define the parser more rigorously based on what we know about how people write dates on the web.

Charles: use W3C dates (rather restrictive) rather than full ISO dates.

Summary 28 July

(Not present: Charles McCathie-Nevile.)

Non-nullability

Brendan/Jeff: want non-nullable types by default.

Lars: disagree, makes instance initialization too hard in practice. Any pointer field that can't be given a constant initializer will end up being defined nullable because it won't be initialized until the constructor.

Brendan: Tucker argued that you can handle initialization through definite assignment analysis, but we don't want to impose that.

Dave: given that we require the C++ style initializers for non-nullable fields it seems like a tall order to require the majority of fields to be initialized that way.

Graydon: disagree, seems just fine to me. The language "nice" http://nice.sourceforge.net/ has more interesting constructor semantics, where the last action in the construction sequence is always a call to a generated constructor that takes arguments for all fields of the class and initializes them.

Interaction of structural types and non-nullability? Brendan: Array and Object and Function are nullable, and structural types are mapped to these, so structural types are nullable by default.

Jeff: null are lightweight default values, with non-nullable as default we force people to write heavyweight default values.

Jeff: who wants non-nullable types? Ajax people who now write executable assertions in their code will probably be happy.

Brendan on whether function types is nullable or not:

```
  var f = Function.<A,B,C,R>("a", "b", "c", "return a+b*c");
  type F = function (A,B,C) : R;
  let g : F = function (a:A,b:B,c:C):R { return a+b*c };
```

One view is that the value of g in 3rd Ed has a nullable type; the other view is that 3rd Ed does not have types at all so the issue does not arise, and clearly a function value is never null. By this view, any structural type could be said to be non-nullable (the type of the object always describes an object).

In the end: types are nullable by default.

Looks like type expression syntax needs to evolve a little to be able to express non-nullable structural types conveniently. Today this requires two type definitions because `!` and `?` can only be applied to type names. Dave thinks we should be able to write e.g.

```
  type F = (function (int) : int)!;   // otherwise ! would bind to the second
int
  type G = {"a": double, "b": int}!;  // no ambiguity
```

Mini-proposal: Namespaces as capabilities

Graydon: Proposal for extending namespace system into simple system of revokable capabilities:

```
namespace ns;
ns var x = 27;
var c = new Namespace(ns);  // make a revokable copy of ns
```

The client obtains c and uses it to access properties:

```
x.c::foo
```

Now later I can do

```
c.revoke()
```

to kill c, and make it impossible for the client to obtain properties in that namespace.

Discussion: Revocation does not kill values previously extracted, notably bound methods, so it's a somewhat limited security mechanism. You could make revokable namespaces wrap bound methods when they are extracted so that they are affected by revocation, but then they are not really namespaces any more.

Also a possibility to use stack-based security: pushing the namespace on the stack while performing an operation. Graydon to elaborate.

Multiple compilation units

Do we want something structured like AppDomains, class loaders? ActionScript has multiple global environments, one per compilation unit. A possibility is to give `eval` a second argument that is the environment object it should use as its outermost rib:

```
eval( "...", new Sandbox(...) )
eval( "...", { ... } )

some_window.eval("...") == eval("...", some_window)
```

Doug: possible to put types in that object that the typechecker (and the code loaded in that context) will use?

Brendan: must be careful about sharing type objects between environments because their prototypes are mutable, for one thing. But desirable. GreaseMonkey already does some of this, it injects privileged functions into the sandbox for user scripts. Brendan will try to write some of this up.

Modules

Two things: naming and loading.

Naming: Javaesque DNS naming (ie p.q.r.useful means something), or content hashing (ie package name is 40-bit hash, say).

Jeff: import cannot affect loading. There must exist some separate loading mechanism, something that says ahead of time to load the imported package. Could happen during the definition phase when the package is referenced. Consider example:

Fragment 1:

```
import p.q.T;
var x : T = ...
```

Fragment 2:

```
package p.q {
  public type T = ...
}
```

With a verification phase, F2 must be loaded when F1 is verified. Without that, it need not be loaded until F1 is executed.

Agreed: the parser throws an error if an imported package is not available at the point of import (aids early error detection, and is pretty much required for some sort of name resolution anyway).

Agreed: Making imported packages actually available is the responsibility of the host environment. (Typical browser scenario: a script writes a script tag that loads one version or another of package named `p.q`; a subsequent script then imports `p.q`. Browser vendors might agree on improvements to SCRIPT, but not under the auspices of this group.)

Agreed: we axe the `include` directive from the language, packages are better both semantically and from a performance point of view.

Eval

Various cases:

```
eval(s)         // ok
window.eval(s)  // ok
eval            // throws EvalError
o.eval(s)       // throws EvalError for some random o
```

In practice this is good enough.

Brendan: Proposal for ES4: o.eval(s) throws Evalerror unless o is a global object.

Another proposal: If we do allow `eval(s, o)` then `o` becomes the outermost object in the scope chain for the evaluation.

(Lars will write up these as a "Resurrected eval" proposal.)

Brendan: SpiderMonkey allows `o.eval(s)` for arbitrary o, this means `with (o) eval(s)`, but it's Mozilla's problem.

Agreed(?): Define `intrinsic::global` to get the global object (or the "current global object" in systems that have more than one).

Iterators/generators

Example of generator:

```
function count(n) {
   for ( let i=0 ; i < n ; i++ )
      yield i
}
g = count(10);
g.next() -> 0
g.next() -> 1
...
g.next() -> 9
g.next() -> throws StopIteration
```

Here StopIteration is a singleton whose [[Class]]=="StopIteration" and whose [[Prototype]] is Object.prototype.

A function containing `yield` returns a generator object when called (no `return` statement is allowed). This is an object with the following methods:

```
g.next()
g.send(v)
g.throw(e)
g.close()
```

Example

```
function toy() {
   print(yield 1)
   print(yield 2)
   print((yield 3) + (yield 4))
}
t = toy()
t.next() -> 1
t.send(10) prints 10
           -> 2
t.send(11) prints 11
           -> 3
t.send(12) -> 4
t.send(13) prints 25
           throws StopIteration
```

On the use of close():

```
function foo() {
  try {
      yield commit();
  }
  finally {
      cleanup();
  }
}
g = foo();
g.next();
g.close();
```

If the program does not call close(), the GC must – so we must have some sort of finalization (weak pointers / guardians, at a minimum).

Generators are parameterized structural types:

```
type Generator.<I,O> = {
       next:  function () : I,
       send:  function (O) : I,
       close: function () : void
     }
```

On the topic of iterators. Consider an enumeration:

```
for ( i in o )
```

The situation is that:

- real world wants property creation order (except maybe for arrays?)
- ecma says impl. dep. order
- shadowing must be handled
- delete coherence required

Now suppose `it` is an iterator, ie `it <: { ''next'' : function () : * }`

```
for ( i in it )
```

Here we get:

- call it.next() until StopIteration is called

You can call `Iterator` on `o`

```
it = Iterator(o)
```

This may return o if it is an iterator already or the result of something like `o.intrinsic::iterator()`.

To correct my misstatement: `for (i in o)` always calls `Iterator(o)` to get or create an iterator for `o`, which will be `o` itself if it is an iterator object. So there's no backward compatibility problem: ES4 introduces intrinsic and defines intrinsic::iterator in Object.prototype. This is the "iterator getter" method delegated to by `Iterator`. Custom iterator getters defined on more derived prototypes, or on individual objects, give custom value iteration instead of ECMA-compatible property enumeration. More in iterators and generators shortly.

— *Brendan Eich 2006/08/03 18:16*

(Extensive discussion not recorded, but in the end there's substantial doubt about whether this proposal pays for itself.)

Proposal walkthrough

(Everything not noted was reaffirmed without discussion)

Reaffirmed after brief discussion

- Type parameters: seem OK
- Nullability: C++ type initializers OK; punctuation is `: initializers`
- Enumerability: Brendan thinks it may be a hack but hard to object

New (some not yet on the wiki)

- (Mid-level) intrinsic::global
- (Mid-level) security wrappers
- (Mid-level) resurrected eval
- (Surface) array comprehensions (split from slice/splice)

Promoted, not yet finished

- “Date and time”, absorbs “Date literals”

Proposed

- (Pratap) Non-type-safe formatting library

Open

- Typing of initializers: a little debate (unresolved): is a full type expression allowed or just name?
- Numbers: proposal open, needs more discussion
- Type refinements: blame assignment is definitely out, an implementation needs to appear for this to live. Dave: perhaps just an assertion mechanism would do
- Syntax for type expressions: add postfix `!` and `?` on type expressions
- Iterators and generators: significant turmoil. No agreement.
- Slice/splice: do we want these?
- JSON library

Demoted

- Date literals are out, but will be folded into “Date and time”

Misc

Implemented proposals should be marked as such with appropriate emoticon

Things generally won’t be written up in the spec until they’re implemented (though there may be exceptions).

Meeting schedule

Mozilla, Mt View, CA – September 21/22

Adobe, Newton MA – October 19/20

Yahoo!, Sunnyvale – November 16/17 – maybe interface with WebAPI group

Soft deadline in September and much harder deadline in October for implementations.