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

held in: ***Sunnyvale, CA***

on: ***19th-20th of July 2007***

Venue

Yahoo! Inc. Building E 700 First Avenue Sunnyvale, CA 94089

http://maps.yahoo.com/#mvt=m&q1=700+First+Avenue%2C+Sunnyvale%2C+CA&trf=0&lon=-122.024953&lat=37.416118&mag=3

Dinner 6:30 Thursday at Kabul (http://www.kabulrestaurant.net/). Please RSVP to crock@yahoo-inc.com

Attendees

- Douglas Crockford, Yahoo!
- Lars T Hansen, Adobe Systems
- Francis Cheng, Adobe Systems
- Jeff Dyer, Adobe Systems
- Allen Wirfs-Brock, Microsoft
- Pratap Lakshman, Microsoft
- Chris Pine, Opera Software
- Cormac Flanagan, UCSC
- Brendan Eich, Mozilla Foundation

Agenda

- [Lars] Arrays (66, 68, 69, 120). See the arrays proposal page.
- JSON, notably its interaction with namespaces, if any, and any lingering discussion about APIs. Please create tickets if necessary.
- [Lars] Various Date tickets (129, 130, 131, 133, 134, 135, 136). See also the date and time proposal.
- [Cormac] Tickets 96, 120, 121, 122, 127 on various type system issues.
- [Lars] Dict. See the dictionary proposal page.
- [Francis] Which version of Unicode are we planning to support? Latest seems to be Unicode 5.0.0.
- Graydon's ticket 113 about `int` and similar types converting from `*`.
- Go through other active tickets by milestone.
- Are monotyped and non-monotyped arrays compatible? Has implications for cost of compatibility checks.

Notes

arrays

- Idea is to keep changes simple. Add a typed array, called 'array' vs. 'Array', that has additional semantics. Namely, allow for dense arrays and eliminate the prototype.

- Specifying two or more types (tuple) creates a fixed length array. There is a read-only fixture named 'fixed' in the array class, which is writable if one type (or *) is specified. Example is [i, x] = [int, string] is a tuple of fixed length 2. If you do []:[int], you get a array that is not fixed length.
- Jason's syntax suggestion is to go with int[] instead of [int], which allows you to have a single-typed tuple. On the other hand, a one-tuple is not useful.
- Error messages: If you index below zero or beyond length, you get an exception (regardless of the value of fixed)
- A = [1, x]:[int, string], then A[0] is type int.
- Might be helpful to have a separate AST node to distinguish single versus multiple tuple.
- Complexity budget analysis. Do we want this added complexity. Folks are already using Array to do things that would be much simpler to do with (small-cap) array. This shifts the burden to lang implementors (and us). This would also resolve a number of open questions about our structural type system.
- Open issues (see array proposal page for all open issues). What about the generic methods from Array? All Array methods should be able transferred to array, perhaps optimized. On the other hand, should it be called 'array' at all? Could make it nameless, or call it vector. The array class will be final and nondynamic. Argument for 'array' is that it is consistent with the 'string' and 'String' classes.
- BE brings up issue about default sort comparison is string comparison. This causes slowdown when sorting numbers, and is especially noticeable in benchmark tests. Making default sort type aware would be desirable. BE will add this to the proposal page.
- CF brings up asymmetry. [int, string, int] <: [int,string]. However, [int, string] is not <: [int], but the following is true [int, string] <: [*].
- LTH: what about [int, string] <: Array. AWB brings up that subtype relationship is not clear. CF thinks that it's clear that there is not a subtype relationship.
- BE: So if we say that subtype relationship doesn't exist, there should not be inheritance? LTH says this is an open issue.
- JD: Bothered by inconsistency of [int] meaning something so different from [int,string].
- So how will sort work with array. Say you call sort() on [int, string, int]. Could be a runtime error?
- Asymmetry between adding/deleting elements? Why is that necessary? LTH will update the proposal to clarify the meaning.
- Will this class have a prototype object? This is an open issue. Last meeting we talked about having a shared prototype for array and Array.
- naming: array vs. vector. Several TG1 members favor vector because it reduces confusion with Array. BE is less enthusiastic because in common usage, vector means monotyped array, not multi-typed tuple.

JSON

- namespaced enumerable properties.
    - Could use :: but servers might not like it. DC: JSON wants to be lang neutral.
    - BE: What if we just throw an error. DC: That, or just give people an opportunity to filter the names out.
    - BE: Could skip namespaces.
    - RESOLVED: JSON encoding will skip namespaces.
- Date syntax. People expect ISO format.
- SAX-like non-blocking/incremental API (discussion deferred)
- toJSON. We are only doing toJSONString methods.

Various Date tickets

- 129, RESOLVED: approved and assigned.
- 130, RESOLVED: approved and assigned.
- PL: asks about nanoage rationale. LTH: Early request from discussion list. AWB: Why not just add a Timer class? BE: we're trying to restrict spec to basic functionality, not adding full libraries.

- 131, This ticket tracks specific incompatibilities with various date formats.
- 133, RESOLVED: approved and assigned.
- 134, RESOLVED: approved and assigned.
- 135, RESOLVED: remove restriction. Approved and assigned.
- 136 RESOLVED: remove restriction. Approved and assigned.

Type issues

- Type system issues

Type annotations on properties specified in structural object type only checks for compatible types when assignment with structural type happens, but does not otherwise constrain type of property.

```
type I = {getIter:function(): J} type J = {next, function(): *}
Array.prototype.getIter = function(): J {var self = this; var i= 0;
return {next: function() { return (i < self.length)
? self[i++]
: undefined }                                                    }
}   a = [0, 1, 2, 5, 4] for (let i in a)   (a to I).getIter() is J
```

LTH:

```
if (x is { equals:function(k):boolean } )     var x : {y:A} = {y : new B}
```

LTH: given assignment, meaning of y's type annotation means only that B must be compatible with A, but it does not mean that there is a type constraint on y when accessed later.

BE:

```
x.y = new C // (where C !<: A and C! ~~>A and this is not known to Strict mode)
OK print (x.y)  // RB (Read Barrier) x = {y: new C} // BAD
```

LTH: We should come up with a set of use cases.

Use cases:

1. There exists a method in this object... (need to figure out when and where; get a contains?)

2. Shape testing, eg.

```
  switch type (v)        case ({p:int,            q:{r:string,
s:boolean}) {...}     default {...}
```

JD example:

```
type MULTINAME =    { nss:   [[NAMESPACE]]      , id: string } function
lookup(mname:MULTINAME)...
```

Friday, July 20

Type Issues (continued)

Take the following function:

```
function f (p : {x:int} ) {     return g(p); }  f({})
```

What happens when f is called?

- current plan is that no error occurs because parameter is compatible with untyped object
- JD wants an error in this case.

CF: What if x is a getter?

```
f ( {get x():* {return "x"} } ) f ( {get x():string {return "x"} } )
```

USE CASES:

1. There is a method...
2. shape testing
3. Typed interfaces, untyped code
4. Avoid performance traps
5. Typo finding in the callee (by restriction) (analogy 2/ 'reformed with')
6. f({...}); f(new C)
7. Check reads & writes deeply

Important issue is that comprehensive type verification could be very costly wrt performance.

What some implementors do is targeted (or ad hoc) type verification to avoid performance hit but still provide some measure of type checking.

AWB: another use case is tooling or IDE support, eg. Microsoft Intellisense.

CF: I see 3 design options, though there may be more

1. No structural object types
2. Structural object types but no 'deep' checking on untyped objects (note: no check on entry for untyped objects)
3. Structural object types and 'deep' checking

LTH : time dimension is not included in above options, on storage and on access. use case 5 is not covered by any of these 3 options.

BE : other dimensions to consider

- time
- strict/standard
- deep
- wide

LTH: One of the main use cases suggested by use case 3 is the interaction between existing prototype code and new typed code.

PL: read barrier may be required to safely interop with untyped code that could possibly mutate the type on a value.

CF: A related issue is if you pass in an untyped array when the formal parameter calls for a typed array, do we check the entire array?

LTH: structural objects will usually be smaller than arrays, because generally used as ad hoc interfaces, so the overhead is less of a factor than it is with arrays. We should add this, however, as a topic for further discussion.

LTH: Here's a proposal, we solve use cases 1-4 in standard mode, and use case 5 in strict mode. Check type when you write, not when you read.

CP: Rationale is that you are catching the error earlier.

CF: One issue with this proposal is that it would be nice to have check on read when concurrent modification occurs.

LTH: I'm arguing that you are talking about a research issue, and we don't have the luxury to consider it given our time frame.

CF: Is there any way to leave some wiggle room in the spec so that an implementor could opt to include checks on read.

BE: Could lead to interoperability problems

JD: What I'd like to have is a type system that guarantees a type on read. I'd rather defer than have a weak guarantee.

LTH: Another way of looking at my proposal is that given

```
function f (p : { x:int } ) {     p.x // not type checking here }
```

The formal parameter is saying something only about the structural object p, namely that there is an object p that has a property x of type int. Consequently, we only type check when we write a value to p.x as we do when we bind the argument value to p when f() is called. We are not saying anything about the property x itself.

JD: So is this future proof with the possibility that we can one day solve this research problem.

BE: I'm more concerned with the present than the future. We may find that there is no use case for solving the research problem.

LTH: The MochiKit use case:

```
type ArrayLike = { slice: function (...): *, length: uint }  f ( x: ArrayLike)
```

BE: better done using array structural types, probably.

PL: the type checker will enforce type invariance in the following case:

```
function f(p: { x : int} ) {   g(p); }   f( { } )
```

BREAK FOR LUNCH

CF: We could restrict deep checking to function entry and exit.

BE: why do that if it's a small difference to support the same meaning for other uses of type annotation syntax?

Array questions relating to runtime types and conversions:

LTH:

```
 fun f (xs: [int]) {      }  f (new [double]) // makes no sense f ([1, 2, 3 ])
// makes no sense  g (xs:Array) {...}  // makes sense g (new [int] )      //
makes sense g (new [*] )         // makes sense
```

Brendan argued that `f([1,2,3])` makes a lot of sense because the syntax is coveted. It may be more desired for tuples than homogeneous arrays. We ended up talking about whether it could be supported (some open source implementations optimize `Array` for the dense, natural-property-order case already into something like a `vector`). Not clear where this ended up, but it may "make sense" yet. — *Brendan Eich 2007/07/20 18:06*

Lars has written up his proposal in the clarification namespace as Solution 9 Write Barrier with deep checking

---

Discussion about merging solution 9 into solution 8 – needs to be captured properly. Essentially, solution-9 like checking at function interfaces (covers the use case), but solution-8 like checking elsewhere.

Cormac: In some sense, solution 9 is more about contracts than types.

That is an interesting point, and in some sense it sounds like maybe we are moving towards contracts for structural annotations and types for nominal annotations.

A further observation: structural object type annotations on interfaces, nominally-typed or untyped object instances passed and returned, complement one another:

- We expect (use case 3) much client code to remain untyped, forever.
- We know of nominal native and AS3 types today, and expect some nominal ES4 library object types tomorrow.
- If you have structural type annotations but nominal actual objects, the invariants preserved by the nominal type system reinforce solution 9 making it equivalent to solution 8 for this combination.
- If you have typed object initialisers (we don't expect a lot of these, but the syntax is there), ditto by virtue of the write barrier.
- If you have (legacy or not) untyped object instances flowing through the write barrier at interface boundaries, you're at least better off in terms of type checking than today's untyped world, where e.g. MochiKit must do shape testing with a few method and property probes (see isArrayLike, isDateLike in MochiKit.Base).
- If you don't want the cost of shape-checking (deep type checking) at the interface boundaries, don't use type annotations there, or provide typed library objects (factories) and promote them as better-performing than untyped objects. People use what's faster (see Array.prototype.join being used instead of += for string concatenation).

— *Brendan Eich 2007/07/20 17:44*

Decision: Solution 9, possibly with some extra syntax to allow the user to specify that a check should be done (Allen's suggestion, something like `verify x`; Lars's solution, something like `x=x`)..

More type tickets:

- Ticket 96: resolved; moved to RefImpl.
- Ticket 121: resolved; moved to RefImpl.
- Ticket 127: remains open, Cormac/Dave/Jason to discuss it.

Self types proposal:

BE: "this" is not a good type name, people get confused and it is not parallel to the meaning of `this` in object initialisers (the value syntax; `this` means the global object for global code, or the function's receiver object for function code). Cormac's `Self` or `self` lexical binding within object structural types seems best (I vote for `self`).

There may be a grand unified theory of self typing that heals the rift between bound-`this` class methods and unbound-`this` functions, but it's not in sight. The case for `this:self` is there, but weaker than the case for `arg:self` for `equals` methods, and `function():self` for `clone` and `iterator::get` methods. Cormac to explore in micro.sml.