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

held in: ***Oslo, Norway***

on: ***23th-25th of May 2007***

Time & Place

May 23-25 at Opera: Waldemar Thranes gate 98, 0175 Oslo

11:00-18:00 *(note the later-than-usual time)*

Hotels

As Brendan discovered, it may be good to stay out of the party/theater district...

- Opera-sanctioned hotels: http://www.opera.com/company/visitors
- Thon Hotel Opera (at the railway station): http://www.thonhotels.com/opera

Directions from Airport

From the airport, take the Flytoget (airport express train) to Oslo S (Oslo central train station, downtown). From there, it's bus, T-bane (subway – look for the big T), trikk (light rail), Osloferjene (Oslo ferry), or taxi to wherever you're staying. For all but the taxis, check out Trafikanten:

- http://trafikanten.no/ (click the UK flag for Norwenglish)

Note, due to construction around Oslo S, most light rail and bus lines currently do not operate from that location. Trafikanten will still tell you where you can find them, if you're so inclined.

Useful words when trying to navigate trafikanten

- *kart* = map
- *tog* = train
- *Jernbanetorget (foran Oslo S)* = the main public transportation hub (right in front of Oslo S).

Contact

- Chris Pine – chrispi@opera.com

Visiting Oslo

The 4-1-1, yo: visiting Oslo

Attendees

- Douglas Crockford, Yahoo!

- Jeff Dyer, Adobe Systems
- Brendan Eich, Mozilla Foundation
- Dave Herman, Northeastern University
- Graydon Hoare, Mozilla Foundation
- Allen Wirfs-Brock, Microsoft
- Lars Thomas Hansen, Adobe Systems
- Chris Pine, Opera Software

Agenda

- modeling generators with `shift` and `reset` (built atop `callcc`) – Dave
- review of runtime type checks – Dave
  - `ExpectedType`
  - "the bit"
- type compatibility and runtime checks – Dave and Cormac
  - merging redundant checks
  - tail calls: merge checks on the stack
  - potentially tighten compatibility (i.e., allow fewer automatic conversions) so merging can always be guaranteed
- revisit last item from minutes_feb_06_2007, which would constrain the type of the binding created by a function declaration only in strict mode
  - want to avoid strict mode inducing extra runtime checks
  - possible revision: constrain only when the function has annotated parameter and/or return types
    - otherwise backward compatible: `function f(){} f = 42` is allowed, but not `function f(this:*){} f = 42`
- All the "Proposal" and "Spec" issues in the Trac database
- Trac/process: rename milestones to something descriptive?
- Grammar for yield, let expressions, expression closures.
- Namespace-qualified property identifiers in object initialisers.
- Discuss proposal for line_terminator_normalization.
- Dictionaries. Again.
- An optional pragma that loads a compilation unit from a URI.

Some useful examples of type checking

```
class RubberChicken { /* no meta static function convert */ }

var RC : * = new RubberChicken;
type ToInt = function() : int
type ToStar = function() : *

// ------------------------------------------------------------
// EXAMPLE 1.
// A function must obey its declared return type.

function thunk() : int { return RC }
function callThunk(f : ToStar) : * { return f() }
var x : RubberChicken = callThunk(thunk)

// result: dynamic error: thunk failed to return int

// ------------------------------------------------------------
// EXAMPLE 2.
```

```
// A call to a typed view of a function must obey the view.

function thunk() : * { return RC }
function callThunk(f : ToInt) : int { return f() }
var x : RubberChicken = callThunk(thunk)

// result: dynamic error: f() failed to return int

// -------------------------------------------------------------
// EXAMPLE 3.
// Inconsistent views of return types are safe if unused.

function thunk() : * { return RC }
function temporaryConfusion(f : ToInt) : ToStar { return f }
function callThunk(f : ToStar) : * { return (temporaryConfusion(f))() }
var x : RubberChicken = callThunk(thunk)

// result: x is successfully assigned RC
```

Notes

- grammar changes
  - yield, let expressions, and expression closures should all produce AssignmentExpression as their rightmost part
  - qualified property identifier proposals
    - ...

- type system
  - strict mode prevents execution of certain programs, the rest run as if in standard mode
  - non-observable runtime changes due to strict mode allowed?
  - type relations
    - subtyping: T <: U means T can be used where U is expected
    - compatibility: T ~: U means T can be allowed by strict mode where U is expected, but may throw at runtime
    - convertibility: T ~~> U means T can be converted to U, not necessarily preserving identity
  - strict mode uses the same rules as standard, plus a short list of restrictions to do with primitives
  - strict mode can be modeled as a transformer from source program to object program
    - extra conversions inserted in object program
    - proving no conversion is necessary (String <: Object) can avoid conversion insertion
  - discussion about v to t ~~ (v is t) ? v : t.meta::convert(v) – the convert call must be magic
    - doesn't convert argument v
    - constrains return type to t
    - spec should not dictate caller vs. callee division of labor, but must say how explicit calls to meta::convert work
  - discussion of some_useful_examples_of_type_checking
    - RubberChicken example could be simplified to not return f(); just call f()
    - EXAMPLE 1 is callee check
    - EXAMPLE 2 is caller check
    - EXAMPLE 2 is just data flow without calling, so no check
  - what are the checks?
    - we've said conversion (to), so convertibility is the relation

        - but convertibility subsumes compatibility subsumes subtyping
        - static checker and dynamic checks will try subtype and compatibility checking first
    - return type annotations vs. proper tail calls
        - could elide compatible conversions
        - stack casts or conversions? casts are idempotent
        - could just say caveat emptor
        - or require a pragma 'use proper tail calls' (syntax made up) that rejects calls whose conversions can't be collapsed
    - the one-bit scheme
        - informative model that optimizes away compatibility checks and conversions where it can check subtype relation
        - reserve a bit in each value (alternative: bit in each reference to a value)
        - functions are safe or unsafe, defaulting to safe

Semantics of type checks

- verifier uses convertibility
- automatic runtime type checks are always `to`-conversion, not `cast`
- for tail calls, elide conversions:
    - if one type is *, don't perform a check for it
    - if one type is a subtype of the other, don't perform the supertype check
    - otherwise:
        - if the tail call expression was inside a 'use proper tail calls' pragma, fault
        - otherwise just stack both conversions (in their original order)
- evaluator always performs checks regardless of strict vs standard mode
- verifier doesn't insert conversions into the AST; rather, evaluator just performs them
- but verifier still has to insert `ExpectedType`, even in "lightweight verifier" mode (i.e., standard mode)
- `meta::convert` behaves a little differently when called by the user rather than by the runtime for a conversion:
    - called as a function, its result needs to be converted according to the `ExpectedType`
    - called by the runtime, its result is just checked, to prevent transitive/infinite conversion

— *Dave Herman 2007/05/24 06:50*

Object property defaults

- object types don't currently support the usage pattern of keyword args with defaults
- made-up syntax, based on http://www.mochikit.com/doc/html/MochiKit/Async.html#fn-doxhr:

```
type MIME_TYPE_TABLE = {*}                // {'Accept': 'text/xml'}
type MIME_TYPE_PAIRS = [[string,string]] // [['Accept', 'text/xml']]

function doXHR(url: string, options: {
                    method: string = 'GET',
                    sendContent: string? = null,
                    queryString: string? = null,
                    username: string? = null,
                    password: string? = null,
                    headers: (MIME_TYPE_TABLE, MIME_TYPE_PAIRS),
                    mimeType: string? = null,
                }): Deferred
{
    ...
```

```
}
```

- but how to provide defaults?
  - mutate the object? (needs `DontDelete` and a type constraint?)
  - create a new object with the original as its prototype?
- alternative: use destructuring with new syntax for defaults?

```
function doXHR(url: string, options: {*}): Deferred {
    let { method: method = 'GET',
          sendContent: sendContent = null,
          queryString: queryString = null,
          username: username = null,
          password: password = null,
          headers: headers = [],
          mimeType: mimeType = null } = options;
    ...
}
```

- this doesn't allow typing the properties, though

Dictionary, Again

- People speaking on es4-discuss want something relatively simple
  - No weak refs, at least not in ES4 – leave for a later edition
  - No magic has/get/set syntax, has/get/set methods are fine
  - Mostly string keyed (we think); `intrinsic::hashcode` is orthogonal
  - Initialization by object literal is very important
  - Short name favored by a few commenters (Hash, Dict)
- Lars will re-work builtins/Dictionary.es within two weeks

Units (Modules)

- Much discussion of packages, provisions, locations
- Goal Doug stated: want to concatenate units server-side and download once
- Discrete principles we seem to agree on:
  - Provision satisfies requirement once (repeated requires do nothing)
  - Provisions are independent of packages
  - Provisions are separate compilation units
  - Provision names and locations should be opaque strings
- Graydon to write up a program units proposal