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

held in: ***San Francisco (Adobe)***

on: ***13th – 15th December 2006***

Attendees

- Brendan Eich, Mozilla Foundation
- Chris Pine, Opera Software
- Edwin Smith, Adobe Systems (phone)
- Francis Cheng, Adobe Systems
- Graydon Hoare, Mozilla Foundation
- Jeff Dyer, Adobe Systems
- Lars Thomas Hansen, unaffliliated
- Cormac Flanagan, UC Santa Cruz
- Dave Herman, Northastern University
- Pratap Lakshman, Microsoft
- Dick Sweet, Adobe Systems

Agenda

- tail calls and coercions (Ed, Dave, Cormac)
- error semantics (Dave, Cormac)
- numbers (Dick)
- `switch type` and `default`
- syntax of function definitions - which of the following *FunctionName* syntaxes are allowed in *FunctionDefinition*s? (ignore static v instance for now)

```
FunctionName                                          Gives meaning to...
----------------------                                -------------------
Identifier                                            f()
OperatorName                                          x + y
to                                                    x to A
call                                                  A(x), a(x)
construct                                             new A, new a
get  [no line break]  PropertyIdentifier              o.foo
set  [no line break]  PropertyIdentifier              o.foo = 10
call  [no line break]  PropertyIdentifier             o.foo()
construct  [no line break]  PropertyIdentifier        new o.foo
[from lars] dot expressions
```

- Not sure about `call [nlb] PropertyIdentifier` and `construct [nlb] PropertyIdentifier`. I don't believe we need those for builtin classes. I do think we need `call` and `construct` as FunctionNames to be in the `intrinsic` namespace, to avoid name collisions (e.g. in class `Function`). — *Brendan Eich 2006/12/12 15:49*
- Several "runtime organization" questions (graydon)
    - What *exactly* gets defined/calculated during the definition phase?
        - open namespaces in identifier-expressions?

        - "compile-time constants"?
        - tail-call status of every expression?
        - "layouts"? "fixtures"!
        - types implicit in definitions (functions, classes, interfaces)?
    - What *exactly* is the set of "values" that the runtime holds? Obviously all the "first class" values that can be assigned to variables, but also...
        - References? (object id + slot name)
        - Classes and Interfaces (distinct from their types and layouts)?
    - What *exactly* is in a layout? Which kinds of thing have layouts?
        - I think interfaces, classes, functions have layouts. Or possibly "all scopes".
        - I think layouts have (name, type, sub-layout, slot-attr) tuples.
- How are we going to work with binders?
    - We have type parameters and function parameters
    - So far I've dealt with function parameters as capturing the definition scope object, returning a closure that adds a new scope object and binds actuals to the declared names.
        - This is mostly tolerable for post-typechecked functions since the only thing you can do with them is invoke them and "tostring" them.
        - Not so good for parametric types: you want to be able to invoke (instantiate) them *and* typecheck them, which – I think – means going under the binder.
- Discuss the meaning of `intrinsic`
- Review AST types
- What can we do about Unicode?

- Conformance Test Suite - we need to come up with an approved conformance suite that implementors can use to validate conformance to ES4 (Pratap)
    - Maybe start with the Mozilla ES3 suite, clean it up, extend it? — *Lars T Hansen 2006/12/13 12:20*
- Default values for type parameters:

```
class Array.<T=*> { ... }
function fib.<T=int>(n : T) { ... }
```

- constructor initialiser syntax (see email sent Thursday)
- `for-in` loop closes generator-iterator on loop exit (normal or abrupt completion)?

Notes (Summary)

MORNING

1. Quick review of agenda items, most deferred to tomorrow or later except for Graydon's "runtime organization" questions.

Definition (Parser?) phase must produce:

- global layout (includes all imports)

- AST with following ref cells set: "all pragmas"
    - set of open namespaces
    - arithmetic modes / literal modes
    - strict vs. standard
- tail call status of each expr/statement

2. ML Implementation issue: Don't use ML closures to represent function closures because you can't look inside the ML closure.

3. Long discussion about parameterized types. No resolution, but many folks liked Cormac's suggestion that a function with a parameterized type must only be used with the type information intact.

AFTERNOON

Question about the possible type tags. This transitioned to small group work directly in ML code:

```
datatype TAG =
      ObjectTag of OBJECT_TYPE
    | ArrayTag of ARRAY_TYPE
    | FunctionTag of FUNCTION_TYPE
    | ClassTag of NAME
```

Thursday, December 14

- Resolved: 'intrinsic' is more than just a namespace. There is consensus that 'intrinsic' implies dontDelete, readonly, and final.

Notes (Raw)

1. Quick review of agenda

- First two agenda items, tail calls and coercions and error semantics, should be merged as one topic. We'll come back to this.
- Numbers (Dick would like to review the six questions he posted). Dick has an additional question as to whether ML supports decimal. Graydon is experimenting right now with SMLNJ. Dick is working on the existing AS3 compiler. We'll come back to this later.
- Switch type and default. JD: defer until after lunch. GH: It works either way, so not crucial to choose one over the other. Easiest one is default has no binding. If you want to bind, use a case, but that would lead to a FAQ about it. Jeff will write up the conclusion.
- Syntax of function def's. Deferred until this afternoon. Lars brought up option of writing a function definition and assigning it to a prototype property. Dave brings up that writing it as such, the function is an expression so it will not be hoisted, but a function declaration is hoisted. It does get hoisted in IE. Dave wrote a test script to confirm it:

```
function test(obj) {
    var x = obj.foo;
    function obj.foo() {alert('hi') }
    return x;
}
```

Dave is concerned that because the type system behaves differently depending on whether programming declarative or imperative, and mingling the syntax could be confusing. But, lars points out, that this style already exists in es3 style programming. Dave concedes that since it's already done this way, this discussion may be moot.

- Runtime organization questions from Graydon.
    - Yes, everything listed in the agenda item must be defined during the definition phase.

Bindings Class Interface Namespace
Name

Layout
AST Definition
compiled form
(constructor)
(method)
(...)

- GH: So do these things have scope?
- DH: classes are defined in packages, but packages are just namespaces. You could just place classes in a table so that you don't have to worry about scope. From the type checking perspective, we'll have a fair approx of the runtime scope. So the type checker creates its own static scope that is an approx of the dynamic scope.
- GH: I'm talking about both the definition and interpreter phase.

Example:

```
Function f() {
  Function g() {
     Var x;
  }
}
```

- CF: Can we ignore g() until f() is processed?
- GH: Then can you type check it?
- CF: yes, we can.
- DH: Definition phase takes an AST and produces a new AST and class table. Then the type checker runs and has its own idea of the static scope. That AST and class table then gets fed to the evaluator.
- JD: What about type definitions?
- DH: That should be done ASAP, maybe in the definition phase.
- JD: And type annotations? They are done at verification. So type defs can't be done at def phase.
- DH: Why not?
- JD: For one thing, there are no forward references allowed.
- DH: Example:

```
Type T = String;

Function f() {
   Function g():T {…}
 …
}
```

Type checker is assuming that all type defs have been expanded already. This suggests that substitution must have already happened.

- CF: Maybe we should pinpoint what must be done in the definition phase.
- JD: So whatever both subsequent phases need should be done in def phase.
- CF: There's a web page about all this
- JD: Yes, strict and standard modes proposal.
- JD: can't nest type defs.

- LH: seems like an arbitrary limitation that I can have a 5,000 line class but can't have multiple type defs in it.
- JD: It's a pragmatic limitation
- LH: What is that limitation?
- JD: You want to know when the value is computed. You could make it static inside of a class, but anywhere else you might not be able to compute its value.
- GH: What about nested classes?
- JD: We don't allow them. Too hard, maybe ed. 5?
- CF: Doesn't say here that def phase must do hoisting.
- JD: No, but it should though.
- CF: If the def phase transforms AST, that's going to be a lot of lines of code.
- GH: I don't mind that
- CF: Could do hoisting during the parsing phase.
- DH: How do we determine compile time constants?
- JD: It's on the proposal page.
- DH: Aside from namespaces, what is important about compile time constants.
- GH: Nothing, aside from namespaces. It's just a way to get to a namespace.
- DH: we should simplify our explanation of statically known identifiers and should not call them compile-time constants.
- GH: expr::, that's my concern (i.e. expression on lhs of '::' operator)
- DH: Algorithm is simpler than this dominate stuff. Given n , if e::n where e is an expression
- CF: So you're saying make the static analysis really dumb so it can be explained very simply.
- DH: We should differentiate between meta-objects and things that can be reflected.
- CF: Should we do that with object, have an internal representation with a wrapper that is exposed to the user?
- GH: We have that for object, it's called bindings in mach.ml
- DH: I think it's helpful to separate internal meta-information from info exposed to the user.

Definition Phase (parser?) must produce:

- A global class and interface table with layouts.
- AST with following ref cells set:
- Set of open namespaces
- Arithmetic modes / literal modes
- Strict vs. standard
- Tail call status of each expr/statement

- GH: Class def's is needed by evaluator whether type checker runs or not
- DH: I'm fine with having some sort of internal object for a class that's used by both the type checker and the evaluator.

So the top level table is mapping names of some sort to a class or interface: Name → (Cls | Iface)

- GH: In the evaluator, when I come across an inner function, I create an ML closure that binds the params, but leaves a hole for 'this'. The problem is the type parameter. When do they get bound.
- LH: I don't think it's a good idea to use functions to represent this stuff, it should be data structures all the way down.
- DH: Type parameters cannot be expressed at runtime, so the type parameter is always determined before runtime.

```
function f.<A>(x) {
   var y = Gfarg();
   function g.<A>() {
```

```
      var x = y;
    }
    return f<A> + f<int>;
}
```

- LH: what about adding var x:[A] . Can we do new A?
- JD: You can do var x:className, and do new x in AS3.
- CF: OK, so I guess new A is okay too.
- LH: what about var A = Complex; new A.
- JD: A would be typed as *.
- JD: We have class Class, which is the ctor for class objects, the base for an anonymous class that constructs class objects.
- CF: Can we have parameterized anonymous functions?
- JD: current parser allows it.
- GH: I think the current proposal says no, but I'll need to check it.
- CF: Proposes that any type parameters must be fully specified.
- CF: So for parameterized functions, we can just say that anytime you mention the function, you have to mention the type too.

So if you have f.<int>, you can only pass around f.<int> not f().

- DH: Seems even easier to require that it be a static type.
- GH: So we can already create a factory function that takes a type and returns a function. E.g.

```
function f(t:Type) {
  return function(a, b) {
    if (a is t iff b is t) {
       return a + b;
    }
   }
 }

 f(type int)(1, 2)
```

vs.

```
function f.<t>(a:t, b:t) {
  var x:t;
  return (intrinsic::typeof(x)).toString();
}

f.<int>(a,b);
```

if we allow f to be passed around, this would imply the following:

```
(foo()).<int>(a,b)
```

- DH: We should not allow this. I propose that we keep it as simple as possible. We don't have to expose all the runtime information we maintain to users. Propose that we allow only static identifiers. This means:

```
f.<int>(a,b)
```

Notes (Thursday, December 14)

Douglas Crockford, Yahoo joined the meeting today.

- Intrinsic
  - There are four topics that are relevant:
    - early binding
    - avoiding conflicts with existing code (namespaces give us this one)
    - hiding builtins
  - Lars gives example:

```
class String {
   prototype function charAt(i) {...}
   intrinsic function charAt(i:int) : String {...}
```

  - DH: So prototype is a keyword, so could you say 'prototype intrinsic'?
  - GH: Yes, but you wouldn't say that.
  - So prototype inclusive of public.
  - JD:
  - LH: So you can override intrinsic functions? If intrinsic is just namespace, you could. For example, the Object class is not final. So we'd probably want it to imply final.
  - BE: So we can make intrinsic more than just a namespace, because it should imply dontDelete and final.
  - Some unhappiness with the name 'intrinsic', but it seems better than 'builtin'.

- Numbers
  - Infinity and NaN exist in decimal. The IEEE spec suggests that anyone parsing numbers is case-insensitive, but es3 only accepts Infinity with cap I.
    - No payloads
  - No signaling NaN.
  - What happens when you store uint in an int and vice versa? LH: There should be a defined procedure for converting numeric types. Look at es3 for details about it. Q: So if you put -4 into an int, what would you expect? LH: I would expect the bit pattern that's represented by that.
  - Re: the standard interpretation section of the Number proposal says that values too large for int or uint should be double. Q is whether decimal should be the default. LH: The simplest safest way is to use double as the default for backward compatibility.
  - Discussion question 1. Assignment isn't affected by use pragma. Lars confirmed that use pragmas nest.
  - Discussion question 2. Implemented through a conversion operator. we should be consistent and say that decimal and double both truncate.
  - Discussion question 3. Boundary case: -0, this is a float (double or decimal).
  - Discussion question 4. rounding pragma is a red herring because truncation determines the result here.
  - Discussion question 5. HALF_EVEN.
  - Discussion question 6. Precision does not affect parsing. If you want precision to "kick in", add 0 to get the truncation.
  - Generally, we should try to avoid throwing exceptions.
  - DC: what's the use case for 1 != 1.0?
    - BE: Example

```
use double;
assert(1.0===1)
assert(1.00 === 1.0)
```

```
use decimal;
assert(1.00 === 1.0)
```

  - we could add methods to the decimal class that compare total ordering. Resolved, we will make 1.0 == 1 return true and 1.0 === 1 also return true. Check with Mike C. about this decision.
- DS: in IEEE 754 -0 + -0 = -0, should we do that in es4? CF: Our decimal class should implement the standard.

Notes (Friday, December 15)

- Conformance Test Suite - we need to come up with an approved conformance suite that implementors can use to validate conformance to ES4 (Pratap)
  - Maybe start with the Mozilla ES3 suite, clean it up, extend it? — *Lars T Hansen 2006/12/13 12:20*
  - We agree that we need a test suite to validate the ML reference implementation, but it is not the charter of this group to produce a test suite.
  - Various participants will run the r.i. against their own test suites

- Default values for type parameters:

```
class Array.<T=*> { ... }
function fib.<T=int>(n : T) { ... }
```

- The case for it doesn't seem to be strong enough given the complexity. It doesn't solve the problem of implementing parametric arrays using Array.<..>

- constructor initialiser syntax (see email sent Thursday)

```
> Consider the RegExp class, which has a non-nullable "matcher" field
> which is initialized with the result of the compiler (a static
> method), which also returns other information about the regexp.  I
> would like to write it like this:
>
>   function RegExp( _source : String!, _flags : String! )
>     : [matcher, nCapturingParens] = compile(_source, _flags)
>     , source = _source
>     , flags = _flags
>   {
>   }
```

- The group likes it
  - The lhs binds to the outer instance variables, the rhs to the local scope

- `for-in` loop closes generator-iterator on loop exit (normal or abrupt completion)?
  - yes

- triple quotes
  - do them as python
  - are most useful for writing text without worrying about what is embedded

- document formatting
  - each section will begin with some prose (normative and non-normative); default is normative

  - the prose will be followed by the heads of the definitions to be include (val x, fun f, and g, datatype t, etc);
  - the ml definitions should be named using a verb (verify, evaluate) followed by the name of the AST node it implements (in camel case)
  - it is necessary only to include {fun|and|var} followed by the name. The script will use this to pull in the code.

- style guides
  - no hard tabs
  - line length = 100
  - function signatures are annotated with types
  - unix line endings
  - others, we should record these guidelines on the spec page

Discussion about runtime conversions

Design space:

```
+-- convertibility = compatibility?
     |
     +-- yes. wrappers for compatibility.
     |
     +-- no. wrappers for compatibility?
          |
          +-- yes. optimize by coalescing redundant conversions
          |
          +-- no. optimize with "the bit". tail calls?
               |
               +-- stack up conversions [everyone's favorite?]
               |
               +-- discard conversions on the stack
               |
               +-- coalesce redundant conversions on the stack
```

Discussion:

- convertibility = compatibility
  - convertibility = compatibility
  - convertible types “lift” through structural types, e.g. (int → int) is convertible to (double → double)
  - pretty much have to use wrappers; otherwise a sequence of legal “to”-conversions may fail

- convertibility != compatibility
  - compatibility = subtyping and conversions from type *
  - compatibility lifts through structural types, e.g. (* → *) is compatible with (int → int)
  - convertibility includes “to” methods
  - convertibility doesn’t lift through structural types, e.g. (int → int) is not convertible to (double → double)

- wrappers just for compatibility
  - converting e.g. (* → *) to (int → int) requires a wrapper
  - could coalesce conversions if they can be proved to be redundant
  - but some conversions aren’t necessarily redundant since user-provided “to” methods might behave differently when performed multiple times

    - so not reliable as a general optimization

- no wrappers for compatibility
    - use a single bit on the edge between a reference (i.e., slots and bindings) and its referent to indicate that its value may not be of the expected type
    - on function call and return, slot read and write, perform additional convertibility checks

- stack up conversions on the stack
    - no optimizations
    - this means apparent tail calls accumulate stack when they are not convertible types
    - in order to get tail recursion, you have to make tail calls to functions that are known to produce compatible results, i.e. that do not have the bit turned on
    - this means we must require a standard one-bit semantics so programmers can rely on it for known tail-recursive behavior

- discard conversions on the stack
    - simply discard intermediate checks in a sequence of tail calls, only performing a check at the end of the sequence of tail calls
    - gives consistent tail recursive behavior
    - but doesn't perform some checks

- coalesce redundant conversions on the stack
    - only discard intermediate conversions if they can be proved to be redundant
    - not in general reliable, since many conversions will require calling "to" methods which may behave differently at different times
    - doesn't generally guarantee tail recursion

— *Dave Herman 2006/12/28 20:17*