Minutes of the: Ecma TC39-TG1

held in: Newton, MA (Adobe)

on: 19th - 20th October 2006

Attendees

- Jeff Dyer, Adobe Systems
- Graydon Hoare, Mozilla Foundation
- Steven Johnson, Adobe Systems
- Edwin Smith, Adobe Systems
- Brian Crowder, Mozilla Foundation
- Douglas Crockford, Yahoo!
- Cormac Flanagan, UC Santa Cruz
- Paul Betlem, Adobe Systems
- Dan Smith, Adobe Systems
- Erik Tierney, Adobe Systems
- Dave Herman, Northeastern University
- Brendan Eich, Mozilla Foundation
- Chris Pine, Opera Software
- Lars Thomas Hansen, Opera Software

Administrivia

Venue: Adobe Corp, Newton MA

Agenda

Thursday (post-hoc):

- Syntax for the semantics
- ML data types tutorial
- Primitive data types and operations
- Sundry

Friday:

- Go over recent changes and unfinished proposals to get ready for republishing the wiki
- More about the abstract machine
- Lars's examples
- New proposals to debate?
- Strategy / task assignments for the next month

Notes - Thursday

Syntax for the semantics

Jeff and Dave have been putting their heads together to sketch out a proposal. The current idea is a somewhat rich SSA-like specification language with high-level control flow mechanism (including exceptions) but abstracting away from accidents of syntax in the source language (eg, all the kinds of `for` loops there are).

Something like SSA is good because:

- the metalanguage is then quasi-functional, hence more readable
- it makes all mutation in the source language explicit
- it makes order of evaluation requirements very clear
- SSA is a good intermediate form for various implementation strategies
- it's readable (more so than the `goto` style of old)
- it might make it easier for the compiler writer to see what's compile-time and what's run-time

The metalanguage exceptions can be used for modelling most control flow: break, continue, and throw, at least; whether return should be modelled this way is open(?).

Cormac: what's called "Loop" in the current draft is what Scheme calls "recur"? Who will like this except from Scheme programmers?

Dave: The idea is to express the requirement that SSA "calls" to the "Loop" must be tail-recursive. And it's nice not having to model state in the metalanguage.

What do we think of the notation? Will developers read it? Should developers be able to read it, or should we just rely on David Flanagan (http://www.oreilly.com/catalog/jscript4/index.html) being able to read it? Now we have `let` and left-arrow, which distinguishes it from the source language; this advantage is also a hindrance to "average" programmers (who will consider it alien).

Maybe left-arrow should be "=" in a funny font. (But it's hard to email a funny font.)

Brendan: The primary role should be to assist implementors to create interoperable implementations. (There are secondary goals, too, but.)

Dave: The "loop" is not that alien. As the name implies, it's just a loop with a funny "do it again" syntax.

Jeff: This is primarily not about accessibility, but about implementability – the spec must give some guidance in this regard.

Brendan: Explicit is good. The E4X spec is "intuitive", but wrong. Being more explicit reduces the scope for being wrong.

Jeff: Should we worry about hitting a wall somewhere?

Brendan: Let's worry about that when we get there.

Discussion: We want to factor primitives of the abstract machine, this aids readers and implementors. We should have `ToBoolean` as a primitive (even if it is in turn defined in terms of some abstract `Convert`), because this aids understanding.

Metalanguage syntax: parens or not around calls to metafunctions? Certainly calls to metafunctions should be distinguished syntactically from calls to ecmascript functions. We can use fonts, but that's a problem for transmitting things through ASCII (email); uppercase or sigils might work. And calls to ecmascript functions need to distinguish between nontail and tail calls anyway, probably with some `call` and `tailcall` keyword.

Jeff: why do we have curly braces when we have indentation?

Dave: the braces are used for sequencing and value generation: the last operation generates a value. And `let` only appears inside braces.

Decision: The consensus seems to be that indentation is OK (except that it's sensitive to email bugs (like Python)).

Doug: Maybe we should keep the bullets to make it visually distinguished? (Again, an ASCII problem.)

Jeff: Do we need to have a keyword for generating values from an expression? (Proposals: `return`, `result`, `yield`, `produce`). Consensus seems to be that we don't benefit from this, it doesn't actually add very much.

Doug: We're drifting awfully far away from the source language, is this good? Should we not instead use a kernel ecmascript to describe the full language?

Dave: I'm concerned about confusing the reader, who has to mentally separate layers of language, whereas with a clearly different metalanguage this is less of an issue.

Brendan: I like an expression language.

Cormac: now we're defining two languages... maybe this is the way to go, but it seems we could benefit from using an existing (meta)language.

Several people want to use a subset of ML... but then the question is, how do we model store, continuations, objects? And what are the impacts? Do we end up using encoding tricks that obscure the semantics? What about type soundness proofs? Do we gain anything significant?

Graydon's ML translation of the `while` fragment:

```
let Eval while_loop (e : parenloop expr) (s : substatement) labels =
  try
   let rec L result =
     let v = Eval e labels in
     let x = Read v in
     let b = Convert x Boolean in
     if Not b
     then
       let current =
       try
         Eval s labels
       with
         Continue lb when inlabels lb labels -> nothing
       in
       let next = (if current = nothing then result else current) in
       L next
     else
       result
   in
     L nothing
  with
     Break lb when lb in labels -> nothing
```

Decision: we'll try using an ML subset rather than the ad-hoc proposed language.

Which subset? The options are really Standard ML and O'Caml. We could pick a subset that works in both dialects.

Digression: Tutorial on ML data types

Sum types:

```
 type color = Red
            | Green
            | Blue of int
```

(The values of color are Red, Green, or some shade of Blue).

Product types: anonymous and named:

```
(int * int)
{ x: int; y: int }
```

Aliasing:

```
type k = (int * int)
type e = {x: int; y: int }
```

Now we do:

```
type color = Red of e
           | Blue of (int*int)
```

Sum type constructor names (`Blue`) need to have initial capital letter, and they need to be unique (can't have Red several places).

Record types must be distinguishable: can't have two types e and f that look exactly the same.

There's no subtyping in Standard ML; O'Caml does have something here (but some debate about what exactly...)

Construction:

```
(2,3) is of type k
{ x=10; y=20 } is of type e

[]    is same as the word "nil" (?)
x::y  conses an element x onto a list y
```

Binding:

```
let e = Red in ...
let e : color = Red in ...
```

Destructuring:

```
match e with
  Red -> ...
| Green x -> ...
| Blue (a,b) -> ...
```

Type variables / parameterized types:

```
type 'a color = Red
              | Green of 'a
              | Blue of (int * 'a)

type 'a 'b pair = ('a * 'b)
```

Here, `int int pair` is the same as (int * int) in every regard.

Instantiation:

```
let e : int color = Red in ...
```

Built-in types:

```
type 'a list = 'a :: ('a list)
             | []

type 'a option = Some 'a
               | None
```

Reference types: `ref` is a keyword and used both as type name and value constructor:

```
type 'a ref = ...

let x : int ref = ref 10 in
    print !x  (* print the value *)
    x := 20   (* update it *)
```

Some sugar for this:

```
type foo = { mutable x: int; y: int }  (* data type with mutable field *)
```

Suppose foo is defined. Then because there is uniqueness of data types (see above) the x is mutable in the following example:

```
let k = {x=10 ; y=12} in
  k.x <- 12
```

Immutable update:

```
type k = { x: int; y: int }
let e = { x=10; y=12 } in
  { e with y=20 }
```

Primitive: call/cc

Discussion: explicit or implicit stack in the abstract machine?

An implicit stack is easier: save machine state in local variables on nontail call; restore it on return.

An explicit stack has a ccontinuation register in machine; there is explicit construction of a new frame on this stack on nontail call, with loading of values from the frame on return. This is more work / hair / detail; we duplicate existing machinery.

But an implicit stack makes yield much harder to implement in a transparent way. We may need to use call/cc or some generator abstraction, or a completion type (effectively CPS).

So something like this:

```
function count(n) {
  for (let i =0 ; i < n ; i++ )
    yield i
}
```

is rewritten as

```
function count(n) {
  return generator( function (result) {
                      for (let i =0 ; i < n ; i++ )
                       result(i)
                    } )
}
```

where `generator` hides the magic of `call-with-current-continuation`.

The problem now is that ML (be it SML or O'Caml) does not have `call/cc`. SML/NJ does, and there are patches for O'Caml that implement it (on some platforms), but now we're running up against limit of the specification language.

Completion types seem hard, because they tend to leak into the spec. We really want to contain this inside an abstraction.

Decision: we'll bite the bullet and use `call/cc`. This will require a custom version of O'Caml; we'll need to put up binaries for this for people (including us) to use.

Primitive data types of the semantics

We should define the data types as ADTs: sets of operations on typed but opaque data.

Multinames: can be modeled either as set of open namespaces on the abstract machine along with a simple name, or as a data type that captures the open namespaces at compile time and is passed around at runtime and is used alone to look up properties. What's better for the spec?

Ed: the set of open namespaces mirrors the environment, so can in principle be maintained as part of or in tandem with the environment. A compiler does that; our definitional interpreter could do this.

Digression on namespaces

Principles at work:

- You don't want code to change meaning when you add annotations
- You don't want real ambiguities to escape uncaught

Thus if you have:

```
class A { ns1 var x; }
class B extends A { ns2 var x; }
function f(o) { use namespace ns1, ns2; return o.x; }
f(new B)
```

There is no ambiguity, and annotating o does not change this fact. (I'm going to get `ns1::x`, and the reason for that is to avoid introducing ambiguity with annotation: if you annotate `o` as type `A` you definitely want to find `ns1::x`. You want the static types and static namespaces to correlate, to pave the way for compilers.)

However, there is an ambiguity in

```
class B { ns1 var x; ns2 var x; }
function f(o) { use namespace ns1, ns2; return o.x; }
```

Ditto for the structural counterpart.

(Long discussion on cost of this, but agreement in the end to stick with the algorithm that Waldemar defined, which basically searches for the top of the hierarchy for matches.)

Primitive data types of the semantics II

The machine

Dave: we should avoid using mutation on the semantic data structures except where it is implied by ECMAScript. This means eg that the machine state should be updated functionally, not by mutation.

Doug: why is `this` seen as a part of the machine and not part of the enviroment?

Lars: ES3 implies this.

Doug: but it's broken in ES3... we want inner functions to inherit the this from the outer function.

Brendan: we agreed to fix that, it's a little subtle:

```
   function outer() {
     print(this)
     return function() {
       return this;
     }
   }
   outer()() => global
   o = { foo: outer }
   o.foo()() => o
```

Doug: Then it's natural for `this` to be in the environment, not in the machine.

Lars: Sure, using a private property name to look it up.

Jeff: Anything else that goes into the machine than the environment? The ML machine state certainly is in there, but right now it looks like a classical interpreter.

Lars: there are static things, like open namespaces and the rounding mode/precision settings. They influence whether we use multinames or have a context for primitives?

Dave: these are all static things, they could be used for rewriting all the code in a prepass. Or they go into the environment.

Lars: Or they could be passed around as a package.

The key seems to be that these things are all environment-like, they are all statically scoped, and at least namespaces are chained in the same sense that environments are – you need to find multiple values. And for the numeric type pragmas, they may shadow some parts and other parts not. Perhaps what's really here is that these values are parts of an attribute set for each `Block` that needs to be activated when the block is activated, but there's no searching involved at runtime. So we're back to Dave's prepass, with annotation happening at an intermediate level.

Jeff: let's put it all into the environment for now.

Environment

Dave/Cormac: updateable environment through `setVariableName` sort of unusual but not really a problem.

Objects and properties

The `getClassName` is the one from ES3.

The `getPropertyValue` should operate only on the object, not on the prototype chain.

Cormac: "property" vs "slot"?

Brendan: "slot" is internal value storage. There may be properties without storage (their values are computed every time).

Cormac: worried about overabstraction here...

Lars: I think that for the moment we are looking for primitives, abstractions, etc. It is useful to use functional abstraction for the moment, at least, and perhaps remove some detail once things are clearer.

There needs to be a type property on objects. What does this point to? Some internal instance type, what ActionScript calls the Traits object (which is immutable).

```
ClassProto
     ^
     |  "prototype"  "constructor"
   Class <-----------> Proto
     |                   ^
     +---+   ...         |
         |   /           |
         v / [[Super]]   |
       Traits            |
         ^               |
         |               | [[Prototype]]
         +---------- Object
             [[Type]]
```

Here the Traits object is entirely internal, it does not exist in the language and can't be expressed in any way.

Tentatively we name it `TypeDescriptor`.

Sundry late-Thursday issues

New proposals?

Monotyped arrays (for byte arrays, say – and a `byte` type). People needing this now uses strings and `charCodeAt` but this is not very good.

The fear is that if we don't do it now, then we won't be able to do it at all. But we're way past the proposal deadline.

Source code control

Mozilla will host a CVS repository for the definitional interpreter. It'll be visible to others, but no big deal. We need to deal with the licensing issue, find a copyright that works for everyone (and for ECMA, and for ISO).

Discussion - Friday

Lars's example (expr evaluator)

Dave: Should binding objects be real ECMAScript objects with all that baggage? There are pros and cons, notably all the machinery we have to be careful to avoid for looking things up in lexical ribs.

Lars: eval kills a lot of clever optimizations here – it can introduce new variables that are deletable, for example. And we can create "plain" objects for internal use.

Dave: Is it good to be destructively updating the binding object?

Lars: see above; eval gets you.

Cormac: Do we really want both a reference implementation and a spec? Nice if we get both...

Jeff: It feels a little alien but is probably a matter of getting used to.

Brendan: Software engineering issues are probably important to make it readable and good.

Jeff: Can we define some sort of transformer to make the ML more readable? (Discussion – probably removes some of the benefit of using ML in the first place.)

Steven: Let's collect some ML references on the wiki, for common use.

We must expect to try and fail a few times. We don't have a budget for this. We need to pin down the DNA fairly soon but before we are done with that it's hard to throw a lot of people at the problem. So that's on the critical path. We should worry about expressiveness, not whether it compiles right now.

Brendan: let's cover some hard cases today and see how far we get.

What are the hard cases?

ASTs

Graydon works on generating ML data types from AST nodes from Lars's java implementation.

Recent changes

- Destructuring assignment: just some notes about things that need to be cleaned up a little. There is no problem vis-a-vis AS3's annotations (which look like array literals) because they don't appear in assignment statements
- Multiple compilation units: now looks like a proposal (but it still seems unfinished)
- Getters and setters: the `call` proposal probably does not have a use case and goes away. Objects with getters and setters can be typed; the argument of the setter and the return type of the getter must conform to the type declared for the property that has getters/setters.

Proposal review

- Grammar: The grammar will contain a mapping from concrete syntax to the AST structures defined in ML. This mapping will also imply the necessary annotation of some structures during parsing
- Type system syntax: Some ML structure... falls out from the AST definition. The type checker evaluates the ASTs
- Abstract machine and syntactic semantics: Goal for today is to have decent first cuts of these (syntactic semantics pretty much goes away because we're using ML). Perhaps also a style guide, and a description of the subset of ML we're using (if we're using a subset). A better name for "abstract machine" might be "reference evaluator"
- Built-in objects: we should just use ECMAScript 4 with some escape clause to a "native" language (eg ML).
- Descriptive prose: we want a simple summary / commentary of what it is that the code defines, connected with the code in some way.

Digression: Literate programming?

How do we keep the prose and the code in sync? Literate programming does not seem like a workable idea, but with the code under SCM and the text on the wiki we're afraid of losing sync. Maybe it's not a big deal provided the commentary is fairly high level. We would like to cross-link from SCM to wiki and back, or make the wiki page include the ML into when it's displayed.

Proposal review II

- Verification: describe this in a separate phase (with separate ML code). Some code will be similar to the regular evaluator
- Multiple compilation units: Still work to be done; Brendan thinks introducing an object in the scope chain may not be the right thing (and does not describe what AS3 currently does). Perhaps move this from proposal to clarification / guideline status.
- Resurrected eval: We drop it.

- Slice syntax: it’s good, and it’s in.
- JSON is in, though it still needs some work
- Trim should use the same whitespace set as regexes and the lexer generally.

We are 100% happy and break for lunch!