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

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

on: ***21st – 23rd February 2007***

Attendees

- Francis Cheng, Adobe Systems
- Jeff Dyer, Adobe Systems
- Dan Smith, Adobe Systems
- Pratap Lakshman, Microsoft
- Brendan Eich, Mozilla Foundation
- Brian Crowder, Mozilla Foundation
- Graydon Hoare, Mozilla Foundation
- Dave Herman, Northastern University
- Douglas Crockford, Yahoo!
- Cormac Flanagan, UC Santa Cruz

Agenda

- Pratap's Browser Profile proposal.
- Outstanding items from past meetings and phone conferences.
  - The late-breaking enum proposal.
  - The `&&=` and `||=` operators need to be included in the spec.
  - We should re-export the wiki this week, or by the weekend.
- Don't forget these issues raised during discussion:
  - Should package initialization be atomic, in the sense that a throw from package code rolls back all effects?
- Reference implementation work.
  - How do we express types in the verifier / evaluator
  - Resolved: `function to C!(v) {...}` is allowed in nullable class `C`, and return type annotation on `function to` declarations are not allowed.
  - Resolved: type parameters are invariant and you can't do anything with them in generic classes or functions except:
    - annotate declarations with them;
    - reflect them into runtime with `(type T)`, e.g.

Reference Implementation

Changes to `TYPE_EXPR`:

- `NominalType` is changed to `TypeName`, which represents any unresolved type name
- `FunctionType` is now a collection of type expressions rather than having extra irrelevant details
- `InstanceType` is new
  - represents class or interface types
  - possibly parameterized
- `TypeName`s get resolved into either structural types or `InstanceType`s

  - could potentially be forward references
  - definition phase can't resolve forward references
  - so `TypeName`s are resolved in the verifier

— Dave Herman *2007/02/22 10:25*

Function types with initializers

Function arguments may have default values. So we need a syntax for type expressions to denote this.

- Dave's wacky GNU-man-page-inspired idea: `function(int [, string, boolean]) : void`
  - blech. one character-transpose and you've got `function(int,[string,boolean]) : void`
  - absurd in the base case of all args having default values: `function([, int, string, boolean]) : void`
- nullability? no, doesn't work
- Brendan's idea: `function(int, string= , boolean= ) : void`
  - follows the principle of type syntax reflecting expression syntax
  - no new special funky Unicode sigils (heh heh heh)
- allow a placeholder like `?` or _ after the = sign? nah, it's dead wood.

Resolved: Brendan's idea is good.

— Dave Herman *2007/02/22 15:23*

Further resolved: function types must not include formal parameter names. We had been thinking of allowing them optionally, for documentation reasons. And the reference implementation required them, but it will be fixed to forbid them.

— Brendan Eich *2007/02/22 15:51*

Parameterized typedefs

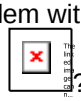
- we had said we didn't want to have to close type definitions over their type parameters
- but now we can't express the `IteratorType`
- we have (roughly) the following type, which is more or less nonsensical:

```
type IteratorType = {
    next: function.<T>():T;
}
```

- the type we need is:

```
type IteratorType.<T> = {
    next: function ():T;
}
```

- but now it appears we can't express this type
- Graydon: just go to an interface instead?
  - Brendan: long-standing decision to avoid imposing nominal type system on iteration protocol;
  - Lots of existing ES3 objects can be iterators already, expensive to wrap in `class` clothing.

- Dave: or allow parameterized typedefs?
- Dave: or allow parameterized record types?
- Brendan: we have function, class, and interface closures – is the problem with record, array, and union closures due to a difference in kind, or just because we got tired  ?

— Dave Herman *2007/02/22 16:45*

Primitives, Yet Again

- See primitive_types_day_2, the updates from Brendan.
- Due to web script extending `String.prototype` and expecting the added methods to work on primitive strings, we probably want `string <: String`. Can it work?
- Same for `boolean <: Boolean`.
- For `Number.prototype` extensions, we have problems: `0, 1, ... 42, ...` are all `int`, so `(42).myNewNumberProtoMethod()` won't work as before (but the other way of writing it, `42..myNewNumberProtoMethod()`, will). IIRC we agreed that the numeric types are not related, they are all direct subclasses of `Object`. But what breaks if we make `{int, uint, double, decimal} <: Number`?
- Subtyping isn't right anyway, unless we make the lowercase-named class hide its methods in `intrinsic`
  - so that unqualified method calls go up to the capitalized-name class's `prototype`;
  - in which case we still have incompatibility due to lack of "wrapping": `let s = ''; s.foo = 42; assert(s.foo === undefined);`
  - but this may be a compatibility constraint we can keep, as Jeff suggested, via catchalls.
- On reflection, the "Day 1" minutes_jan_24_2007 proposal to unify `string` into `String`, likewise for `Boolean`, looks better again.
  - It does require optimizations at least as involved as today's ES3 implementations have to do to avoid doing `new String('')` for every `''` literal.
  - But that's not so bad – question is, what further optimizations (e.g. such as AS3 does) are "hard".
- `Number` is still a problem, and (without detailed memories) I believe there are use-cases for AOP on `Number.prototype`.
- UPDATE: to be precise, what ES1-3 specify are primitive types that get wrapped by every `ToObject`, but what "Day 1" proposed was only ever (the appearance of) a `new String` instance for every literal or concatenation result.
  - Not fully backward compatible as noted above (`let s = ''; s.foo = 42; assert(s.foo === 42)` – unlike ES1-3, it's as if the primitive empty string "becomes" in the Smalltalk sense a `new String`).
  - Suppose we support primitive types via `final class string!`, etc., and model wrapping as follows:
    - `class string! extends String`.
    - The fixed methods of the final primitive class are all `intrinsic`.
    - Which implies that the standard methods of `String` are **only** `prototype`.
    - Thus we split intrinsic functionality from `String` and push it down into `string`.
    - `'hi'.charAt(0)` therefore invokes `String.prototype.charAt` on the `string` `'hi'`.
    - If you want speed and not AOP, `use namespace intrinsic` or call `'hi'.intrinsic::charAt(0)` (or in this case, use indexing).
    - To support `'hi'.nosuch === undefined` and `'hi'.foo = 42`, we use Jeff's catchalls suggestion, which simulates ES1-3's wrapping under `ToObject`

    - Code may freely use `string` and unless it explicitly uses `intrinsic`, AOP via `String.prototype` still works.
    - And (thanks to the catchalls) generic get-and-test-against-`undefined` (or equivalent) and useless sets still work.
  - Please poke holes in this.

— Brendan Eich *2007/02/23 10:12*

Alpha-renaming

There appear to be places in the language where we can't avoid the need for doing alpha renaming.

Example 1:

```
class C.<T> {
     type A = T
     function f.<T>() {
        var x : A = ... // x has outer type T
     }
}
```

- the verifier needs to associate `x` with the type of the outer `T`, so it must rename the inner binding of type parameter `T` (the type argument of `f`) to prevent it from capturing the outer `T`
- could we forbid nested `type` declarations?
- there are some useful type declarations we then wouldn't be able to express
- for example, the `type A = T` can't be lifted out to top-level, because it refers to the type parameter `T` which isn't in scope outside the class declaration

Example 2:

```
function f.<T>() {
    return (function <U>():[T,U] { ... });
}

class C.<U> {
    f.<U>() // has type function.<U'>():[U,U']
}
```

- here, the application of f to the type `U` inside class `C` requires that we alpha-rename the returned function's type parameter `U` to prevent it from capturing the other `U` (the argument to class `C`)

This latter example is especially hard to avoid. No clear restriction that we could impose on the language to prevent it.

- Graydon: optimize for parameterize classes; most important use case
- Cormac: but without parameterized functions/methods, can't do things like `List.map`
- decision between complexity budget vs. unuseful toy implementation of generics
- Dave: reflection concern – let's just prevent uninstantiated parameterized types from being exposed
- Brendan: we know how to implement it, generics are vital, let's do it
  - Just for the record in detail, I argued that generics types (structural as well as nominal, and not just function types) are vital for retro-fitting and future-proofing via structural compatibility rather than nominal subtyping. — Brendan Eich *2007/02/23 12:13*

Resolved: alpha-conversion is a must.

— Dave Herman *2007/02/23 11:37*

Another example:

Example 3:

```
type A.<X> = function.<T>():[X,T]
class C.<T> {
   var a : A.<T> = ... // should be function.<T'>:[T,T']
}
```

Dave notes that example 1 is an example of what's known in the hygienic macro community as "referential transparency" (an unfortunate choice of terminology, not unrelated to the general concept but not exactly the same thing, either), and example 3 is an example of the "hygiene condition".

— Dave Herman *2007/02/25 04:58*

Further resolved: adopt the "more general and composable" form of type parameters: a 'TyLam(["X"], tyexpr)' constructor for type expressions that generally represents parametric types.

Fallout:

- Parametric function, class and interface closures lose their type-parameter list and "isApplied" flags
- They gain, instead, a 'type' field that begins with 'TyLam'
- As before, operator 'new' refuses to work on a class closure with a 'TyLam' in its type.
- As before, function-call refuses to work on a function closure with a 'TyLam' in its type.
- The 'type-apply' operator '.<tyExpr>' at runtime strips one level of 'TyLam' off its operand (closure) and returns a new closure with the stripped type and a new 'TypeFixture' binding in the closure.
- All type definitions can be parametric. If you say 'type T.<X> = Z.<X>' you have produced binding for 'T' with type 'TyLam (["X"], TyApp(TyName("Z"), TyName("X")))'
- The verifier will have to alpha-rename type terms in order to compare them.
- The interface between the evaluator and verifier will need to provide type environments for any type-compatibility questions it wants answered.

Some further concern was raised about the visibility of types defined *inside* classes. The consensus view of this is the following:

- Types defined inside classes cannot be seen outside classes. There is no 'TyMember' constructor for type expressions. 'C.T' is not meaningful when 'T' is a type.
- Types defined inside classes may refer to type parameters of the class, but all types defined inside classes are *invisible* to static initialization code for the class. If you want a type to be visible to static initialization code, you must move the type out of the class.

— graydon *2007/02/23 14:23*