Notes - 4/18/2014

Wednesday, April 9, 2014 2:18 PM

Agenda

- Update on ES6 module discussions
- Module resolution algorithm
- __extends and its optional inclusion
- ES6 alignment:
  - Symbols and the type system improvements to support them
  - Proxies and the type system impact (if any)
  - ES6 classes
    - How do we want to handle 'extends' of an expression?
    - Should get/set be reflected in the type system?

Update from ES6 committee discussions

There's a potential new ES6 module syntax change under consideration. Rather than using the 'module' for loading the default module, the committee is considering a simplification that uses import. This would give a bit more streamlined syntax. Example:

```
import $ from "jQuery";  // export default loader
import {ajax} from "jQuery"; // exported member loader
```

You can get to a reflected object representation using Module. The result has getters that can reach into the reflected module.

Questions that came up in discussion:

- How is the export default reflected in this object?
- Is there a readable reference for the ES6 modules?
  - There's an annotated implementation (https://people.mozilla.org/~jorendorff/js-loaders/Loader.html)

Schedule-wise we could start implementing in earnest soon. July still sounds like "pencils down" for the ES6 spec, which is only a couple months away. Expect that most of the syntax is pretty settled at this point.

(INTERNAL-ONLY: Plan for Chakra for ES6 modules is post-Threshold)

Updates from the ES7 front

Interesting proposals coming down the pipeline that we may want to explore as they mature:

- Decorators/annotator straw man
- async/await
- ParallelJS

Module resolution algorithm

We currently we have a pseudo-node process that works as something of a hybrid of AMD nor node, but it doesn't allow any specialization for the type of things that Require, Common, and other module loaders might want to do.

One user wants to extend our process:
https://typescript.codeplex.com/SourceControl/network/forks/kayahr/typescript/contribution/6570

One possibility is to create a pluggable resolver architecture that lets people plug into the compiler a way of resolving that's flavored for Node and AMD rather than the pseudo-Node approach we use now.

Action item: Let's research this more deeply to see what a pluggable loader would work like.

__extends

The __extends function is one of the only places in the compiler that we inject code into what the user provided. Would be nice to be able to suppress this and allow the user to assume the responsibility of providing their own. We would also suppress in the case of ES6 output, once that's added.

Proposed Solution: Check if there is *any* global symbol __extends in the user code, even if it's a declare symbol. If we see it, we don't emit our own __extends. We may want to check that the type of the provided __extends matches the shape of the one we will expect to call.

Aligning with ES6

Symbols

How do we want to describe symbols in the type system?

We need to introduce a new basic type for symbols which can be used as a new kind of key into objects.

Take as an example [Symbol.iterator]

```
var x = {  [Symbol.iterator]: value };
```

Where does the Symbol.iterator get the value that's used as the key? Is this something we track in the type system? Are each symbol type fresh? We could introduce a notation 'new Symbol' as annotation for creation a fresh new symbol type.

```
declare var Symbol: {
    iterator: new Symbol;
}

var I = Symbol.iterator;
myObj[i];
```

How does this relate to existing type rules? Is a new fresh type like creating a new enum on the spot?

The indexes would gain a new 'symbol' indexer. We would use syntactic sugar to be able to reference the symbols through the common ES6 name (which we would have typed ahead of time in lib.d.ts for the common symbols):

```
interface Foo {
    [x: string]
    [x: number]
    [x: typeof Symbol.iterator] : FooBar
    [Symbol.Iterator]: FooBar // short form of the previous line
}
```

We need to continue investigating to better understand the implications of having symbols be unique types.

Min bar for the design: for-of needs to have the item type strongly typed.

Luke's gist for talking to (https://gist.github.com/lukehoban/ebb6165511480841dcfc):

Here's what the Symbol API would look like:

```
interface Symbol {
 // branded with [[SymbolData]]
}
declare var Symbol: {
 new (description: string): Symbol;
 iterator: Symbol;
}
```

And here's what a common usage might look like:

```
let fibonacci = {
 [Symbol.iterator]() {
  let pre = 0, cur = 1;
  return {
   next() {
    [pre, cur] = [cur, pre + cur];
    return { done: false, value: cur }
   }
  }
 }
}
```

Object types now need to allow properties whose name is not a string, but instead is a Symbol. However, the identity for lookup of these names in the type system cannot be based on runtime values as would be done in the JavaScript semantics. So instead, the identity likely has to be based on TypeScript symbol identity - as in Symbol.iterator resolves to a particular symbol, and that symbol's identity is the key that is used for this property.
That would allow this to work:

```
fibonacci[Symbol.iterator]().next().value // works
```

But not this:

```
var i = Symbol.iterator;
fibonacci[i]().next().value // doesn't work
```

That lack of aliasing may be okay for common cases. It's unclear if there is any option for supporting these aliased cases.
There's also a question of how to know when a looku is a symbol name vs. a dynamic string lookup. The lookup rules can presuably know that if the type of the key expression is Symbol, then it is a symbol lookup. This would

work in the example above, but would bake in Symbol to the language. Unlike normal foo[bar] lookups, if bar resolves to a Symbol, it is presumably an error if foo does not have a member keyed by that Symbol.
For diagnostics, errors could use the name of the TS symbol used as the key, for example:
({})[Symbol.iterator]
/// Error: The property symbol 'Symbol.iterator' does not exist on value of type '{}'.

From <https://gist.github.com/lukehoban/ebb6165511480841dcfc>