Notes - 6/6/2014

Friday, June 6, 2014 10:38 AM

Agenda:

- Updates on compiler adoption
- Update on the ES6 process
- ES6 features for TypeScript vNext
- Symbols

Spread

Arrays

[1, 2, …items, 3, …moreItems]

Type check: BCT

Codegen:

[1,2].concat(items, [3], moreItems)

Start with spread:

items.concat([1, 2], …)

Functions

Codegen:

f(…items) => f.apply(void 0, items) // strict mode. Otherwise, 'this'

obj.foo(…items) => obj.foo.apply(obj, items), but only want to evaluate obj once, so we inject an:

```
__apply(obj, method, args) {
  obj[method].apply(obj, args);
}
```

__apply(obj, "foo", items);

CoffeeScript introduces a fresh var, hoists it to top, and uses it to prevent re-eval of 'obj'

Local vars could be done inline

Typecheck:

f(1, 2, …items, 3, …more) : f (T, T, BCT (…items, 3, …more))

Issues is that this would prevent potentially valid code from passing typecheck.

Key idea is that we would be checking the arity that we know as well as the BCT, knowing that the spread arrays could be empty.

Do we want to be able to spread array-like, like the array-like objects that come from the DOM? If we do, we could use the CoffeeScript approach. We could tell arrays and array-like apart statically in the type system.

What if it's 'any'? We don't know, so we have to codegen a slice.

Concat doesn't take array-like, it takes arrays only. .apply should take array-like.

Destructuring

var [x,y] = myFun();

Codegen:

```
var __a = myFun();  // fresh var __a
var x = __a[0];
var y = __a[1];
```

Type annotations:

var {x: myX, y: myY}: {x: …; y: …} = myPoint(); // {x: myX} is a ES6 destructuring with renaming

Symbols

Background

The way ES6 symbols work, in truth it's actually difficult to use them as a simple 'private' member. Because of this, it's quite possible that people will generally only be using the built-in ones like Symbol.iterator rather than creating new ones. Let's focus on a lighter-weight tooling of this scenario that will effectively type the builtins and would require the users to do a little extra work to type their own unique symbols rather than complicating the type system for a scenario that may not be core.

Proposal

Taking another approach, which doesn't track uniqueness of the type automatically

1. symbol() just returns Symbols
2. Symbols that you want to be unique, you need to create a unique type for each

```
// Solution #1 = use classes

class SymbolBase { }
class IteratorSymbol extends SymbolBase { private uniqueIteratorSymbol; }

module Symbol {
    export var iterator: IteratorSymbol;
}

// Solution #2 = introduce privates into interfaces (to prevent trying to instantiate
classes)

interface SymbolBase { }
interface IteratorSymbol extends SymbolBase { private uniqueIteratorSymbol; }

module Symbol {
    export var iterator: IteratorSymbol;
}
```