Notes - 7/11/2014

Monday, July 7, 2014 11:53 AM

- ES6 design status
- Imports and removing code during codegen
- Widening rules and return types
- Untyped calls

Dev14 TypeScript ES6 Design Status

Here's an update on our progress for locking down designs for the ES6 work we're doing for Dev14. We've talked about spread and destructuring recently, and the notes from the whiteboard are below. I've also added some notes to get us started on class expressions.

1. Overview

Feature	**T-Shirt Size**	**Downlevel**	**Design Complete**	**Dev14 Commit**
Arrow functions	XS	Y	Y	
default function params	XS	Y	Y	
rest parameters	XS	Y	Y	
spread call operator	M	Y	In Progress	
spread array operator	M	Y	In Progress	
class expressions	L	Y	In Progress	
classes extending from expressions	L	Y	In Progress	
Template Strings. Type is based on the usage of each tagged template funs	S	Y		
Destructuring (L = deeper typing changes to support tuples)	M/L	Y	In Progress	
Use privates in ES6	M	N/A		
Comprehend well-known symbols	L	N/A	N	
const	S	N		
let	M	N		
computed properties	M	N		
For…of loops	M	N		
Iterators (mostly for free based on earlier work)	S	N	N	
Generators (yield)	M	N	N	
RegExp "y" flag	XS	N		
Modules	XL	??	N	
Standardize internal modules	XL	??		
WeakSet (just the lib.d.ts interfaces)	XS			
Proxy (add apis to lib.d.ts with new Proxy returning its target type)	XS			
Block-level function declaration	??	??		??
Promises (add apis to lib.d.ts)	XS			

Notes:
- For..of arrays we could do so efficiently. For..of general doesn't have a downlevel.
- What about sparse arrays and for..of? Would we need to do for..in?
    - Can we do this efficiently?
- What about things like Abstract classes?
- Block-level functions sound tricky as there is a more complex hoisting rules, where function expression in a loop my reclose over their variables
    - Scoping of name is like let (in strict-mode)
    - In non-strict mode, according to Luke there's a looser set of semantics that works closely to current browser implementations
    - Can TS get away with block scoping? All the time or just in strict mode?
    - Can we assume that the function is at the function-level scope, but it's an error to be used outside of the block it's defined in. This helps us catch redeclaration issues that would be unsafe codegen.
    - We already restrict example where eg if and then can't create the same function
- Do we want to be doing more in strict mode to catch more classes of errors
- Can we do a cheapo minification to get started?
-

2. Spread Arrays

Example

```
[1, 2, …items, 3, …moreItems]
```

Type check:

BCT

Codegen

```
[1, 2, …items, 3, …moreItems]
```

Becomes:

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

If the array starts with a spread, you can concat off the spread:

```
[…items, 1, 2, …moreItems]
```

Becomes:

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

Questions

Q: What if it's 'any'? We don't know, so we have to codegen a slice. One possible issue is that `.concat` doesn't take array-like, it takes arrays only. This won't be a problem below because `.apply` should take array-like.

3. Spread Arguments

Example

```
 f(1, 2, …items, 3, …more)
```

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))
```

Issue 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.

4. Class Expressions

Example

```
class B extends class C {
 constructor() {
   this.y = 4;
  }
}
{
  constructor() {
   this.x = 3;
  }
}
```

Codegen

Should be able to unpack the class expressions and codegen as we do now. We can pass the resulting expression as the parameter to the codegen'd constructor function as we do already.

Typecheck

The type of the expression follows the same rules we have for class declarations now. The expressions create constructor functions as before.
In the extends position, eg "class A extends B", `extends` the extends clause must be an expression, or a Generic NamedTypeReference. If it is an expression, the `extends` clause uses the type of the expression. This type must have a constructor signature.
This would allow it to be compatible with the current system and be more flexible with arbitrary expressions, as we would treat class name TypeReferences as constructor functions we will get the type of.

Questions

Q: If C is visible outside of the scope, then we need to lift out and make visible. Else, we can just translate the expression and put it into the call to creating B's constructor function
A: Traceur says it's not in scope outside of B's expression. Looks like we may be able to translate the expressions in-place as a call to the constructor function

Q: If we base the typecheck on constructor signature, what about overloaded constructor signatures?
A: We require all overloads to return the same type

Q: What about generic constructor signatures?
```
        interface I {
    new<T>(): T;
 }
```
A: We would disallow generic constructor signatures. These can't arise as the type of a signature of a class's constructor, so we'd follow a similar restriction.

Q: What are the scoping rules of class B<T> extends class C { /* can use T here? */ } { ... }

Q: There is an ambiguity for parsing these as expressions (less than B, greater than, etc etc)
A: We would need to tweaking the parsing precedence to ensure the correct parse is checked first.

Additional questions (from Luke):

Annotations
The alternative to a declarative annotations system is to wrap class declarations in function calls which modify the class in some way. This is a case where you may legitimately want generic class expressions. Generalizations of this could probably even create the situation Cyrus and Jason were discussing where you could want a generic class in an extends clause. However, I don't think this works because TypeScript doesn't allow generic variables, so you can't bind the `List` name to the generic class produced by this, even though the typing ES6 and the typing seem to make sense.

```
function intercept(C, logger) {
  for(var m in C.protoype) {
   C.prototype.m = function() {
    logger(this, arguments);
    m.apply(this, arguments);
   }
  }
}

var List = intercept(class <T> {
  items: T[];
  constructor(items: T[]) {
   this.items = items;
  }
  get(i: number) { return items[i]; }
})
```

Notes:

- How do we get at the type arguments?
- How do we get at the instance type?
- typeof new List<string>()? Blah
- How do we work with anonymous types?

Mixins
Even with class expression support, I don't think we'll be able to handle this because we can't express the static merge of two types (the type of the `mix` function).

```
class A {}
class B {}

class C extends mix(A,B) {
}
```

Metaclasses
Even with class expression support, we don't support classes whose instance type is a class. You can write the interfaces for this sort of thing, and can implement it with ES6 classes, but can't implement it

with TypeScript classes. Specifically, the "return class…" in the constructor below would not be legal TypeScript.

-
```
class Type {
  byteLength: number;
  constructor(byteLength: number) {
   this.byteLength = byteLength;
  }
}

class ArrayType {
  constructor(type: Type) {
   return class extends Type {
    length: number;
    constructor(length: number) {
     super(type.byteLength * length);
     this.length = length;
    }
   }
  }
}
var Int32Array = new ArrayType(Int32)
var arr = new Int32Array(100);
```

5. Destructuring

Example

```
var [x,y] = myFun();
```

Codegen

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

Type annotations

```
    var {x: myX: number, y: myY: number} = myPoint(); // {x: myX} is a
ES6 destructuring with renaming, this looks awkward
    var {x: myX, y: myY}: {x: …; y: …} = myPoint();  // Moving type
    annotation to beside destructuring left hand side
```

Type annotation for arrays proposal and questions: Notes - 4/25/2014

Imports and removing code during codegen

In a global file:

```
module M {
    export var v;
}

module N { // Only old compiler emitted
    import i = M.v; // Neither compiler emitted
}

import j = M.v; // Both emitted

module P {
    export import k = M.v; // Both emitted
}

module Q {
    module R {
        export import l = M.v; // Both emitted, no emit needed!
    }
}
```

1. We seem to always emit exported imports that reference entities, even if they are not exported all the way up the global level. This is not really consistent with our elision pattern.
2. External module imports that are unexported (e1 and e3) are never emitted. They can certainly never be referenced, but they may have side effects, so maybe we should emit them.
3. We do not need to emit type only modules (e3 and e4) because they can never be used in value position.
4. If we buy the side-effects argument from #2, then we might want to consider always emitting for ambient external modules even if they are type only. This is because a type-only ambient external module could correspond to a .js file with side effects (the side effects would never be captured in a declaration file, so it would look like a type-only module). In this case, we would emit e3 and e4, but only if ext2 is ambient (which it is in the example.

More:

RE Bug
979557

Widening rules and return types

From Jason's email:

The problem with this widening is observable in type argument inference and no implicit any:
For type argument inference, we are prone to infer an any, where we should not:

```
function f<T>(x: T, y: T): T { }
f(1, null); // returns number
f(() => 1, () => null); // returns () => any, but should return () => number
```

So after we get to parity, I propose we do the following. We do not widen function expressions. A function is simply given a function type. However, a function declaration (and a named function expression) introduces a name whose type is the widened form of the type of the function. Very simple to explain and simple to implement.

I've been told that this would be a breaking change, because types that used to be any are now more specific types. But here are some reasons why it would not be a breaking change:

- In the places where you actually need the type to be any (because there are no other inference candidates), you would still get any as a result
- In places where there was a better (more specific) type to infer, you'd get the better type. It's not a breaking change if we return what the user actually expects.
- With the noImplicitAny flag, you'd get fewer errors because there are actually fewer implicit anys

Questions:

- Is a principle of design changes going forward to not switch from 'any' to a more precise type because it can be a breaking change?

RE
Widening

Untyped Calls

In certain cases old compiler used to emit "Value of type is not callable. Did you mean to include 'new'" error when during analysis of call expression discover that type does not have call signatures and function being called was bound to constructor' symbol. This yields quite inconsistent results, for example this code will contain error:

```
class A { }
var x = A();
```

but this one - does not:

```
class A { }
var x = A(); // error
interface I { new (): I };
var c: I;
var y = c(); // OK
var c1: { new (): string };
var y1 = c1(); // OK
```

Per spec (4.12 Function calls) all these examples are correct and will result in untyped call. However I think having error here still has value because in most cases it will correctly signal about missing 'new'.

Anders: I think an untyped call should be permitted only if the type has no signatures at all (call or construct). Technically that's a breaking change, but I think it is a good one. The current behavior makes little sense and I highly doubt the change would break any real world code.