Notes - 10/29

Monday, October 29, 2012 12:03 PM

[In 41/5717 at 1:00 today.]

Agenda:

- Generics - review current status of thinking:
  - Interface
    - Assignment compatibility rules
    - Implements rules
    - Restrictions?
  - Class
    - Inheritance rules
    - Statics
  - Type argument inference
  - Type rules for generic type parameters
    - Assignment compatibility
    - What is a T compatible with?
  - Constraints?
  - Some motivating scenarios:
    - Arrays
    - Underscore
    - KnockoutJS
    - Promises
    - D3
    - Iterators/generators
    - Function.prototype.bind/call/apply/etc
    - Others?
- Others?

```
interface Foo<T, U> {
   bar<W>(t: T, w: W): W;
   bar: { <W>(t: T, w: W): W };
}
```

```
var foo: Foo<number, string> = {
  bar: function<W>(t: number, w: W) {
    return w;
  }
}
```

```
interface Array<T> {
  map<U>(f: T => U): Array<U>;
}

var _elems = [];
var arr = {
  map: function(f: any => any) {
    return this. _elems.map(f);
  }
}
```

```
var _elems = [];
var arr: Array<string> = {
  map: (f) => _elems.map(f);
}
```

```
interface Func<T,U> {
  (t: T): U;
}

var x: { bar<W>(w: W): W };
```

```
var foo2: Foo = { // This one means Foo<any, any>
  bar: function(t,w) { return "bob"; }
}
```

```
var x: {
  <T>(t: T): T;
  foo<T>(t1: T, t2: T): void;
}
```

```
Interface List<T> {
  add(t: T): void;
  get(index: number): T;
  map<U>(f: T= U): List<U>;
  first: T;
  last: T;
}

// With constraints
Interface List<T extends Sortable> {
  add(t: T): void;
  get(index: number): T;
  map<U extends Sortable>(f: T= U): List<U>;
  first: T;
  last: T;
}
```

Object Literals:

1. Generic functions

Classes:

```
Class MyList<T> {
}
```

```
var f: {<T>(t: T): T };
var g = function(x) { return x; }
f = g;
```

Rule is:

1. For assignment compatibility, erase

```
var f: {<T extends Sortable>(t: T): T };
var g = function(x: Sortable) { return x; }
f = g;

function id(x: any) { return x; }
var f = function<T>(x: T) { <T>id(x); }


var p = MyList<number>
p.static1
var x = new p();



class MyList<T> {
  items: T[] = [];
  map<U>(f: T => U) {
    var res = new MyList<U>();
    for(var I = 0; i< this.items.length; i++) {

    }
  }

}
```

Simplest:

- **No constraints**
- **Cannot do any lookups on T, cannot assign anything other than any to T, cannot assign T to anything other than any and T ("T is a branded {}")**
- **Function expressions cannot declare generic parameters**
- **Class methods, function declarations**
  - **Later: What about object literals?**

What's left:

- **Inference**
  - **Would like: Infer class type parameters from constructor calls**
- **Type compatibility**
  - **Maybe: any's can fill in for T?**
- **Try this out for 5 libraries**
-

Thoughts:

1. We understand how the types work
2. Validating that an implementation implements a generic signature is less important
3. Class List<T>

Note:

- Generics on named interfaces can be instantiated at interface name usage
- Generics on call signatures part of type

Desiderata:

1. Declare a generic interface
2. Declare a generic interface method
3. Declare a generic class
4. Declare a generic class method
5. Declare a generic function

Questions:

1. Do we need to allow "Foo" to mean "Foo<any,any>"?