Notes - 10/24

Monday, October 17, 2011 6:28 PM

- Scoping

Statics:

```
class Point(x: number, y: number) {
  function getX()  => x;

  static {
     property Origin = new Point(0,10);
     var lookup  …;
  }
}
```

Should T be in scope in static block – no.

Bug: Closure capture of

Decision: Improvement on where we were.

Questions;

1. Brands?
2. What types can you write? Type literal.
3. Mechanics of extends and implements
4. Self names

Brands

Enums: Branded. No new members over number. Bunch of statics. This is useful.

Problem with brands: There is a distinction between a class and it's unbranded interface contract. Collection<T> provides a constructor, but may also want to use it as an interface that others can implement.

Not trying to unify classes and interfaces.

Sounds like we are okay with brands

Everything declares types.

- There are

```
var x = class : Base {
 function foo(): number
}

var y = new x();
```

Problem 1: Type literals to express base class?

Let's say that we keep brands:

Modules

- import
- ///require

One question – when I try this I'm seeing things bind as shown below. Your description sounded like the "WScript.Echo(name)" case should have been yellow.

Luke

From: Steve Lucco
Sent: Monday, October 17, 2011 12:34 PM
To: Strada Design Team
Subject: consequence of design change

A consequence of enabling method bodies to see constructor parameters and locals is that within method bodies we need a simple name lookup rule that prefers properties over parameters/locals. I added that but thought it worthy of further discussion. Example:

```
class B(property name: string) {
   name="B"+name;
}

class C(name:string): B(name) {
   WScript.Echo(name);
   function f()=> name;
   function g()=> {
     var name = "hello";
     return name;
   }
}

WScript.Echo(new C("hello").f());
```

Steve

Reflecting on our conversation, there are more requirements on that class var than we talked about. The type of that var is derived from the following type:

```
type Class<T> {
   // a class must have a prototype containing the methods of T
   prototype:MethodsOf<T>;
   // new can actually return undefined; it does not need to
return T because
  // the system will return the created object if new returns
undefined
   new(/* args */):T;
```

- Classes and enums are branded
- class C: X, Y, Z
  - Mixins still can work if X,Y are classes
- Merges:
  - Compare:
    - X = A | B
    - interface X: A,B { … }
  -
- Covariant subtyping:
  - interface T1 { x; y; }
  - T1 t1 = { x: 1, y: null }
- No self names
- Anaonymous types cannot escape functions

- Algorithm for covariant subtyping

```
Var x: (x: number, y: number) => string
Var y: (…args: any[]) => string

var f: (x: number, y: number): string
var g: (x: number): string

f = g // This should be allowed, because
```

We do want to be able to type:

```
Promise<T> {
 then<R>(f: (T) => R): Promise<R>
 then<R>(f: (T) => Promise<R>): Promise<R>
}
```

From: Luke Hoban
Sent: Monday, October 24, 2011 9:59 AM
To: Strada Design Team
Subject: Strada Design Meeting Agenda - 10/24

[In 41/5731 today]

Agenda:

- Update on implementation from Steve
- De-branding of classes (first two attached)
- Import and modules (last attached)
- Simple generics
- Backlog:
  - Promises typing
  - “is”
  - Optional parameters
  - Top-level scope and global module
  - Finish discussion of expression typing
  - “await”
  - Enums
  - Foreach
  - Any -> number (what gets emitted?)
- Anything else?

```
   // this can actually also return void or undefined as long as
it has
   // the correct side-effects on its this parameter
   ():T;
}

class Point(x:number,y:number) {
   move(xo:number,yo:number):Point {
     x+=xo;
     y+=yo;
     return this;
   }
   static {
     property origin=new Point(0,0);
   }
}

// translates to

type PointClass : Class<Point> {
 origin:Point;
}
```

The inheritance pattern we are using assumes all of these requirements: the constructor must either return this or return undefined. The constructor, when used as a function must have the same side-effects as the constructor when called with new. The constructor must have a prototype member that has the methods of T but not the properties. Here’s an example of an expression that returns the base class but doesn’t meet these other criteria:

```
class Base {
   property x:string;
   function f() => x;
}

function P() {
   return new Base();
}

class Derived : P(){

}

var d=new Derived();

d.f(); // won't be there
```

This won’t work because we will try to create the prototype of Derived by new P(), but P does not have a prototype that contains f and so f won’t be there.

Steve