Notes - 11/22/2013

Friday, November 22, 2013 10:01 AM

Wrapping up the spec bugs today.

- Update from Engineering
  - Progress toward 1.0
  - Generic constraints and covariance and engineering impact (Bug #813197)
    - The key piece is that Array<T> <: Array<U> if T <: U. We can repair the hole later.
    - DONE
- Modeling non-inherited static members on JavaScript libraries
  - Proposal is to not inherit the static members added to a symbol by a module, rather only the static members inside of the class are inherited by subclasses
  - Issues:
    - The __extends function currently is general case, but would copy over non-inherited members.
      - One solution would be to copy specific members by hand
      - Or, we could just copy them over and only make them visible at design team. This is preferred
    - Having two separate types, one for C and one for the static 'this' inside of C being different adds complication to the type checking
      - We need to cost this
    - One design approach: only during inheritance do we check what is visible from classes rather than what came from a module (forming a clodule)
    - Will make work item for this
    - DONE
- Rules for recursive/circular type inference
  - Need a spec that describes what constructs are allowed to be used in a class that references itself or else the type becomes 'any'. These would then be errors with no-implicit-any.
  - Related, need to close on self-referencing initialization (Bug #827214)
  - Also related, using typeof in a recursive position for function return types (Bug #764275)
  - Taking this offline to look at examples we'd like to work vs okay with not working
  - Action item: Anders will spec
- Generic classes with constraints unable to extend non-generic classes

```
class Event {
    target: EventTarget;
}

class MyEvent<T extends EventTarget> extends Event {
    target: T;
}
```

```
class BaseEvent {
    target: Object;
}

class Event<T> extends BaseEvent {
    target : T;
}
```

  - Related, should this be allowed?

```
class EEvent {
    target: Object;
}

class MyEvent extends EEvent {
    target: any;
     x: string;
}

var x = [new EEvent(), new MyEvent()]

var z: any;
var y  = [3, z]
var y2 = [z, 3]
```

  - Proposal 1:
    - If you have no declared constraint, then for purposes of sub/super/assignment compat, the apparent type is 'any'. But, for property access, and calls, the apparent type is '{}'.
    - For type arguments, we change constraint checking to be subtype checking. This disallows 'any' for type arguments with constrained parameters
      - Investigate: how much code would it break to enforce this rule?
    - Prototype property is instantiated at base constraints or is given type 'any' if any base constraints references any type parameter in the same parameter list
    - Don't validate the prototype when doing 'extends' and 'implements' checking

    - Investigate: recursive generic constraints and removing them
    - Two solutions:
        - Stick with 'any' given to the prototypes. And then, we use assignment compatibility check instead of subtype check when a class subtyping via 'extends' and 'implements' and for index checks.
        - Give an existential type for constrained types, and then use that for the prototype
        - Questions:
            - What do we with prototype? We currently give it 'any' for all type parameters. We can't give it the constraint type in the case of generic constraints
            -
    - Type argument inference in the absence of candidates also uses the rule from Proposal 1 for treating T as the constraint in the absence of candidates
- Specialization + overload resolution rules allow required parameters to be optional (Bug #829546)
    - Perhaps this is just an acceptable hole.
- The compiler should check type parameter constraints during generic instantiation on function assignments (Bug #808522)
    - This appears to be fixed in more recent builds
- Contextually typing return types (Jason's email, attached)