Notes - 5/23/2014

Monday, May 19, 2014 8:31 AM

Agenda

- Design changes
  - Overload ordering of globally-scoped functions
  - Type argument inference candidate collection algorithm
  - Overload resolution algorithm changes
  - Legal alias merging
- Infinitely expanding types
- Generic inference
- Symbols
- Protected

Notes

- Overload ordering of global-scoped function signatures across files: When multiple overloads exist in different declaration spaces, later declarations should come first even if they're in the global scope. Clarifying, the 'declaration space' container boundary for ordering reasons in the global scope is a file. TypeScript 1.0 did not handle this for the global scope correctly.
- Type argument inference candidate collection
  - Current behavior: During inferential typing of something with signatures for the purpose of generic type inference, we relate the N signatures of the target to the M signatures of the source in an NxM manner. However, only signatures with at least as many parameters as their targets are considered possible sources during this operation.
  - New behavior: When making inferences between a two types, correlate the signatures in a 1:1 parallel fashion, starting with the last signatures first and proceeding backward until we run out of signatures. We suspect no arity checking is needed at this point. This seems to produce better results, and is faster.
  - Common behavior: Disjoint overload sets lead to poor inference. Unclear how to solve that.
  - Concern: Doesn't this possibly lead to inference of types that won't be assignable to the actual calls?
- Overload resolution algorithm
  - Current behavior: "Provisional binding" occurs during overload resolution. Example: when a function expression with a contextually-typable signature is passed to an overloaded function, the compiler needs to "try" to apply each contextual type and see if it causes a failure, which is potentially very large and quadratic.
  - Goal: Only apply a contextual type to a function expression *once* during type inference?
  - New behavior: During overload resolution, *first*, look at all arguments that aren't subject to contextual typing of function expressions. Apply overload resolution to these 'primary' arguments first, treating 'secondary' arguments as being universally compatible. At that point, apply the matching overload and push the contextual type into the secondary arguments and proceed.
  - Question: What is a contextually-typable argument? Answer: It's based on the lexical shape of the argument
  - Question: What about non-function contextually-typable arguments? Answer: Those are 'primary' because they are re-typable
  - This fixes Array#reduce :D
  - This is based on the observation that functions cannot reasonably change their behavior based on the expectations of the body of a function expression
- Legal alias merging
  - Old behavior: It was legal to 'import' two times as long as one was value-only and one was type-only
  - New behavior: Only one import into a lexical name is allowed
  - Applies to both internal and external module imports
- Infinitely expanding generic types
  - Old behavior: Illegal and/or confusing, depending
  - New behavior: Detect structural comparison at depths > 10 that are identical in the 'stack' of how we got there on both sides. When that stack is sufficiently deep and identical, assume it continues forever and presume that the relationship in question is true. At depth > 100, assume the compiler is going to crash and issue an error for diagnostics' sake.
  - This may have false positives, but they are wholly degenerate (and humans can't reason about it anyway)
- Generic inference to { } and its unfortunate consequences
  - Danger here is created by covariance of function parameters allowing { } to be a legitimate inference
  - See what happens if BCT = {} being an error during generic type inference
- Protected
  - What does it really mean?

Generic Inference

[from Cyrus' email]

I'm finding our generic inference algorithms provide very suboptimal results in very simple situations. During real world coding I've been bitten by this numerous times. When it happens my first instinct is that I absolutely must have run into a bug. And yet, that's not the case. The algorithm is working

correctly, and yet does something completely unintuitive and unhelpful. Here's a simplified example of how this occurs:

```
function forEach<T, U>(array: T[], func: (v: T) => U): U[] {
    var result: U[] = [];
    for (var i = 0; i < array.length; i++) {
        result.push(func(array[i]));
    }
    return result;
}

var strings = ["foo", "bar"];
var addOne = (v: number) => v + 1;

var result = forEach(strings, addOne);
```

This example typechecks and emits without 0 warnings or errors, despite being completely broken and not typesafe. According to the language, 'result' is a number[] from typescript's perspective, but is actually a string[] at runtime. If I try to use the values in that array as numbers, I will break completely given that they are actually strings.

Why does this happen? Well, at the point where we call 'forEach(strings, addOne)' type inference spins up. It ends up coming up with both 'string' and 'number' as candidates for 'T'. Because of how our type inference rules work, the only type we can end up inferring there is {}. And, both 'string' and 'number' fit that inferred type. By falling back to {}, we've ended up making type inference so lenient that it doesn't catch even the simplest typing errors when generics are involved. This is exactly the *opposite* of what I want. I am using generics to provide relations between types. And I want the type system to stop this sort of mistake dead in its track.

Note: This is not a contrived example. Vlad and I have into this several times when using Linq-like functions on arrays. We would commonly be doing things like:

```
forEach(arrayOfDocuments, processNode)
```

This would succeed during typecheck for the reason listed above. However, it would crash nearly immediately because 'processNode' was passed a 'Document' instead of a 'Node' as it was typed to accept.

I really think we need to do something about this. Inferring {} when type inference fails means, effectively, that type inference is always succeeding, and type errors are moved to crashes (if you're lucky) or subtle problems at runtime.

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;
}
```

Protected

Let's start collecting these as we find them

Calling base class methods from derived when using fat arrow methods

```
// https://typescript.codeplex.com/workitem/2491
```

```
class Base {
    // General handler
    // Desired visibility: protected
    public _doHandleEvent(e: MouseEvent) {
        // Do some stuff
    }

    // 'this'-safe handler
    public handleMouseEvent = (e: MouseEvent) => this._doHandleEvent(e);
}

class Derived extends Base {
    public handleMouseEvent = (e: MouseEvent) => {
        super._doHandleEvent(e);
        // Do some other stuff
    };
}
```

Change logic for getters and setters

```
class Base {
    private _x: string;

    // Intended public surface area
    public get x() {
        return this._x;
    }
    public set x(newX) {
        this.changeX(newX);
    }

    // Desired visbility: protected
    public _changeX(x: string) {
        // Perform change notification, etc.
        /*...*/
        this._x = x;
    }
}

class Derived extends Base {
    public _changeX(x: string) {
        // Perform additional recalculation
        /*...*/
        super._changeX(x);
    }
}
```