Notes - 7/18/2014

Thursday, July 17, 2014 11:56 AM

Agenda

- Empty object contextually typed by type with indexer
- Type argument inference from signature infers fewer candidates
- Imports and removing code during codegen
- Widening rules and return types
- Untyped calls
- Wrapping up current ES6 work
- Working with anonymous types

Empty object contextually typed by type with indexer

```
interface Foo { a }
interface Bar { b }

interface Object {
    [n: number]: Foo;
}

interface Function {
    [n: number]: Bar;
}

var o = {};
var f = () => { };

var v1: {
    //!!! Cannot convert '{}' to '{ [x: number]: Foo; }':
    //!!!   Numeric index signature is missing in type '{}'.
    [n: number]: Foo
} = o;
var v2: {
    //!!! Cannot convert '() => void' to '{ [x: number]: Bar; }':
    //!!!   Numeric index signature is missing in type '() => void'.
    [n: number]: Bar
} = f;
```

Expected: no errors
Actual: errors in the comments above

http://vstfdevdiv:8080/DevDiv2/DevDiv/_workitems/edit/955684

Anders: fine this is an error. Properties, but not indexers or call signatures, are added to the apparent type.

We may need to change no-implicit-any to not error on using the object indexer.

Type argument inference from signatures infers fewer candidates

```
interface Promise<T> {
     then<U>(success?: (value: T) => U, error?: (error: any) => U, progress?: (progress: any) => void): Promise<U>;
     done<U>(success?: (value: T) => any, error?: (error: any) => any, progress?: (progress: any) => void): void;
}
interface IPromise<T> {
     then<U>(success?: (value: T) => IPromise<U>, error?: (error: any) => IPromise<U>, progress?: (progress: any) => void): IPromise<U>;
     then<U>(success?: (value: T) => IPromise<U>, error?: (error: any) => U, progress?: (progress: any) => void): IPromise<U>;
     then<U>(success?: (value: T) => U, error?: (error: any) => IPromise<U>, progress?: (progress: any) => void): IPromise<U>;
     then<U>(success?: (value: T) => U, error?: (error: any) => U, progress?: (progress: any) => void): IPromise<U>;
     done? <U>(success?: (value: T) => any, error?: (error: any) => any, progress?: (progress: any) => void): void;
}

declare function testFunction11(x: number): IPromise<number>;
declare function testFunction11(x: string): IPromise<string>;
declare function testFunction11P(x: number): Promise<number>;
declare function testFunction11P(x: string): Promise<string>;

var s11: Promise<number>;
var s11a = s11.then(testFunction11, testFunction11, testFunction11); // ok
var s11b = s11.then(testFunction11P, testFunction11P, testFunction11P); // ok
var s11c = s11.then(testFunction11P, testFunction11, testFunction11); // ok

Used to all succeed, but now they all fail. This is because when we collect candidates for U from testFunction11 and testFunction11P, we
only use the last overload instead of all the overloads. This is an algorithmic change in the new compiler.

They used to all succeed, but they were Promise<{}>, which is not super meaningful anyway.

r11a has similar behavior in promisePermutations3.ts
```

http://vstfdevdiv:8080/DevDiv2/DevDiv/_workitems/edit/959083

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?

Would this manufacture two types?

- In essence, we already have two types. The original and the widened type.

Has someone tried it?

- Jason willing to try it out and report back

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.

ES6 Work

Destructuring

(from: Notes - 4/25/2014)

function f() {
    return ["hello", 2];
} // today returns {}[]

var {a,b} = f(); // not enough to destructure

Proposal

arrays can have element types

var x = ["hello", 2]; // would infer {}[string, number]

{}[string, number] ==> [string, number, {}, {}, …]

x[0] // type string

x = []; // also allowed

We don't check the element types.

Issue:

- how do you assign between x[0] and x[i]? We get two different types, even if i == 0
- Also wouldn't catch x = ["",""]
- What about x[0] = 2? Would break now. Perhaps we never check the element types, we only use it when destructuring.

What about having a tuple type?

Possible: never infer [string, number], infer {}[string, number]. If you specify this is a tuple type, we don't let you access beyond that. The [string, number] would be a two element tuple.

Question, how do you name a tuple type? We can't just put it into an interface.

Working with anonymous types

Original example:

```
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]; }
})
```

We wanted to be able to:

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

Proposal: Use functions and pattern matching

Type functions/operators should be able to:

- Pattern match into a given type and produce another type
  - Ability to match and return the instance type of a class
  - Ability to match and return the type parameters of a class
- Reflect over a given type, with the ability to enumerate its components

```
typefunction instance_of<T>({constructor(...x): T}) {
  return T;
}
```

Notes:

- Need to be able to "don't care" about parts of the type you don't care about. Here, I try to ignore any parameters constructor might have

```
typefunction type_parameters_of<T>({constructor<...T>}) {
  return [...T];
}
```

Notes:

- In order to work with type parameters, we may need rest/spread operations that can treat the type parameters as a group.
- This function returns something that isn't really a type, but instead is a list of type parameters

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

```
typefunction +(T, U) {
  type retVal = {};
  for t in T {
    retVal[t] = T[t]
  }
  for u in U {
    retVal[u] = U[u];
  }
  return retVal;
}
```

Notes:

- This is getting more involved, showing we actually want to be able to expando a type based on components of types coming in to the function