Notes - 1/6 - SENT

Friday, January 06, 2012 10:15 AM

[In 41/5541 today at 10:15am]

Agenda:

- Engineering Update
  - January milestone plans
  - Hiring
- Affirm decisions from December:
  - Align with subset of ES6 modules
  - Remove “requires” and multiple inheritance (for now)
  - Add support for ES5 getter/setter to classes and modules in --es5 mode
  - Remove use of extern classes from our own lib files
    - Dependent on decisions about extern classes - no change yet
  - Optional parameters
  - Partial interfaces
    - Accumulate base interfaces
- Go or no go:
  - Class expressions
    - No go until proven need
  - Lambdas without “function” prefix
    - Go: need to implement
  - ASI for Strada constructs
    - Go: need to implement
  - Classes in static bodies
    - No go
  - Overload on constants
    - Go: but when and more details needed - put on agenda
  - Overloading in class implementations
    - No go
  - Notation for recursive types

Notes:

- Decision: Change "extern" to "declare"
- Consider: Omit "var" and "function" after "declare"
- Consider: Nameless modules

Rule:
- No 'thing'
- Yes to prototyping target typing

```
foo.bar = function() {
  // gets typechecked with this: typeof<bar>
}
```

```
assert()
assert.something()
```

```
// Option 1
function Assert() {
  static {
    something
  }
}
```

- Notation for recursive types
    - No go - until proven need
    - Report errors (function f() { return f; })
  - Enums:
    - Tentative remove - test on the compiler codebase first
- Design work:
  - Extern typing overlays
  - Minimal interface file representation (do we support extern class in interface files?)
  - Generic methods
  - /r and #reference and other meta-directives
  - Inference for lambdas
  - ES6 alignment: consider targeted expression-level features (destructuring? const?)
- Backlog:
  - Readonly/writeonly in type system
- Anything else?

```
// Option 2
var assert = {
  (): function() {   },
  something: function() { },
}
```

```
// Option 3
function assert() {

}
module assert {
  function something() {

  }
}
```

```
// Option 4
module assert {
  function () {

  }
  function something() {

  }
}
```

Classes:

Thanks for doing this thought experiment. It is a gre
continue to drive to do that.

This particular one seems like a bad trade-off becaus
exchange for reducing specification surface area (ext
module). Extern modules are defined by restriction
“unique” interface).

I love all attempts to remove complexity at this point
the language and I feel we have a “hit” on our hands

However, seems like extern module is better than its
with a great suggestion for overlaying extern class or

Steve

From: Anders Hejlsberg
Sent: Wednesday, December 21, 2011 4:16 PM
To: Strada Design Team
Subject: Minimal interface file representation

I’ve been thinking about a possible “minimal interfac
interfaces and extern variables and functions. If noth
core concepts out of which higher level constructs (s
what our interface files would look like if we were to

Consider this Strada code which exercises all of our c

```
property gp: number;
```

at goal to simplify Strada by removing concepts and I think we should

se we are adding complexity to core concepts like interface in
tern module grammar) and removing a less-core concept (extern
, which is relatively easy to understand (compared to an idea like

t, because having written 15K lines of Strada, I really like the core of
with the core.

s alternatives, especially since on another thread Anders has come up
nto constructors.

ce file" subset that represents every Strada construct using just
ning else, such a representation is useful when reasoning about the
such as modules, classes and enums) are built. It is also illustrative of
get rid of extern modules, classes and enums.

concepts:

I wanted to summarize my thoughts on “this typing” for our meeting tomorrow.

I think there are a couple of common situations where we could usefully and reliably infer a typ
“this”. Specifically:

- Assignment of a function expression to a property of an object, as in obj.prop = function(
  Here I think it would be useful and very reasonable to infer the type of “this” in the funct
  to be the type of “obj”. After all, almost always the function will be invoked as obj.prop(..
  we combine this type of inference with the proposed typing of the “prototype” property
  very nice scheme for inferring the this type in methods of classes written in plain old Java
- Assignment of a function expression to a property of an object literal, as in var obj = { … f
  function(…) { … } … }. Here I think we should infer the type of “this” in the function body t
  type of the object literal. That type in turn is either inferred from the structure of the obj
  or using target typing (as per proposal below).

Some examples of all this in action:

```
interface Point
{
    x: number;
    y: number;
    add(dx: number, dy: number): Point;
}

declare var Point: {
    new(x: number, y: number): Point;
    prototype: Point;
```

```
class Point
{
  constructor(x: number, y: number) {

  }
  property origin = new Point(...);
}
```

```
class Point(x: number, y: number) {

}
```

Rule:
- No 'thing'
- Yes to prototyping target typing

```
foo.bar = function() {
  // gets typechecked with this: typeof<bar>
}
```

```
assert()
assert.something()
```

```
function gf(): void;

class C
{
    property cp: number;
    function cf(): void;
    static {
        property cs: string;
    }
}

module M
{
    property p: number;

    function f(x: number): number;

    class D extends C { }

    interface I { }

    enum E { X, Y, Z }

    module N
    {
        interface J { }
    }
}
```

Three extensions are required to represent a minima
(branded) interfaces, the ability for interfaces to exte
With those in place, the representation is:

```
extern var gp: number;

extern function gf(): void;

unique interface C
```

al interface file for the code above: The ability to declare “unique”
end primitive types, and the ability to give interfaces dotted names.

```
    origin: Point;
};

// Type provided by extern declaration
// Because Point is a constructor function, this is inferred
// to be Point and return type is inferred to be void
function Point(x, y) {
    this.x = x;
    this.y = y;
}

// Point.origin declared as type Point
Point.origin = new Point(0, 0);

// Point.prototype declared as type Point
// this inferred as Point because of obj.prop assignment
// dx, dy, and return type inferred using target typing
Point.prototype.add = function(dx, dy) {
    return new Point(this.x + dx, this.y + dy);
};

// Object literal type inferred using target typing
// this in function add inferred to be type of object literal (i.e
Point)
// dx, dy, and return type of add inferred using target typing
Point.prototype = {
    x: 0,
    y: 0,
    add: function(dx, dy) {
        return new Point(this.x + dx, this.y + dy);
    }
};

// Type of thing inferred from object literal
// this in functions inferred to be object literal type
var thing = {
    name: "",
    setName: function(s: string) {
        this.name = s;
    },
```

```
// Option 1
function Assert() {
 static {
  something
 }
}



// Option 2
var assert = {
 (): function() {  },
 something: function() { },
}



// Option 3
function assert() {

}
module assert {
 function something() {

 }
}

// Option 4
module assert {
 function () {

 }
 function something() {

 }
}
```

```
{
    cp: number;
    cf(): void;
}

extern var C: {
    new(): C;
    cs: string;
};

unique interface M.D extends C { }

interface M.I { }

unique interface M.E extends numbe

interface M.N.J { }

extern var M: {
    p: number;
    f(x: number): number;
    D: {
        new(): M.D;
    };
    E: {
        X: M.E;
        Y: M.E;
        Z: M.E;
    };
    N: { }
};
```

Note that the representation of a type declared insid

We currently don't support importing modules, but y
value namespace and the type namespace. For exam
properties of M as well as all types whose dotted na
containing context.

The above doesn't factor in visibility, but I don't thin

```
r { }
```

de a module is an interface with a corresponding dotted name.

you could imagine that an import clause would import from both the
mple, “import * from M” in the example above would import all the
me starts with M into the value and type namespaces of the

k there’s a lot of complexity there. Basically, non-exported members

```
        getName: function() {
            return this.name;
        }
    };
```

Anders

From: Anders Hejlsberg
Sent: Wednesday, January 04, 2012 12:29 PM
To: Luke Hoban; Steve Lucco; Chris Anderson (DEVDIV); Strada Design Team
Subject: RE: Adding types to plain Javascript

I forgot to say, note in particular the recursive nature of target typed object and array literals b

From: Anders Hejlsberg
Sent: Wednesday, January 04, 2012 12:22 PM
To: Luke Hoban; Steve Lucco; Chris Anderson (DEVDIV); Strada Design Team
Subject: RE: Adding types to plain Javascript

In preparation for our meeting later today, let me outline my latest thinking.

I'm basically proposing two new features:

1. Allow separate declaration and definition of entities through "extern" declarations. This a to ascribe types to unadorned (or only slightly modified) JavaScript.
2. Introduce "target typing" (inference from the left hand side) in assignments, function cal return statements.

Target typing would occur in the following situations:

- In variable and property declarations with an initializer and a known target type, the righ side is target typed to the given type. The target type is known if the declaration has a ty annotation or a corresponding extern declaration.
- In function declarations with no parameter and return type annotations but a known targ the parameter and return types are inferred from the target type. The target type is know there is an extern function or class declaration for the function.
- In assignment statements, the right hand side is target typed by the left hand side.
- In function calls, argument expressions are target typed by parameter types.
- In return statements, return values are target typed by the return type of the containing

are simply omitted from the interface file, and types base types, if any).

Thoughts?

Anders

Classes:

```
class Point
{
  constructor(x: number, y: number) {

  }
  property origin = new Point(...);
}
```

```
class Point(x: number, y: number) {

}
```

```
var assert =
(function() {
  var failureCount = 0;

  declare assert: { (): number; something(): number }

  function assert() { }
  assert.something = function() { failureCount++ }
  assert.totalFailures = function() { return failureCount; }
  return assert;
})();
```

of exported members must themselves be exported (as must their

- In return statements, return values are target typed by the return type of the containing

For expressions occurring in other contexts, no target typing takes place and our existing infere
apply.

Target typing would proceed as follows:

- In function expressions that have no type annotations for parameters and return type, th
parameter and return types are inferred from the target type.
- In object literals, the target type governs what properties are permitted and each initializ
processed as a target typed assignment.
- In array literals, each element expression is processed as a target typed assignment to th
element type.

Target typing would have no effect on any other expression construct. So, for example, enclosi
expression in parentheses makes target typing disappear.

With the rules above, the constructor function inference example at the beginning of this mail
can be viewed as a special case of target based inference, i.e. the extern class declaration decla
constructor function and the function definition "assigns" it.

We can go over examples in our meeting—I need to go eat lunch now. ☺

One cool thing about this scheme is that it provides complete type checking for JSON documen
a target type, the above rules automatically type check any JSON document. And if we impleme
statement completion based on these rules, JSON documents just fall out naturally.

Anders

From: Luke Hoban
Sent: Tuesday, January 03, 2012 9:46 PM
To: Steve Lucco; Chris Anderson (DEVDIV); Anders Hejlsberg; Strada Design Team
Subject: RE: Adding types to plain Javascript

+1. I love the idea of this. The external typing of a library also serves as the internal typing.

Feels like there will be a number of corner cases to address – including:

- What type annotations take precedence when an extern class provides types for the cons

function.

```
module {
  var total = 0;
  function assert() { ... }
  assert.something := function(...) {...}

  export assert;
}
```

ence rules

ne

zation is

e array

ng an

thread
ares the

ts. Given
ent

structor,

but the actual constructor function is declared with type annotations? I suppose the ex
wins, and the constructor function needs to be assignable to this type?

- The inference from assignment here sounds like it could get murky – would it be just for
expressions assigned directly into the Point.prototype.add? What if it's a reference to fu
valued variable? What if the assignment is to an alias to Point.prototype.add? What if th
developer uses Object.defineProperty(Point.prototype, "add", {value: function…})? Wh
object literals assigned to Point.prototype? Some of these are quite common. Luckily th
failing to catch these would just lead to type information not flowing into the method bo
which is less likely to cause immediate errors, and can easily be patched with type
annotations. But it could make it very subtle to know which parts of the code are type an
and which aren't.
- When there are overloads in the extern class – how is the implementation type checked?
- Privates – as Steve raises. I suppose the specific issue here is that if the extern class does
declare the intended-to-be-private fields, then the implementation will produce errors o
assignments. Which would force the developer to write out the entire public+private sur
in the types, which would then be visible to external consumers.

Luke

From: Steve Lucco
Sent: Tuesday, January 03, 2012 9:00 AM
To: Chris Anderson (DEVDIV); Anders Hejlsberg; Strada Design Team
Subject: RE: Adding types to plain Javascript

This is a great direction. It would provide an intermediate option in blending Strada and Javasc
code. I have converted some JQuery plug-ins into Strada classes and others I have kept as type
annotated constructors. Using an "overlay type" on the constructors would increase the amou
information in the system and also give the possibility of derivation.

To accomplish this, we need to solve the key problem with extern class: private fields. As I und
them the proposals are:

1. Make all fields public. This places more burden on the scenario above, but is relatively ea
fresh Strada code.
2. Have a private modifier and mangle. The mangled name must be able to distinguish two
with the same name and the same leaf class name.
3. Have a mangling convention that is taken to mean private. A problem with this is that a c
using an interface file and not seeing the mangled private may re-introduce the same ma
private and hide the original. Another problem is that the mangling appears at every use
identifier (vs. just at declaration for a private modifier).

tern class

function
nction-
ne
at about
ough,
dies –

nnotated

sn’t
n
rface area

cript
-
nt of type

erstand

asy for

fields

class
ngled
e of the

To me, #1 and #2 have more appeal than #3. So it comes down to whether we want a private feature. For setting public surface area, it makes sense to have it.

Thanks,
Steve

From: Chris Anderson (DEVDIV)
Sent: Friday, December 23, 2011 5:40 PM
To: Anders Hejlsberg; Strada Design Team
Subject: RE: Adding types to plain Javascript

This would be very nice. My favorite part is that this would give us a super FxCop style validatio folks that want to dip their toes in with strada. For example, we probably can't get Strada fully integrated into the Windows build process in the next quarter, but we could definitely add out annotation files… very slick!

From: Anders Hejlsberg
Sent: Friday, December 23, 2011 3:11 PM
To: Strada Design Team
Subject: Adding types to plain Javascript

In the last design meeting we started discussing ways to gradually ease into strong typing with Javascript. Specifically, we were trying to find a way whereby you could specify the type of a fu being declared such that the "static" members of the function would be known.

It occurred to me that "extern" declarations are very close to what we need. Currently we perm an extern declaration or a normal declaration for a particular identifier. What if we extended o to permit both in a single compilation? In a sense that is already what extern means, it's just th compiler never sees the extern and the normal declaration in the same compilation. We could that.

Consider this Javascript code:

```
function Point(x, y) {
    this.x = x;
    this.y = y;
};

Point.prototype.add = function(dx, dy) {
```

on for

of band

plain
nction

mit either
ur rules
at the
change

```
    return new Point(this.x + dx, this.y + dy);
}

Point.origin = new Point(0, 0);
```

We could add the missing type information by including an extern declaration in the same com

```
extern class Point(x: number, y: number)
{
      property x: number;
      property y: number;
      function add(dx: number, dy: number): Point;
      static {
            property origin: Point;
      }
}
```

This would cause the compiler to type check the plain Javascript code as follows:

```
// Known to be a constructor function
// Infer type Point for "this"
// Infer type number for x and y
// Infer void return
function Point(x, y) {
    this.x = x;
    this.y = y;
};

// Infer type Point for "prototype" member
// Infer types of dx and dy and return type from assignment to "ad
Point.prototype.add = function(dx, dy) {
    return new Point(this.x + dx, this.y + dy);
}

// Point known to have static "origin" member
Point.origin = new Point(0, 0);
```

I think we could pretty much type check everything we discussed in our last meeting this way—
few more things, such as the type of “this” in the constructor and the type of Point’s “prototyp
member. In the assignment of the “add” function we would have to infer from the left hand sid

pilation:

dd"

-plus a
e”
de. I’m

not a great fan of this, but on the other hand it is the same mechanism that we would need for
inference in method calls (a'la LINQ), so it would be consistent to also support it in assignments

If we think that relaxing extern in this manner might lead to unreported duplicate identifier err
in case an extern and regular declaration in the same compilation accidentally pick the same na
could instead consider a "declare" modifier that declares the identifier (and its type) and *requi*
normal declaration to also appear in the same compilation—but otherwise works just like exte
kind of like the extern behavior because it would also work for cases where we never even see
"normal" declaration because it takes place through an abstraction such as Win.JS.define.

Anyway, I think solving the problem this way would be very consistent with what we already do

Anders

lambda
s.

ors, e.g.
ame, we
res a
rn. But I
the

o.