Notes - 12/9

Friday, December 09, 2011 9:53 AM

52

[In 41/1753 today at 10:15am]

JSConf: May 1-3

Agenda:

- Update on engineering and hiring
  - Progress toward 12/16 drop
  - jQuery
  - Test
- Make calls on current spec/implementation divergences:
  - Private
  - Statically-extensible interfaces and extern classes
    - Don't need an 'extension' modifier - we don't do this for modules or elsewhere
    - Don't do extern class extension
  - Others?
- Question: Should we consider including some targeted ES6 features? (destructuring perhaps?)
- Design work:
  - Mixins and inheritance
  - Generics
  - ‘import’
  - Module ‘code-loading’
  - Inference for lambdas
  - Abstract classes
- Anything else?

Inheritance:

- Issue with 'this' in
- Suggestion: Allow extending interfaces
  - Extending a class means normal prototype inheritance
  - You can extend multiple classes, the first is the spine
  - Extending an interface means assume it will be there at runtime and type 'super' wrt it
  - Extending any interfaces makes you non-instantiable

```
interface Ifoo {
  foo(): string;
}

class Base {
  foo()=> "Base";
}

class A extends Base { // regular extensions

}

class B extends IFoo { //
  foo() => super.foo() + "A";
}

class Fancy extends D2, A {

}


x with A(...) // requires x be an Ifoo
```

- Or does it need to be "extend x with A(...)";
- Idempotent - do we need a marker

Decision: Go for the extends/implements simplification with extending interfaces. No abstracts.

Implementation: For now, remove 'requires'.

Generics:

- View is generic method is bundle of overloads
- Overload resolution is not about picking which method to call

```
class Vehicle() {}

class Bus extends Vehicle() {}
class Bicyle extends Vehicle() {}


var x: {(x: Bus): any;}[]= [function(x: Vehicle) => null, function(x: Bus) => null]
```

We discussed adding ES5 accessor property support to Strada classes on Tuesday. Here’s what I expect the spec for those looks like. Note that the code generation is something we didn’t discuss, and is potentially concerning as it requires falling back to a library call instead of using the ES5-specific syntax given our use of imperative object construction instead of object literals. I think this is okay, but does involve a less trivial mapping of source to emitted code.

Re-rooting modules

Two common Node.js modules are Net and HTTP. Both of these expose a Server constructor c
This is hard to write in Strada today:

```
    // This is wrong because there are not global objects named
    // Net and HTTP, and these module names cannot be used to
```

Syntax:

ModuleSourceElement:

…

GetAccessorPropertyDeclaration

SetAccessorPropertyDeclaration

ClassSourceElement:

…

GetAccessorPropertyDeclaration

SetAccessorPropertyDeclaration

GetAccessorPropertyDeclaration:

`property get` *Identifier* `( )` *ReturnTypeAnnotation*$_{opt}$

FunctionImplementation`;`

SetAccessorPropertyDeclaration:

`property set` *Identifier* `(` *NamedParameter* `)` *ReturnTypeAnnotation*$_{opt}$

FunctionImplementation`;`

Type Checking:

Classes should allow properties which use ES5 accessor syntax. Type-wise, they represent a single property. If both present, the parameter type of the setter must be assignable to the return type of the getter. If a getter is present, the type of the property is the return type of the getter. If a getter is not present, the type of the property is the parameter type of the setter. The getter must have zero parameters, and the setter must have one parameter.

Code Generation:

The combination of get and set accessors for a name is combined into a single statement in the output, as a call to:

```
Object.defineProperty(/*class-prototype*/,
            Identifier,
            { get: function Identifier() {
/*GetAccessorPropertyDeclaration-FunctionImplementation*/ },
             set: function Identifier(NamedParameter) {
/*SetAccessorPropertyDeclaration-FunctionImplementation*/ },
             enumerable: true,
             configurable: true });
```

Luke

Yesterday we decided that

a. It probably makes sense to think of generic methods as an infinite bundle of overloads, and
b. We should try addressing calls to generic methods *without* the presence of other overload declarations first.

So here are some examples looking at calls to generic methods from the viewpoint of "overload bundles". I am making one more limiting assumption and not thinking about lambdas passed to functions yet – instead when methods take function parameters I am restricting to passing functions that already have a declared type. That way we can worry about dealing with the contravariance effects of function types first.

I do want to non-humbly tease that there is *awesome insight* towards the end, but you should read it all in sequence ☺. After, I deeply appreciate any thoughts you have.

```
// type instances returned from require('http')
module Net {
  class Server() {}
}
module HTTP {
  class Server() {}
}

// To deal with the above, we just provide interfaces for the module
// shapes of Net and HTTP.  However, then we can't nest the interfaces
// within them for scoping.  So we elevate to the top level, and get
// name conflicts
interface INet {
  Server: { new(): IServer }
}
interface IHTTP {
  Server: { new(): IServer }
}
interface IServer {  // Is this the net.Server or http.Server?
}
```

We really do want to use modules here, but those modules are not going to be guaranteed to top level. Instead, they can be requested and rooted based on what the user wants. I believe support this using ES6 syntax.

```
//consumer.str
module net from 'net'
module http from 'http'

var server: http.Server = http.createServer();
```

Imports can still be used in conjunction with this, but will potentially reintroduce name conflic to the developer to avoid these and keep the modules separate if there are name conflicts. (In these name conflicts can be dealt with statically in the same way as C#, lazily rejecting static re to individual conflicting names – though ES6 is stricter than this currently).

```
//consumer.str
module net from 'net'
module http from 'http'
import * from http

var server: Server = createServer();
```

This is still not quite right though. An attempt to access the value net.Server will statically app succeed, but will not be there at runtime. This is because 'module' conflates the static type hi with the static value hierarchy. What we really want here is the ability to import only the "typ the module. Notable, with classes and interfaces, we have the pair of both how to describe th only" portion and how to define the "type-plus-instance" portion. For modules, we do not hav "type-only" concept. Interfaces can partly play this role also – but cannot themselves contain We're missing something like a 'moduleface':

```
moduleface Net {
    class Server() {
    }
}
moduleface HTTP {
    class Server(): Net.Server() {
    }
}

Net.Server // error!  There is no value Net, similar to interfaces

var server : Net.Server = require('http').createServer(); // okay!
```

This has several benefits: (1) it allows types to be provided completely independently of struct

Recommendation: I don't think any of these options is yet good enough to pursue, but the no moduleface does seem like a hole we will want to try and address in some form. I like the dire

Mads

One covariant T in parameters:

```
m<T>(t:T): T;
var r = m(7);
```

Here it seems clear that `r` should get the type `number`. Out of the infinite bundle it seems that the result type `number` is "best". There could be a couple of reasons *why* it is best, though. First of all we have to decide whether to allow "reverse assignability" (assigned less specific to more specific types) for arguments to methods. If we don't, then only overloads with the type `number` and its supertypes even apply in the first place. If we do, then all the overloads with enum types, plus the `null` and `undefined` types (which are subtypes of `number`) *also* apply.

We probably want to say that overloads requiring reverse assignability of arguments either *don't* apply at all, or only apply if there aren't any others. If that is the case, then the set of applicable overloads here is those where T is `number` or a subtype.

There are still a couple of potential reasons we could employ why `number` should be the return type. Either because the `number` overload is *best* by some rule (e.g. subtyping on parameter types), or because `number` is the winner, or summary, or intersection of all the *return* types of all the applicable overloads.

One contravariant T in parameters:

```
m<T>(f:(t:T)=>string): T;
g: (x:number)=>string;
var r = m(g);
```

This example is a little weird. I take a function of something and return that something? Surely this is a bit contrived? What kind of function would do that? We can make it more realistic by returning a function of T instead of a T:

```
m<T>(f:(t:T)=>string): (t:T)=>bool;
g: (x:number)=>string = …;
var r = m(g);
```

Now you can imagine that m e.g. composes the incoming function with some post-processing of the string that results in a bool. The question is, what is the type of `r`? Clearly we'd like it to be `(x:number)=>bool`, but let's see what we can get from rules like what we applied above. First of all, the overloads of m that are applicable are those where the parameter type for `f` is `(x:number)=>string` or a supertype thereof. Wait, what is a supertype of a function type? It is one where the return type is more general (but they are all `string` here so that doesn't matter) *or the parameter types are more specific!!!* For instance, `m<{}>` is *not* applicable, but `m<E>` for some enum type `E` *is*.

So the set of applicable overloads is mighty different than before, in that the T's are an infinite set of *subtypes* rather than a finite set of *supertypes*. But it doesn't matter much. The most specific parameter type (by subtyping) is still the one where T is `number`. Remember, those other candidates where applicable precisely because their parameter types were supertypes of `(x:number)=>string`. So that one is a subtype of all the others.

Similarly if we look at return types: If we pick the most specific of the return types it will be the right one. If we combine them all with intersection or whatever, it will be the right one. So, somewhat surprisingly perhaps, nothing new here compared to the covariant case above.

Two covariant T's in parameters:

```
m<T>(a: T, b: T): T;
var r = m(bicycle, bus);
```

Imagine a `choose` method which returns one of its two arguments based on the phases of the Moon. The applicable overloads are those where `T` is a supertype of `Bicycle` and also of `Bus`. This amounts to finding the most specific common supertype; let's call it `Vehicle` (though it may not be defined with a name anywhere in the program). Clearly the overload with `Vehicle` is the "best" by all the definitions

adding additional basic tools for making typing more flexible. This can be adapted to other use more easily than building a holistic module-based code-loading system into Strada which tries abstract over all possible module systems in current use.

References and command line

Let's say I want to compiler stradac-node.str, and it depends on types from strada.str and ..\lib\node.str. However, I do not want to re-emit the .js files for the latter two files – they ma be generated, or may be in folders I don't even have write permissions to. Either way, for com stradac-node.str, there is no need to code-gen the others.

This command line will try to compile them all to individual output files:

```
stradac stradac-node.str strada.str ..\lib\node.str
```

This command line will concatenate them all into one, which is also not desired:

```
stradac stradac-node.str strada.str ..\lib\node.str –out stradac-node.js
```

What we want in principle is:

```
stradac stradac-node.str /r:strada.str /r:..\lib\node.str
```

That is, the auxiliary strada files are references, not sources. They are used for populating the value environments for type checking, but not involved in code generation at all.

Now, in principle we want most of these sorts of command line options to be available to be s within source code, so that I do not need a project file to drive tooling.

```
//stradac-node.str
#reference 'strada.str'
#reference '..\lib\node.str'

var compiler = new Tools.StradaCompiler();
// …
```

This replaces the current comment driven model. The use of '#reference' keeps the notion of "reference", not a dynamic import. The # also helps separate this as a metadirective instead o of executable code.

Now, ideally – I could just write the following, and the compiler resolves the references implici source, just like other tools will need to:

```
stradac stradac-node.str
```

This relies on treating the "#reference" like one of the "/r" command line options.

Recommendation: Adopt /r and #reference as static-only reference mechanisms, separate this dynamic code loading for now.

above, so the only thing new here is that we have to accept synthesizing best common supertypes (which isn't particularly hard to do in a structural system).

Two contravariant T's in parameters:

```
m<T>(f:(t:T)=>string, g:(t:T)=>number): (t:T)=>bool;
h: (x:Bicycle)=>string;
i: (x:Bus)=>number;
var r = m(h,i);
```

OK, let's say that this method returns a function that returns true if the length of the `string` returned by `f` applied to its argument `t` is greater than the `number` returned by `g` applied to `t`. (if that just turned your brain to mush, don't worry about it – just saying that the signature is plausible). The applicable overloads are those where `T` is a subtype of `Bicycle` and also of `Bus`. This amounts to finding the most general common subtype; say `Buscycle`. While it is rather unlikely that any buscycles actually exist, it is no harder for the compiler to produce that type than it is to compute a most specific common supertype as we did above.

While applicable overloads exist using *subtypes* of `Buscycle` (Buscyclerries and Buscyclanes for instance), the one with `Buscycle` is clearly the *least* useless, and indeed the one rules like above would choose. Note that it is no accident that the result here is rather useless. We are trying to combine functions that only work for bicycles with functions that only work for busses. There shouldn't be a whole lot of common ground there. So it is no failure of the system to produce this useless result; it is a failure of the scenario.

More interesting would be if we had used `Vehicle` instead of `Bus` in the example. Here the outcome would be (using all suggested approaches) the overload based on `Bicycle`. I.e. when we combine a more with a less general function we get the type of the less general.

Co- *and* contravariant T's in parameters:

```
m<T>(a:T f:(t:T)=>bool): T;
g: (t:Vehicle)=>bool;
var r = m(bus, g);
```

Combining what we have seen above, the applicable overloads are those where `T` is a *supertype* of `Bus` and a *subtype* of `Vehicle`. Now something interesting happens, though. Whenever an overload gets better for the first argument it gets worse for the second. Going by parameter types, *there is no best overload!!* Look at the endpoints, Bus and Vehicle for instance. Clearly `Bus` is better than `Vehicle` for the first parameter. However, almost as clearly `Vehicle=>bool` is better than `Bus=>bool` for the second parameter! So based on parameters, `m<Vehicle>` and `m<Bus>` are equally good, and so are all the overloads in between.

It is tempting to apply an arbitrary rule here – e.g. "pick the most specific of the T's", which in this case would cause us to infer the type of r to be `Bus`. That seems right enough in this case. But this might as well be wrong. Consider this slight variation:

```
n<T>(a:T f:(t:T)=>bool): (t:T)=>bool;
g: (t:Vehicle)=>bool;
var r = n(bus, g);
```

Everything but the return type (and the name of the method) is the same. But now that arbitrary rule would give us the *least* useful type possible as the return type.

It is time to think deeper about what these overloads really mean. Recall that the overloads represent *one* implementation at runtime. That implementation has *no* access to any type argument we happen to infer, and it has *no* access to the compile time types of any of its arguments. It only has access to the arguments *themselves*, including what runtime type information it can glean.

We should therefore feel confident that the return type provided by *any* applicable overload represents a correct (if incomplete) type for what the implementing method would produce on *any* arguments satisfying that overload's static types. Think about that for a second. What that means is that *all of the applicable overloads are right*. In other words, the actual value returned by the call will have *all the return types that the applicable overloads provide*!

To see this more clearly, take the arguments bus and g above. Without any unsafe stuff going on I can do the following:

```
var h: (t:Bus)=>bool = g; // totally safe and legal
var r2 = m(bus, h);
```

We've called the same runtime method with the same objects, but now clearly the result is a Bus! There's just one applicable overload!

Conversely we can do:

```
var vehicle: Vehicle = bus; // boringly legal
var r2 = n(vehicle, g);
```

And just as clearly the result is a (v:Vehicle)=>bool. Again, there's just one applicable overload.

In both cases, by *weakening* our static knowledge about an argument, we got the most specific result type possible. That must mean that result type really actually applies! (Unless of course the overloads are not a correct representation of the behavior of the runtime method – but let's deal with that kind of issue another day).

So because all overloads represent the same runtime method, *everything* we know about the returned value from *every* applicable overload is true. And of course, the combination of all that knowledge is – you guessed it – the intersection of all those return types. The most general common subtype of all the return types.

Co- and contravariant T's in return types:

This is an awesome simple rule to follow and think about: The result type of a method application is the most general common subtype of all the result types of all the applicable overloads.

For generic methods we have to take into account the little snag that there are sometimes infinitely many applicable overloads. So we need some algorithmic handle on this infinity.

For all the results above it so happened that the combined result type was at one of the endpoints of a spectrum - it was the most specific of the return types available. However, there may not be such a most specific return type. Let's combine the last two versions of m and n:

```
m<T>(a:T f:(t:T)=>bool): { a:T; f:(t:T)=>bool; }
g: (t:Vehicle)=>bool;
var r = m(bus, g);
```

The endpoints of the spectrum are `{ a:Vehicle; f:(t:Vehicle)=>bool; }` and `{ a:Bus; f:(t:Bus)=>bool; }`. However, the combined information tells us more than each of these; namely that the result is always a `{ a:Bus; f:(t:Vehicle)=>bool; }`. It is in fact amazing that this is the result – it fits so snugly with the arguments we passed in! But we do need a way to find it that terminates within the lifetime of the Universe. I believe there is a straightforward algorithm for this:

The parameters have essentially given rise to an upper and a lower bound for T (Vehicle and Bus in this case). As we construct the result type we keep track of where T occurs contravariantly and where it occurs covariantly in the declared return type. In the contravariant positions we substitute its upper bound. In the covariant positions we substitute its lower. Done!

Thoughts?

Mads

From: Mads Torgersen
Sent: Tuesday, December 06, 2011 9:47 AM
To: Strada Design Team
Subject: Some thoughts on generic methods

I've been doing some thinking on generic methods, and the below is as much of a summary as I had time for before today's meeting. Looking forward to discuss, and apologies for sending this so close to the meeting – it had to get written first! ☺·

Mads

Generic types seem relatively straightforward. In a structural world such as ours, they are really just a templating mechanism for types.

The real fun comes in with generic methods. Those cannot in the same way just be thought of as templates – rather they are intricately tied to the "ground level" type system itself. When a type has a generic method, that describes a real-world entity – an actual method present at runtime in an object – that cannot just be "expanded away." The underlying runtime reality therefore needs to play a role in how we shape generic methods.

A model of generic methods
How should we think of generic methods? A good place to start is with subtyping. Assume we have the following types:

```
interface A : { f(a: any): any }
interface B : { f(s: string): string }
interface C : { f<T>(t: T): T }
```

What are the subtype relationships between them? There is no subtype relationship between A and B because co- and contravariance of function types work in opposite directions. How about the relationship between C and the other types though? How should we think of the type of C.f and compare it to the others?

In C# we don't have such problems. The type system is nominal, and C.f is just a different f than A.f or B.f – they have no relationship. When assigned to a delegate a generic method in C# must first be instantiated with type arguments; it doesn't have a type in and of itself.

The only reasonable model I can think of in Strada is to say that a generic method conceptually is a bundle of overloads, one for each possible instantiation. The type of C.f is { <T>(t: T): T } which is a short form of saying { (t: any): any; (t: {}): {}, … , (t: number): number, (t: string): string, … }; i.e. an infinite number of similarly structured overloads.

This model lets us compare C to the other types. By our usual rules of subtyping, C is a subtype of both A and B, because for each overload in each of those, there is a conforming overload in C. Of course dealing with infinite lists of overloads is interesting territory from an algorithmic perspective, but we'll figure that out: after all these are highly regular sets of overloads induced by a single declaration.

Conceptually I think the model makes a lot of sense. If a set of overloads describes the ways in which a function can be called (and the result types that correspond to each), then this is really an accurate description of what generic methods are for.

This model also means that "inferring the type argument" for a generic method call is an act of overload resolution: which of the infinitely many overloads are *applicable*, and which do we pick? So let's talk some about that.

Overload resolution
Overload resolution is again one of those concepts where we cannot just transfer out experience from C# directly. At runtime there *are* no overloads in Strada – they are purely there to describe the typing as accurately as possible at compile time. So "overload resolution" really has two purposes:

- Deciding whether a given function call is well typed
- Determining the return type of the call expression

Because overload resolution does not have runtime semantic import, there are two kinds of errors it really seems unfortunate to have to give as a result:

- Ambiguity: there is no ambiguity at runtime. If two overloads apply equally well, that should be even better than one, and not cause the compiler to give up.
- Getting it wrong: Lack of static information shouldn't be able to lead us down the wrong path, making type assumptions that are patently bad.

Addressing the first one first, what *should* we do if there is more than one applicable overload and there isn't a good way to choose? We could use a *bad* way to choose (i.e. arbitrary), but that would lead to false type assumptions down the road. *Or* we could somehow "choose all of them". And by that I mean find a way to "merge" the result types of all the applicable overloads (or at least all the "best" applicable overloads for some meaning of that word) and use that as the type of the call expression.

There are a number of different approaches one could take to this "merging" business:

- Union types – which (confusingly) means types constructed by taking the intersection of member sets
- Intersection types – which (just as confusingly) means types constructed by taking the union of member sets
- Something in between – e.g. coming up with a notion of optional members meaning that *some* objects of the type will have those members

Union types are safe and correct, in that they produce a supertype of all the possible return types, therefore adequately describing what they have in common and what can therefore be known about *all* of them. However, they are not very useful because that intersection of members is often quite trivial. The first thing you want to do is assign to the type you *actually* (as a programmer) know it to be.

Intersection types are a lie, but maybe a good lie. They sort of claim that the result has *all* the possible return types at the same time, which is of course preposterous. However, if as a programmer you know what it is *really* supposed to be, it is super useful to be able to directly dot into the right members, and for the result to be a subtype of what you know it to be. This might offer the opportunity to get rid of our reverse assignability rule, I may get to that later.

I've spent a lot of time in the wilderness of optional members. All I got was a bunch of complexity without much payoff. I think we'd venture there at our own peril, and waste a lot of time getting little value. I think both union type and intersection type approaches are worth investigating.

Applicability
When is a function overload applicable? With our C# hat on we would base that on "implicit convertibility" – i.e. assignability – of arguments to parameter types. There are a number of things to consider though:

- Should we consider expected result types also? After all Java does that, and it could really help limit the number of applicable candidates nicely. As long as we keep it within the statement, it shouldn't get *too* complicated.
- What kind of assignability do we consider? Does the argument type have to be a *subtype* of the parameter type, or do we admit overloads that only apply by reverse assignability?

I won't get into too many pro's and cons now. One thing I will dive into: with the right combination of settings we might get rid of reverse assignability from the language! How so? The motivating scenario

for reverse assignability is for factories that produce a number of different kinds of things to be able to have their return value (typed by a common supertype) directly assigned to one of those product types. However, if we describe such a factory method as a generic method:

make<T>(recipe: string): T

And we allow overload resolution (and hence type inference) to take expected type into account, then we can call this as:

Result: MyKindOfProduct = make("MyKindOfProduct");

MyKindOfProduct would become an upper bound on T in the call, and hence only the overloads where T is a subtype of MyKindOfProduct would apply. Similarly you could consider

make("MyKindOfProduct").Foo(7);

to be legal, imposing { Foo(a: int): any } as an upper bound on T. In general, the expectations on the return type of "make" can probably be summarized as a structural type.

There are guaranteed to be subtleties here, but it is an interesting idea to pursue.

Constraints

One simplification we've considered is to not have constraints on type parameters. If we go with a "merge" model for the return types of applicable overloads, this might not be the best option. When we call a generic method, the argument types (and also perhaps expected result types) will introduce both upper and lower bounds on type parameters. To the extent that the result type is itself generic, it would be better (at least more precise) to capture those upper and lower bounds on the type parameters of that result type.

Lambda arguments
In C# we infer through lambda arguments by "pushing in" known parameter types and seeing what we get out the other end.

There is an alternative possible approach in Strada: We could find the type of the function expression compositionally, without considering its context. We could do that by letting each unspecified parameter type of the lambda give rise to a fresh type parameter on the (generic) function type that we are inferring, and then collecting constraints on that type parameter based on how the corresponding parameter is used in the lambda.

Mads