Notes - 12/19

Monday, December 19, 2011 9:32 AM

[In 41/4749 today at 10:00am]

Agenda:

- Design work:
  - Optional parameters
  - Mixins and traits
  - Support for dynamic loading
  - /r and #reference and other meta-directives

  - Generics
  - Inference for lambdas
  - ES6 alignment: consider targeted expression-level features (destructuring? const?)
- Anything else?

Optional parameters:

```
   function foo(x: number, y: number, z: string = undefined, w: bool = true) {
   }

interface Foo {
   foo(x:number, y:number = ?, z: string = ?): number;
   bar(x:number, y?:number, z?:string): number;
}
```

? Is optional in declaration when "= true"

Overloading

May be back for heterogeneous return types

```
function Vec
  this.x = x;
  this.y = y;
  this.z = z;
}
```

Thoughts:

- No classes across compilations:
  - Large software would need to all compile together

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

```
Function () {
  var x = {a: 1, 
  x.c = "hello";
  x.d = 0;
}
```

Auto-lift constants

Decision: We
Could we say

But if we infer
side? Do we h

Mixins:

class Foo extends C1, …, CN, I1, …, In requrires J1, … Jn implements K1, … Kn {

}

Decision: Remove mixins, keep classses/interfaces ctor: Vector_Static = (x,y,z) {

b: 2}

think there is something we can do, but
- any function declaration

the static side, what about the instance
nave to track the

Modules:

```
// .str
export interface I { … }
export function foo(s: string): stirng;

// .js
This.foo = function…
This.x = "…"';

// .i.str
Interface I {…}
Function foo(s: string): string;
```

Lib.str
Mona

```
//-----------
module M from "foo.js", "bar.js"
```

One of the things we didn't get to yesterday was to explore how to do dynamic module loa
the other progress that we made. Below is a start at doing that, assuming that we keep th
derivation without representation" paradigm that we discussed yesterday. The below also
export vs. private etc.

Decision: Remove mixins, keep classses/interfaces

Yesterday we decided that

a) It probably makes sense to think of generic methods as
b) We should try addressing calls to generic methods *with*
declarations first.

So here are some examples looking at calls to generic metho
bundles". I am making one more limiting assumption and no
functions yet – instead when methods take function parame
that already have a declared type. That way we can worry ab
of function types first.

I do want to non-humbly tease that there is *awesome insigh*
in sequence ☺. After, I deeply appreciate any thoughts you h

Mads

co

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 o
type `number` is "best". There could be a couple of reasons *wh*
decide whether to allow "reverse assignability" (assigned les
arguments to methods. If we don't, then only overloads with
apply in the first place. If we do, then all the overloads with e
types (which are subtypes of `number`) *also* apply.

ding, given
e "no
adopts ES6

We probably want to say that overloads requiring reverse as
at all, or only apply if there aren't any others. If that is the ca
here is those where T is `number` or a subtype.

s an infinite bundle of overloads, and
hout the presence of other overload

ds from the viewpoint of "overload
t thinking about lambdas passed to
ters I am restricting to passing functions
bout dealing with the contravariance effects

t towards the end, but you should read it all
have.

f the infinite bundle it seems that the result
hy it is best, though. First of all we have to
s specific to more specific types) for
n the type `number` and its supertypes even
enum types, plus the `null` and `undefined`

signability of arguments either *don't* apply
se, then the set of applicable overloads

First, we have been discussing over the past few weeks that modules combine multiple ide view modules as establishing a namespace in which to place types and simultaneously the a variable inside a scope. To get dynamically loadable modules, we need to keep a version but ditch the latter. To do so we need some syntax, which for this draft will be the keywor indicating that the module is not placed in its surrounding scope.

Next, we need some way to describe the type of the dynamic module. We can do so with t steps:

1. We let interfaces nest. This has the beneficial side-effect of clearing one of our long- issues: representation of recursive types. For example: `function f()=>f` has the `{ interface f { ():f; }; ():f }`.
2. We describe the *compile-time* effect of a hidden module named `M` on its surrounding (which is the global scope because only non-nested modules can be hidden), as creat interface named `M`, but not a variable named `M`. This is different from a regular modu establishes a namespace. One of the consequences of this is that if a regular module class C, a client of that module can derive from C, because C is a type accessible thro namespace established by the regular module. However, the hidden module's type i interface, identical to the interface that would be generated for an interface file from module.
3. We describe the *run-time* effect of a hidden module as adding the exported variables module to '`this`' whatever its current value (it does not have to be the global object
4. Clients of a hidden module use a loading function to add the hidden module's proper object. For example,
   a. `var localM={} as M;`
   b. `loaderCode.call(localM,"M.str");`
5. Clients of multiple hidden modules can combine those modules by creating an interfa `interface M extends M1, M2;` The client could then execute the loader code a accumulating both `M1` and `M2` into `localM`.

A bonus of nested interfaces is that we may be able to use them to eliminate extern modul `extern` as something that modifies only class properties or module variables. To do this, v express what is now an extern module as a nested interface `M` followed by the declaration variable of the same name as in `extern var M:M;`

Thinking through this set of changes made me think again about a consequence of the ES6 which we discussed yesterday: we have module variables which may be exported but we h properties which may be private (Mads brought this up at the end of the meeting). Seems are the correct defaults (module default private; class default public), but the vocabulary is

as. We can
declaration of
of the former
d **hidden**,

the following

standing
type

g scope
ting an
ule, which
e declares a
ugh the
s a nested
n a regular

s of the
t).
rties to an

ace as in
bove,

les, leaving
we would
of an extern

adoption
ave class
s like these

There are still a couple of potential reasons we could employ
Either because the `number` overload is *best* by some rule (e.g
`number` is the winner, or summary, or intersection of all the

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 a
bit contrived? What kind of function would do that? We can
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 fur
string that results in a bool. The question is, what is the type
`(x:number)=>bool`, but let's see what we can get from rules
overloads of m that are applicable are those where the parar
a supertype thereof. Wait, what is a supertype of a function t
more general (but they are all `string` here so that doesn't m
specific!!! For instance, `m<{}>` is *not* applicable, but `m<E>` for

So the set of applicable overloads is mighty different than be
subtypes rather than a finite set of *supertypes*. But it doesn't
type (by subtyping) is still the one where T is `number`. Remem
applicable precisely because their parameter types were sup
one is a subtype of all the others.

Similarly if we look at return types: If we pick the most specif
one. If we combine them all with intersection or whatever, it
surprisingly perhaps, nothing new here compared to the cov

Two covariant T's in parameters:

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

why number should be the return type.
. subtyping on parameter types), or because
return types of all the applicable overloads.

and return that something? Surely this is a
make it more realistic by returning a

nction with some post-processing of the
of `r`? Clearly we'd like it to be
s like what we applied above. First of all, the
meter type for `f` is `(x:number)=>string` or
type? It is one where the return type is
natter) *or the parameter types are more*
some enum type `E` *is*.

efore, in that the T's are an infinite set of
matter much. The most specific parameter
nber, those other candidates where
ertypes of `(x:number)=>string`. So that

fic of the return types it will be the right
will be the right one. So, somewhat
ariant case above.

different. We need a word that means the opposite of export, but not import, more like pr
export. The only things that come to mind seem somewhat political: `embargo`, `blockad`

Steve

revent
e, etc.

```
var r = m(bicycle, bus);
```

Imagine a `choose` method which returns one of its two argu
The applicable overloads are those where `T` is a supertype of
finding the most specific common supertype; let's call it `Veh`
name anywhere in the program). Clearly the overload with `V`
above, so the only thing new here is that we have to accept s
(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)=>bo
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
by `f` applied to its argument `t` is greater than the `number` ret
your brain to mush, don't worry about it – just saying that th
overloads are those where `T` is a subtype of `Bicycle` and als
general common subtype; say `Buscycle`. While it is rather u
no harder for the compiler to produce that type than it is to
as we did above.

While applicable overloads exist using *subtypes* of `Buscycle`
instance), the one with `Buscycle` is clearly the *least* useless,
choose. Note that it is no accident that the result here is rath
functions that only work for bicycles with functions that only
whole lot of common ground there. So it is no failure of the s
failure of the scenario.

More interesting would be if we had used `Vehicle` instead o
would be (using all suggested approaches) the overload base
more with a less general function we get the type of the less

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);
```

ments based on the phases of the Moon.
f `Bicycle` and also of `Bus`. This amounts to
`icle` (though it may not be defined with a
`ehicle` is the "best" by all the definitions
synthesizing best common supertypes

```
ol;
```

true if the length of the `string` returned
urned by `g` applied to `t`. (if that just turned
e signature is plausible). The applicable
o of `Bus`. This amounts to finding the most
nlikely that any buscycles actually exist, it is
compute a most specific common supertype

(Buscyclerries and Buscyclanes for
and indeed the one rules like above would
her useless. We are trying to combine
work for busses. There shouldn't be a
system to produce this useless result; it is a

f `Bus` in the example. Here the outcome
ed on `Bicycle`. I.e. when we combine a
general.

Combining what we have seen above, the applicable overloa
and a *subtype* of Vehicle. Now something interesting happe
better for the first argument it gets worse for the second. Go
overload!! Look at the endpoints, Bus and Vehicle for instanc
the first parameter. However, almost as clearly Vehicle=>bo
parameter! So based on parameters, m<Vehicle> and m<Bus
overloads in between.

It is tempting to apply an arbitrary rule here – e.g. "pick the r
would cause us to infer the type of r to be Bus. That seems ri
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
would give us the *least* useful type possible as the return typ

It is time to think deeper about what these overloads really r
one implementation at runtime. That implementation has *no*
infer, and it has *no* access to the compile time types of any o
arguments *themselves*, including what runtime type informa

We should therefore feel confident that the return type prov
a correct (if incomplete) type for what the implementing me
satisfying that overload's static types. Think about that for a
applicable overloads are right. In other words, the actual valu
return types that the applicable overloads provide!

To see this more clearly, take the arguments bus and g above
do the following:

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

We've called the same runtime method with the same objec
There's just one applicable overload!

ds are those where `T` is a *supertype* of `Bus` ens, though. Whenever an overload gets oing by parameter types, *there is no best* ce. Clearly `Bus` is better than `Vehicle` for ool is better than `Bus=>bool` for the second > are equally good, and so are all the

most specific of the T's", which in this case ight enough in this case. But this might as

) is the same. But now that arbitrary rule e.

mean. Recall that the overloads represent o access to any type argument we happen to f its arguments. It only has access to the tion it can glean.

vided by *any* applicable overload represents thod would produce on *any* arguments second. What that means is that *all of the* ue returned by the call will have *all the*

e. Without any unsafe stuff going on I can

al

ts, but now clearly the result is a `Bus`!

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

In both cases, by *weakening* our static knowledge about an a
type possible. That must mean that result type really actually
are not a correct representation of the behavior of the runti
issue another day).

So because all overloads represent the same runtime metho
value from *every* applicable overload is true. And of course, t
you guessed it – the intersection of all those return types. Th
return types.

Co- and contravariant T's in return types:

This is an awesome simple rule to follow and think about: Th
most general common subtype of all the result types of all th

For generic methods we have to take into account the little s
many applicable overloads. So we need some algorithmic ha

For all the results above it so happened that the combined re
spectrum - it was the most specific of the return types availa
specific return type. Let's combine the last two versions of m

```
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:Veh`
`f:(t:Bus)=>bool; }`. However, the combined information
that the result is always a `{ a:Bus; f:(t:Vehicle)=>bool`
result – it fits so snugly with the arguments we passed in! Bu
terminates within the lifetime of the Universe. I believe ther

, there's just one applicable overload.

rgument, we got the most specific result
y applies! (Unless of course the overloads
me method – but let's deal with that kind of

d, *everything* we know about the returned
the combination of all that knowledge is –
he most general common subtype of all the

e result type of a method application is the
he applicable overloads.

snag that there are sometimes infinitely
ndle on this infinity.

esult type was at one of the endpoints of a
ble. However, there may not be such a most
n and n:

`hicle)=>bool; }` and `{ a:Bus;`
tells us more than each of these; namely
`; }`. It is in fact amazing that this is the
t we do need a way to find it that
e is a straightforward algorithm for this:

The parameters have essentially given rise to an upper and a
case). As we construct the result type we keep track of where
occurs covariantly in the declared return type. In the contrav
bound. In the covariant positions we substitute its lower. Do

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 b
for before today's meeting. Looking forward to discuss, and a
meeting – it had to get written first! ☺

Mads

Generic types seem relatively straightforward. In a structural
templating mechanism for types.

The real fun comes in with generic methods. Those cannot in
templates – rather they are intricately tied to the "ground le
generic method, that describes a real-world entity – an actua
that cannot just be "expanded away." The underlying runtim
how we shape generic methods.

A model of generic methods
How should we think of generic methods? A good place to st
following types:

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

lower bound for T (Vehicle and Bus in this
e T occurs contravariantly and where it
variant positions we substitute its upper
ne!

below is as much of a summary as I had time
apologies for sending this so close to the

l world such as ours, they are really just a

n the same way just be thought of as
vel" type system itself. When a type has a
al method present at runtime in an object –
e reality therefore needs to play a role in

tart is with subtyping. Assume we have the

What are the subtype relationships between them? There is
because co- and contravariance of function types work in op
relationship between C and the other types though? How sh
compare it to the others?

In C# we don't have such problems. The type system is nomi
B.f – they have no relationship. When assigned to a delegate
instantiated with type arguments; it doesn't have a type in a

The only reasonable model I can think of in Strada is to say th
bundle of overloads, one for each possible instantiation. The
form of saying { (t: any): any; (t: {}): {}, … , (t: number): numb
number of similarly structured overloads.

This model lets us compare C to the other types. By our usua
and B, because for each overload in each of those, there is a
with infinite lists of overloads is interesting territory from an
that out: after all these are highly regular sets of overloads in

Conceptually I think the model makes a lot of sense. If a set o
function can be called (and the result types that correspond
description of what generic methods are for.

This model also means that "inferring the type argument" fo
resolution: which of the infinitely many overloads are *applica*
some about that.

Overload resolution

Overload resolution is again one of those concepts where we
C# directly. At runtime there *are* no overloads in Strada – the
accurately as possible at compile time. So "overload resoluti

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

Because overload resolution does not have runtime semanti
really seems unfortunate to have to give as a result:

no subtype relationship between A and B
posite directions. How about the
ould we think of the type of C.f and

nal, and C.f is just a different f than A.f or
a generic method in C# must first be
nd of itself.

hat a generic method conceptually is a
type of C.f is { <T>(t: T): T } which is a short
er, (t: string): string, … }; i.e. an infinite

l rules of subtyping, C is a subtype of both A
conforming overload in C. Of course dealing
algorithmic perspective, but we'll figure
nduced by a single declaration.

of overloads describes the ways in which a
to each), then this is really an accurate

r a generic method call is an act of overload
able, and which do we pick? So let's talk

e cannot just transfer out experience from
ey are purely there to describe the typing as
on" really has two purposes:

c import, there are two kinds of errors it

- Ambiguity: there is no ambiguity at runtime. If two ove
  even better than one, and not cause the compiler to gi
- Getting it wrong: Lack of static information shouldn't b
  making type assumptions that are patently bad.

Addressing the first one first, what *should* we do if there is m
isn't a good way to choose? We could use a *bad* way to choo
false type assumptions down the road. *Or* we could someho
find a way to "merge" the result types of all the applicable o
overloads for some meaning of that word) and use that as th

There are a number of different approaches one could take t

- Union types – which (confusingly) means types constru
  sets
- Intersection types – which (just as confusingly) means
  member sets
- Something in between – e.g. coming up with a notion o
  objects of the type will have those members

Union types are safe and correct, in that they produce a supe
therefore adequately describing what they have in common
of them. However, they are not very useful because that inte
The first thing you want to do is assign to the type you *actua*

Intersection types are a lie, but maybe a good lie. They sort o
return types at the same time, which is of course prepostero
what it is *really* supposed to be, it is super useful to be able t
for the result to be a subtype of what you know it to be. This
our reverse assignability rule, I may get to that later.

I've spent a lot of time in the wilderness of optional member
without much payoff. I think we'd venture there at our own
value. I think both union type and intersection type approach

Applicability

When is a function overload applicable? With our C# hat on

erloads apply equally well, that should be
ive up.
be able to lead us down the wrong path,

more than one applicable overload and there
ose (i.e. arbitrary), but that would lead to
w "choose all of them". And by that I mean
verloads (or at least all the "best" applicable
he type of the call expression.

to this "merging" business:

ucted by taking the intersection of member

types constructed by taking the union of

of optional members meaning that *some*

ertype of all the possible return types,
and what can therefore be known about *all*
ersection of members is often quite trivial.
lly (as a programmer) know it to be.

of claim that the result has *all* the possible
us. However, if as a programmer you know
o directly dot into the right members, and
might offer the opportunity to get rid of

rs. All I got was a bunch of complexity
peril, and waste a lot of time getting little
nes are worth investigating.

we would base that on "implicit

convertibility" – i.e. assignability – of arguments to paramete
consider though:

- Should we consider expected result types also? After a
  limit the number of applicable candidates nicely. As lo
  shouldn't get *too* complicated.
- What kind of assignability do we consider? Does the a
  parameter type, or do we admit overloads that only a

I won't get into too many pro's and cons now. One thing I wi
settings we might get rid of reverse assignability from the la
for reverse assignability is for factories that produce a numb
have their return value (typed by a common supertype) dire
However, if we describe such a factory method as a generic

make<T>(recipe: string): T

And we allow overload resolution (and hence type inference
we can call this as:

Result: MyKindOfProduct = make("MyKindOfProduct");

MyKindOfProduct would become an upper bound on T in the
is a subtype of MyKindOfProduct would apply. Similarly you

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

to be legal, imposing { Foo(a: int): any } as an upper bound o
return type of "make" can probably be summarized as a stru

There are guaranteed to be subtleties here, but it is an inter

Constraints

One simplification we've considered is to not have constrain
"merge" model for the return types of applicable overloads,
call a generic method, the argument types (and also perhaps
upper and lower bounds on type parameters. To the extent t
be better (at least more precise) to capture those upper and
that result type

er types. There are a number of things to

ll Java does that, and it could really help ng as we keep it within the statement, it

rgument type have to be a *subtype* of the pply by reverse assignability?

ll dive into: with the right combination of nguage! How so? The motivating scenario er of different kinds of things to be able to ctly assigned to one of those product types. method:

) to take expected type into account, then

e call, and hence only the overloads where T could consider

n T. In general, the expectations on the ctural type.

esting idea to pursue.

ts on type parameters. If we go with a this might not be the best option. When we expected result types) will introduce both that the result type is itself generic, it would lower bounds on the type parameters of that result type.

Lambda arguments
In C# we infer through lambda arguments by “pushing in” kn
get out the other end.

There is an alternative possible approach in Strada: We coul
compositionally, without considering its context. We could d
parameter type of the lambda give rise to a fresh type param
are inferring, and then collecting constraints on that type pa
parameter is used in the lambda.

Mads

own parameter types and seeing what we

d find the type of the function expression
o that by letting each unspecified
neter on the (generic) function type that we
rameter based on how the corresponding