Notes - 9/11/2013

Wednesday, September 11, 2013 1:00 PM

Agenda

- Generics and assignment compatibility
- Type of variables in recursive initialization
- Treatment of modules and classes as read-only
- Do we expand enums?

Generics and assignment compatibility

Given the example, how do we simply check for assignment compatibility?

```
var f: <T>(x:T) => T;
var g: (x:string) => string;

//Now, which of the two is supported?
f = g;
g = f;
```

The way to think about this is to think of this as "take parameter list of f, and write a lambda that calls g, and type-check this new lambda"

```
//f = g
//typechecks the following expression to check for compatibility:
<T>(x:T):T => g(x);

//g = f
//typechecks the following expression to check for compatibility:
(x:string):string => f(x);
```

This is now in the spec as "contextual signature instantiation"

Discussion

"Using the process described in 3.8.6, inferences for *A*'s type parameters are made from each parameter type in *B* to the corresponding parameter type in *A* for those parameter positions that are present in both signatures."

Action item (spec bug #780691): Needs to be reworded to focus on fixing the matching types for each type parameters. The example starts with the fixing of T and string (or setting up the candidate set with them as a pair).

Next example:

```
var f: <T>(x:Array<T>): T
var g: (x: Array<string>): string;
```

Now we apply a the pattern matching from 3.8.6 to look into the structure of types and try to create the candidate set. This is not necessarily a directed match.

Aside question: Does the compiler do the subtyping in a way that lifts properly from T <: U to Array<T> <: Array<U>?

If there are overloads, we try all of the overloads when checking for this compatibility. If at least one overload works out, then assignment compatibility succeeds. This is in line with existing assignment compatibility rules.

For every overload of the left-hand side, the right-hand side must have at least one in the overload set that is assignment compatible.

Action item (spec bug #780690): clarify 3.8.6 "If T is one of the type parameters for which inferences are being made and S or any type occurring in a member of S is not the wildcard type, S is added to the set of inferences for that type parameter." Actual intended meaning: Where (S is not a wildcard) && (No member of S contains wildcard).

Another example from 4.18:

```
map<T, U>(a: T[], f: (x:T)=>U): U[];

function identity<Z>(x: Z): Z { return x; }

var a: string[];
var b = map(a, (x)=>x.length);
var c = map(a, identity);
```

Currently, for c, we don't make the type inference for x's type Z. We need to unify the T = string and Z. We do a step of contextual typing to help fix these type variables so that we can solve the ultimate type.

What about:

```
interface I(<T>) {

    f<U extends T>(x: U) { }
```

```
}
```

Is this an occurrence or not? What if x:U is missing?

Action item *(spec bug #780693): need to have a more formal definition for "occurrence".*

Resolution

Accept Anders' proposed change to type-checking the assignments of generics.

Type of variables in recursive initialization

What does:

```
var x = x() < 5;
```

…mean?

Discussion

Anders: In spirit, if the right-hand side directly or indirectly references 'x', then x would have time 'any' and this type is now bound to 'x'. This is any use of 'x', in the process of inference, that is defined on the lhs of the assignment

What about:

```
var x = x = 3;
```

Answer: treated as: var x = (x = 3), so x would be 'any'

What about:

```
x = f();

function f() { … x …. }
```

The issue here is that this appears to weaken the system, because the "letrec" style was typed correctly before would now be typed as 'any'. This is a step backward for some examples that are correctly more precise.

What about:

```
var x = (y: number)=> …x…)
```

Issue is that x is currently the more accurate (y:number)=>any, but under the discussed rule would become 'any'.

Action Item *(spec bug #780699): May need to clarify 6.3: "Otherwise, if f's function body directly references f or references any implicitly typed functions that through this same analysis reference f, the inferred return type is Any."*

What if the function's return is what references through the "same analysis" f? We may want to explore documenting how we handle this in the pull model in the spec.

Resolution

The dev team currently implements something that is more sophisticated. We actually have implement something similar to a "2-phase" type check because of the pull type-checker work. This should likely be a bug on the x() based on this in the original var x = x() < 5 example.

Action item (Ask on Dev Team, Task #780704)*: make a list of where we short circuit or are more "fancy" than the spec directly states. We'll use this to drive spec revisions that describe this more robust approach.*

Treatment of modules and classes as read-only

Currently, the spec and implementation differ on whether classes, modules, functions and enums should be read-only. In 0.9.1, the compiler allowed functions to be writeable but not modules. In 'develop', this is reversed. What is the expected behavior?

Discussion

For readonly-ness of classes, modules, functions, we currently treat functions as readonly. All assignments fail in this example:

```
module m {
    export function f() {

    }

    f = function() { }
    m.f = function() { }
}

function g() {
    function h() {
```

```
        }

        h = function() {}
    }

    g = function() {}
```

Our current implementation does allow writing to modules, but not assigning to classes and functions (as of 0.9.1.1).

Resolution

Top level declarations can be checked for read-only, but members are writeable. This would be more restrictive than the 0.9.1.1, as modules would no longer be directly assignable.

For enums, this would allow people to assign to enum members.

Action item *(spec bug #780717): Need to explore the impact of making this change*

Do we expand enums?

Discussion

Cyrus: yes, by default enums are expanded/inlined if they are constants.

Resolution

If enums can be modified, we may want to investigate a flag that does not expand/inline enums so that the user can mutate and observe the new value.