Hello everyone, I'll be presenting three GHC Language Extensions today, namely: ScopedTypeVariables, RankNTypes, and FlexibleInstances.

\*Flip\*

The first extension I'll be presenting is ScopedTypeVariables. Now, this code here, does not need ScopedTypeVariables, but if we wish to add an explicit type signature, for the sub-function `go`, we need to add a `forall` keyword, and the `ScopedTypeVariables` pragma.

This is because..

\*Flip\*

`forall` enables the parametric `a`s, in the type signature of the sub-function `go`, to be ad-hoc polymorphic `Ord a`s, as you can see in this code.

\*Wait\*

\*Flip\*

The next extension is RankNTypes. Of the three extensions, this is the most difficult to understand. Firstly, we need to know the difference between Rank-1 and Rank-2 polymorphism. The difference is in the placement of brackets. The code here shows Rank-1 polymorphism..

\*Flip\*

and here, the code shows Rank-2 polymorphism. See the brackets? The outermost brackets is required. The inner brackets can be omitted for this code.

\*Wait\*

\*Flip Back\*

We also need to know that in this function, the function rank1 is the caller, and the function f is the callee.

In Rank-1 polymorphism, the caller is applied to and chooses the

type of the callee. Here, the type of the callee becomes the function, n right arrow n, as defined in the type signature. The type variable n cannot unify with the output type Double.

\*Flip\*

Here, the function rank2 is the caller, and f is the callee.

In Rank-2 polymorphism, the caller does not choose the callee type. The callee `f` chooses its own type. This is why the type variable n is able to unify with Double.

\*Flip\*

This is an example of the callee `f` freely choosing the Int and Double types in the tuple.

\*Wait\*

\*Flip\*

Finally, we have FlexibleInstances. As stated in the error for this code here, instance declarations should firstly not have lists inside because they should be distinct type variables. Secondly, these distinct type variables should appear only once in the instance head.

So, can we use String instead of List Char?

\*Flip\*

No, because all instance types should not be synonyms. In this case, the suggested TypeSynonymInstances pragma doesn’t work, only the FlexibleInstances pragma works.

\*Flip\*

In this code here, we have the pragma, and instance Something List a. This works.

\*Wait\*

\*Flip\*

In this code here, we have the pragma, and instance Something String. This also works.

\*Wait\*

\*Flip\*

But this doesn’t work. There is an instance Something2 String and an instance Something2 List a, at the same time. This is considered overlapping instances.

\*Wait\*

If we need both instances, ..

\*Flip\*

we can create another function doSomethingWithList in class Something2, with a default output, and also applying ad-hoc polymorphism to the instance of Something2 List a. Here, we do not need the FlexibleInstances pragma.

NOW, this code here is done with the help of Andreas.

At the bottom of the code are 2 tests.

`test` returns `3` because "hello" has type List Char, so the function `doSomethingWithList` under `instance Something2 Char` is called.

`test2` returns `4` because List 1,2::Int has type List Int, so the default function `doSomethingWithList` under `class Something2 a` is called.

\*Flip\*

That's all. Any questions?