GHC Language Extensions

```
{-# LANGUAGE ScopedTypeVariables, RankNTypes, FlexibleInstances #-}
```

ScopedTypeVariables

- This code does not need ScopedTypeVariables:

```
mini :: Ord a => [a] -> Maybe a
mini [] = Nothing
mini xs = Just $ foldl1 go xs
  where go = min
```

- However, 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.

ScopedTypeVariables

- `forall` enables the parametric `a`s in the type signature of the sub-function `go` to be ad-hoc polymorphic `Ord a`s.

```
{-# LANGUAGE ScopedTypeVariables #-}
mini :: forall a. Ord a => [a] -> Maybe a
mini [] = Nothing
mini xs = Just $ foldl1 go xs
  where
    go :: a -> a -> a
    go = min
```

Reference:

1. https://limperg.de/ghc-extensions/#scopedtypevariables

RankNTypes

```
printing = print $ rank1 (+1)
rank1 :: Num n => (n -> n) -> Double
rank1 f = f 1.0
```

- *Error: Couldn't match expected type ‘Double’ with actual type ‘n’*

  ‘n’ is a rigid type variable bound by the type signature

- The `rank1` function is Rank-1 polymorphic. It is also known as the caller of `f`. In Rank-1 polymorphism, the caller is applied to and chooses the type of `f` (the callee), which becomes rigidly `(n -> n)` as defined, so the variable `n` cannot become `Double`. To overcome this problem, we must declare `rank1` as a Rank-2 polymorphic function.

RankNTypes

- So, we include the `RankNTypes` pragma and include a `forall` keyword whereby `forall`, `Num n` and `n->n` are enclosed in parentheses: `(forall n. Num n => n -> n)`, so that the function rank2 is Rank-2 polymorphic. The callee `f` is then applied with the type signature in parentheses together: `(forall n. Num n => n -> n)`, which allows it to choose and match the desired output type `Double`. `n` is unified with `Double`. This is because in Rank-2 polymorphism, the callee chooses its own type.

```
{-# LANGUAGE RankNTypes #-}
printing = print $ rank2 (+1)
rank2 :: (forall n. Num n =>  (n -> n)) -> Double
rank2 f = f 1.0
```

RankNTypes

- This works too. Here, the callee `f` is applied to and chooses `Int` and `Double` separately in the tuple.

```
{-# LANGUAGE RankNTypes #-}
 printing = print $ rank2 (+1)
 rank2 :: (forall n. Num n =>  (n -> n))  ->  (Int, Double)
 rank2 f = (f 1, f 1.0)
```

References:

1. https://stackoverflow.com/questions/33446759/understanding-haskells-rankntypes
2. https://www.schoolofhaskell.com/school/to-infinity-and-beyond/pick-of-the-week/guide-to-ghc-extensions/explicit-forall#rankntypes--rank2types--and-polymorphiccomponents
3. https://stackoverflow.com/questions/67823600/haskell-why-are-we-using-rankntypes-for-single-type-outputs
4. https://stackoverflow.com/questions/42820603/why-can-a-num-act-like-a-fractional

FlexibleInstances

```
class Something a where
  doSomething :: a -> Integer
instance Something Integer where
  doSomething x = 1
instance Something Char where
  doSomething x = 2
instance Something [Char] where
  doSomething x = 3
```

- Error: Illegal instance declaration for ‘Something [Char]’
  (All instance types must be of the form (T a1 ... an)
  where a1 ... an are *distinct type variables*,
  and each type variable appears at most once in the instance head.
  Use FlexibleInstances if you want to disable this.)

FlexibleInstances

- If we change `instance Something [Char] where` to `instance Something String where`, a slightly different error is thrown:
- Error: Illegal instance declaration for ‘Something String’

  (All instance types must be of the form (T t1 ... tn)

  where T is not a synonym.

  Use TypeSynonymInstances if you want to disable this.)
- However, including the TypeSynonymInstances pragma doesn’t work, only the FlexibleInstances pragma works.

FlexibleInstances

- This works:

```
{-# LANGUAGE FlexibleInstances #-}
class Something a where
  doSomething :: a -> Integer
instance Something Integer where
  doSomething x = 1
instance Something Char where
  doSomething x = 2
instance Something [a] where
  doSomething x = 3
```

FlexibleInstances

- This works too:

```
{-# LANGUAGE FlexibleInstances #-}
class Something a where
  doSomething :: a -> Integer
instance Something Integer where
  doSomething x = 1
instance Something Char where
  doSomething x = 2
instance Something String where
  doSomething x = 3
```

FlexibleInstances

- But this doesn't work due to overlapping instances of `String` and `[a]`:

```
{-# LANGUAGE FlexibleInstances #-}
class Something2 a where
  doSomething2 :: a -> Integer
instance Something2 Integer where
  doSomething2 x = 1
instance Something2 Char where
  doSomething2 x = 2
instance Something2 String where
  doSomething2 x = 3
instance Something2 [a] where
  doSomething2 x = 4
test = doSomething2 "hello"
```

- Error: Overlapping instances for Something2 [Char]
  arising from a use of `doSomething2'

FlexibleInstances

- If both `[a]` and `String` typed instances are required, the pragma FlexibleInstances is not required, but the code is more complicated:

```
class Something2 a where
  doSomething2 :: a -> Int
  doSomethingWithList :: [a] -> Int
  doSomethingWithList x = 4
instance Something2 Int where
  doSomething2 x = 1
instance Something2 Char where
  doSomething2 x = 2
  doSomethingWithList x = 3
instance Something2 a => Something2 [a] where
  doSomething2 = doSomethingWithList
test = doSomething2 "hello"
test2 = doSomething2 [1,2::Int]
```

FlexibleInstances

- `test` returns `3` because `"hello"` has type `[Char]`, so the function `doSomethingWithList` under `instance Something2 Char` is called.
- `test2` returns `4` because `[1,2::Int]` has type `[Int]`, so the default function `doSomethingWithList` under `class Something2 a` is called.

Reference:

1. https://connectionrequired.com/blog/2009/07/my-first-introduction-to-haskell-extensions-flexibleinstances
2. https://wiki.haskell.org/List_instance