What's new in GHC 7.8

Francesco Mazzoli, Erudify <f@mazzo.li>

Type holes

A type hole can be placed wherever a Haskell expression should go.

It tells you:

- Which type the compiler is expecting where the hole is;
- The types of bound variables.

Type holes

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = _

    Found hole ‘_’ with type: a -> c
    Where: ‘c’ is a rigid type variable bound by ...
           ‘a’ is a rigid type variable bound by ...
    Relevant bindings include
      g :: a -> b (bound at ...)
      f :: b -> c (bound at ...)
```

`newtype` coercions

The problem:

```
newtype Age = Age Int

toAgeMap :: HashMap Int String -> HashMap Age String
toAgeMap = ?
```

`newtype` coercions

```
coerce :: Coercible a b => a -> b

newtype Age = Age Int

toAgeMap :: HashMap Int String -> HashMap MyInt String
toAgeMap = coerce
```

`newtype` coercions

`Coercible` is not a normal type class, its rules being hard-coded in GHC:

- `∀ a. Coercible a a`
- `∀ a b. Coercible a b => Coercible b a`
- `∀ a b c. (Coercible a b, Coercible b c) => Coercible a c`
- `∀ a. given newtype C` t_1 … t_n `= C a => Coercible a (C` t_1 … t_n`)`
- `∀` a_1 … a_n b_1 … b_n`.`
  `(Coercible` a_1 b_1`, … , Coercible` a_n b_n`) => Coercible (C` a_1 … a_n`) (C` b_1 … b_n`)`

`newtype` coercions

```
newtype Ptr a = Ptr Addr#

Coercible (Ptr CInt) (Ptr CString)?
```

Type families

```
type family Key (f :: * -> *) :: *
class Functor f => Lookup f where
  lookup :: f a -> Key f -> Maybe a
```

```
type instance Key [] = Int
instance Lookup [] where
  lookup xs ix | ix < 0 = Nothing
  lookup [] _           = Nothing
  lookup (x : xs) ix    =
    if ix == 0 then Just x
    else lookup xs (ix - 1)
```

```
type instance Key (Map k) = k
instance Lookup (Map k) where
  lookup = Map.lookup
```

Closed type families

Type families are open, and thus overlapping definitions are forbidden:

```
{-# LANGUAGE DataKinds, TypeFamilies #-}
data Nat = Zero | Succ Nat

type family CountArgs (f :: *) :: Nat
type instance CountArgs (a -> b) = Succ (CountArgs b)
type instance CountArgs a        = Zero
```

Closed type families

Lets us write closed “type functions”:

```
type family CountArgs (f :: *) :: Nat where
    CountArgs (a -> b) = Succ (CountArgs b)
    CountArgs a        = Zero
```

Matching from top to bottom, like value-level functions.

Lets us write overlapping patterns, something we cannot do with open families.

Overloaded lists

```
class IsList l where
  type family Item l :: *
  fromList  :: [Item l] -> l
  toList    :: l -> [Item l]
  fromListN :: Int -> [Item l] -> l

instance IsList (Vector a) where
  type instance Item (Vector a) = a
  fromList  = V.fromList
  toList    = V.fromList
  fromListN = V.fromListN
```

Minimal type class instances

```
class Bifunctor p where
  bimap :: (a -> b) -> (c -> d) -> p a c -> p b d
  bimap f g = first f . second g

  first :: (a -> b) -> p a c -> p b c
  first f = bimap f id

  second :: (b -> c) -> p a b -> p a c
  second = bimap id
```

Minimal type class instances

This instance emits no warnings, but every method invocation will result in an infinite loop:

```
class Bifunctor (,) where
```

Minimal type class instances

```
class Bifunctor p where
  {-# MINIMAL bimap | (first, second) #-}
  bimap :: (a -> b) -> (c -> d) -> p a c -> p b d
  bimap f g = first f . second g

  first :: (a -> b) -> p a c -> p b c
  first f = bimap f id

  second :: (b -> c) -> p a b -> p a c
  second = bimap id
```

Typed TemplateHaskell

- `TExp a`, like `Exp` but indexed by the type of the contained expression.
- To introduce `TExp`s we use typed quasis (`[|| … ||]`) or typed splices (`$$x`).

```
compose :: Q (TExp (b -> c)) -> Q (TExp (a -> b))
        -> Q (TExp (a -> c))
compose f g = [|| $$f . $$g ||]
```

We can safely compile and run `TExp`s at runtime.

Safe `GeneralizedNewtypeDeriving`

- `GeneralizedNewtypeDeriving + TypeFamilies = unsafeCoerce`
- In other words, GHC Haskell < 7.7 is unsound.
- The problem: GND treats the `newtype C = C Int` as “the same” as `Int`, but they might have different instances for a type family.
- Fixed in 7.8 using “roles”.

But wait, there's more!

- Parallel builds with `ghc -j`
- Improved IO manager
- Type natural solver
- `-XNumDecimals`
- Dynamic by default
- `clang` support
- iOS support and cross compilation

A more complete picture at https://ghc.haskell.org/trac/ghc/wiki/Status/Oct13

Future plans

- `class Applicative m => Monad m`
- Pattern synonyms
- Explicit type application