Livin' la `via` loca
Coercing Types with Class

Baldur Blöndal **Ryan Scott**[1] Andres Löh[2]

[1]Indiana University
[2]Well-Typed LLP

rgscott@indiana.edu
github.com/RyanGlScott

Haskell

Glasgow **H**askell **C**ompiler
(or, **GHC**)

Glasgow **H**askell **C**ompiler
(or, **GHC**)

`deriving`

A `deriving` primer

```haskell
data Exp = Lit Int | Plus Exp Exp


class Eq a where
  (==) :: a -> a -> Bool
```

A deriving primer

```haskell
data Exp = Lit Int | Plus Exp Exp


instance Eq Exp where
  (Lit i1) == (Lit i2)
    = (i1 == i2)
  (Plus e1 f1) == (Plus e2 f2)
    = (e1 == e2) && (f1 == f2)
  _ == _ = False
```

A `deriving` primer

```
data Exp = Lit Int | Plus Exp Exp
               | Times Exp Exp

instance Eq Exp where
  (Lit i1) == (Lit i2)
    = (i1 == i2)
  (Plus e1 f1) == (Plus e2 f2)
    = (e1 == e2) && (f1 == f2)
  (Times e1 f1) == (Times e2 f2)
    = (e1 == e2) && (f1 == f2)
  _ == _ = False
```

A `deriving` primer

```
data Exp = Lit Int | Plus Exp Exp
         | Times Exp Exp
  deriving Eq
  -- Autogenerates the
  --
  --   instance Eq Exp
  --
  -- behind the scenes
```

A `deriving` primer

```
data Exp = Lit Int | Plus Exp Exp
  deriving Eq
```

A `deriving` primer

```
data Exp = Lit Int | Plus Exp Exp
               | Times Exp Exp
   deriving Eq
```

```haskell
newtype Age = MkAge Int
```

```haskell
newtype Age = MkAge Int
```

These have the same representation at runtime.

```haskell
newtype Age = MkAge Int
```

These have the same representation at runtime.

```haskell
succInt :: Int -> Int
succInt i = i + 1

succAge :: Age -> Age
succAge (MkAge i) = MkAge (i + 1)
```

```haskell
newtype Age = MkAge Int


instance Show Int where ...

instance Show Age where
  show (MkAge i) = “MkAge ” ++ show i
```

```haskell
newtype Age = MkAge Int


instance Num Int where ...

instance Num Age where
  (MkAge a1) + (MkAge a2)
    = MkAge (a1 + a2)
  (MkAge a2) - (MkAge a2)
    = MkAge (a1 - a2)
  ...
```

```haskell
newtype Age = MkAge Int


instance Integral Int where ...

instance Integral Age where
  div (MkAge a1) (MkAge a2)
    = MkAge (div a1 a2)
  mod (MkAge a2) (MkAge a2)
    = MkAge (mod a1 a2)
  ...
```

GHC's solution: generalize `deriving`!

GHC's solution: generalize `deriving`!

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
```

```haskell
newtype Age = MkAge Int


instance Num Int where ...

instance Num Age where
  (MkAge a1) + (MkAge a2)
    = MkAge (a1 + a2)
  (MkAge a2) - (MkAge a2)
    = MkAge (a1 - a2)
  ...
```

```haskell
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

newtype Age = MkAge Int
  deriving Num

instance Num Int where ...
```

```haskell
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

newtype Age = MkAge Int
  deriving Num

instance Integral Int where ...

instance Integral Age where
  div (MkAge a1) (MkAge a2)
    = MkAge (div a1 a2)
  mod (MkAge a2) (MkAge a2)
    = MkAge (mod a1 a2)
  ...
```

```haskell
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

newtype Age = MkAge Int
  deriving (Num, Integral)

instance Integral Int where ...
```

Things currently in GHC

New GHC features
(our work)

Things currently in GHC

New GHC features
(our work)

Class instance patterns

Class instance patterns

```haskell
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a

class Applicative f where
  pure   :: a -> f a
  liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

Class instance patterns

```
instance Monoid a
    => Monoid (IO a) where
  mempty  = pure   mempty
  mappend = liftA2 mappend
```

Class instance patterns

```
instance Monoid b
    => Monoid (a -> b) where
  mempty  = pure   mempty
  mappend = liftA2 mappend
```

Class instance patterns

```
instance (Monoid a, Monoid b)
    => Monoid (a, b) where
  mempty  = pure    mempty
  mappend = liftA2 mappend
```

Class instance patterns

```
instance (Monoid a, Monoid b, Monoid c)
    => Monoid (a, b, c) where
  mempty  = pure    mempty
  mappend = liftA2 mappend
```

Class instance patterns

```
instance (Monoid a, Monoid b, Monoid c, Monoid d)
    => Monoid (a, b, c, d) where
  mempty  = pure    mempty
  mappend = liftA2 mappend
```

Class instance patterns

```
instance (Applicative f, Monoid a)
    => Monoid (f a) where
  mempty  = pure    mempty
  mappend = liftA2 mappend
  -- Can we abstract this pattern out?
```

“Solution”: use a newtype

“Solution”: use a newtype

```haskell
newtype App f a = MkApp { unApp :: f a }
```

“Solution”: use a newtype

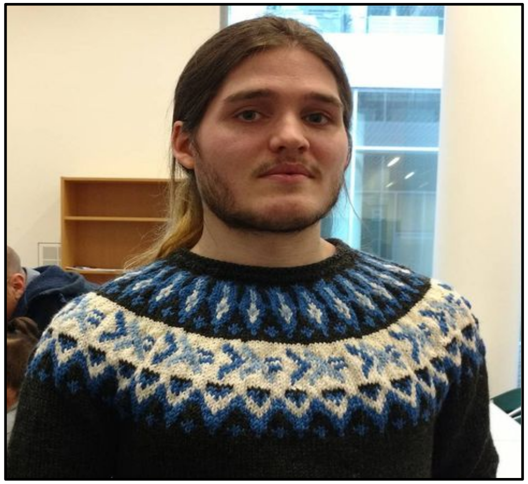
```
newtype App f a = MkApp { unApp :: f a }

instance (Applicative f, Monoid a)
    => Monoid (App f a) where
  mempty = MkApp (pure mempty)
  mappend (MkApp fa) (MkApp fb)
    = MkApp (liftA2 mappend fa fb)
```

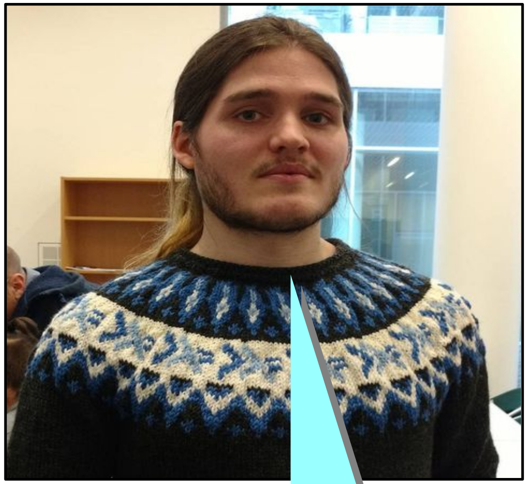
“Solution”: use a newtype

```
data Pair a = MkPair a a

instance Applicative Pair where ...
```

“Solution”: use a newtype

```
data Pair a = MkPair a a

instance Applicative Pair where ...

instance Monoid a => Monoid (Pair a) where
  mempty = unApp (mempty :: App Pair a)
  mappend p1 p2
    = unApp (mappend (MkApp p1) (MkApp p2)
               :: App Pair a)
  -- Agh! More boilerplate!
```





“`deriving` ought to be able to write this code for you!”

Can `deriving` save the day?

```
newtype Age = MkAge Int


instance Num Age where
  (MkAge a1) + (MkAge a2)
    = MkAge (a1 + a2)
  (MkAge a2) - (MkAge a2)
    = MkAge (a1 - a2)
  ...
```

Can `deriving` save the day?

```
newtype Age = MkAge Int
  deriving Num
```

Can `deriving` save the day?

```
data Pair a = MkPair a a


instance Monoid a => Monoid (Pair a) where
  mempty = unApp (mempty :: App Pair a)
  mappend p1 p2
    = unApp (mappend (MkApp p1) (MkApp p2)
             :: App Pair a)
```

Can `deriving` save the day?

```
data Pair a = MkPair a a
  deriving Monoid???
```

Our solution: generalize `deriving`! (again!)

Our solution: generalize `deriving`! (again!)

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
```

Our solution: generalize `deriving`! (again!)

```
{-# LANGUAGE GeneralizedGeneralizedNewtypeDeriving #-}
```

Our solution: generalize `deriving`! (again!)

~~`{-# LANGUAGE GeneralizedGeneralizedNewtypeDeriving #-}`~~

```
deriving via
```

`deriving via` in action

```
data Pair a = MkPair a a
  deriving Monoid via (App Pair a)
```

deriving via in action

```haskell
data Pair a = MkPair a a
  deriving Monoid via (App Pair a)

-- This code gets autogenerated:
instance Monoid a => Monoid (Pair a) where
  mempty = unApp (mempty :: App Pair a)
  mappend p1 p2
    = unApp (mappend (MkApp p1) (MkApp p2)
              :: App Pair a)
```

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

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

unsafeCoerce :: a -> b
```

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

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

Only typechecks if types a and b have the *same runtime representation*.

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

Only typechecks if types a and b have the *same runtime representation*.

```
newtype Age = MkAge Int
```

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

Only typechecks if types a and b have the *same runtime representation*.

```
newtype Age = MkAge Int

instance Coercible Age Int
instance Coercible Int Age
```

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

Only typechecks if types a and b have the *same runtime representation*.

```
newtype Age = MkAge Int

instance Coercible (Age -> Age) (Int -> Int)
instance Coercible (Int -> Int) (Age -> Age)
```

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

Only typechecks if types a and b have the *same runtime representation*.

```haskell
newtype Age = MkAge Int

succInt :: Int -> Int
succInt i = i + 1

succAge :: Age -> Age
succAge = coerce succInt
```

`deriving via`, revisited

```
data Pair a = MkPair a a
  deriving Monoid via (App Pair a)
```

deriving via, revisited

```
data Pair a = MkPair a a
  deriving Monoid via (App Pair a)

instance Monoid a => Monoid (Pair a) where
  mempty  = unApp (mempty :: App Pair a)
  mappend p1 p2
    = unApp (mappend (MkApp p1) (MkApp p2)
             :: App Pair a)
```

deriving via, revisited

```
data Pair a = MkPair a a
  deriving Monoid via (App Pair a)

instance Monoid a => Monoid (Pair a) where
  mempty  = coerce (mempty  :: App Pair a)
  mappend = coerce (mappend :: App Pair a
                            -> App Pair a
                            -> App Pair a)
```

`deriving via`, revisited

```
data Pair a = MkPair a a
  deriving Monoid via (App Pair a)

instance Monoid a => Monoid (Pair a) where
  mempty  = coerce (mempty  :: App Pair a)
  mappend = coerce (mappend :: App Pair a
                            -> App Pair a
                            -> App Pair a)
```

Typechecks since `Pair a` and `App Pair a` have the same runtime representation.

Use case: `QuickCheck`

`QuickCheck` is a library for testing random properties of Haskell programs.

Use case: `QuickCheck`

`QuickCheck` is a library for testing random properties of Haskell programs.

```
class Arbitrary a where
  arbitrary :: Gen a -- Generate random ‘a’ values
```

Use case: QuickCheck

QuickCheck is a library for testing random properties of Haskell programs.

```
class Arbitrary a where
  arbitrary :: Gen a -- Generate random ‘a’ values



> sample’ (arbitrary :: Gen Bool)
[False,False,False,True,True,True,False,True,
False,False,True]
```

Use case: QuickCheck

QuickCheck is a library for testing random properties of Haskell programs.

```
class Arbitrary a where
  arbitrary :: Gen a -- Generate random ‘a’ values



> sample’ (arbitrary :: Gen Int)
[0,1,-1,-6,6,-7,5,13,1,8,1]
```

Use case: `QuickCheck`

`QuickCheck` is a library for testing random properties of Haskell programs.

```
class Arbitrary a where
  arbitrary :: Gen a -- Generate random ‘a’ values




> sample’ (arbitrary :: Gen [Int])
[[],[],[3],[1],[1,1,-6,5,-5],[],[7,-11,7],[5],
[-16,15,-14,-12,5,-11],[6,1,-8,-16,9,1,15,4,-5,
-18,-15,-18,-2],[-16,17,9,-3,-13,-9,11,-18,
-6,8,1,-4,-5,-1,-17]]
```

Q: What if we want to generate random values subject to constraints?

Q: What if we want to generate random values subject to constraints?

A: Use newtypes!

```
newtype NonEmptyList a = NonEmpty [a]
```

Q: What if we want to generate random values subject to constraints?

A: Use newtypes!

```
newtype NonEmptyList a = NonEmpty [a]

instance Arbitrary a
    => Arbitrary (NonEmptyList a) where
  arbitrary = fmap NonEmpty
                (arbitrary `suchThat` (not . null))
```

```haskell
newtype Nums = MkNums [Int]
  deriving Arbitrary
```

```
> sample' (arbitrary :: Gen Nums)
[[],[0,-2],[],[0,-2,-3],[5,4,-5,5],[9,0],
[-5,1,-5,2,11]]
```

```haskell
newtype NonEmptyList a = NonEmpty [a]
  -- Generate non-empty lists
```

```haskell
newtype Nums = MkNums [Int]
  deriving Arbitrary
    via (NonEmptyList Int)

> sample' (arbitrary :: Gen Nums)
[[2,1],[1],[-3,2],[-6,3,-4,6],[-1,6,7,4,-3],
[2,10,9,-7,8,-9,-7,4,4],[12,5,5,9,10]]
```

```haskell
newtype NonEmptyList a = NonEmpty [a]
  -- Generate non-empty lists
newtype Positive a = MkPositive a
  -- Generate values x such that x > 0
```

```haskell
newtype Nums = MkNums [Int]
  deriving Arbitrary
    via (NonEmptyList (Positive Int))

> sample' (arbitrary :: Gen Nums)
[[2],[1,2],[3,4],[2,5],[1],[8,2,4,3,4,5,1,7],
[10,6,2,11,10,3,2,11,12]]
```

```haskell
newtype NonEmptyList a = NonEmpty [a]
  -- Generate non-empty lists
newtype Positive a = MkPositive a
  -- Generate values x such that x > 0
newtype Large a = MkLarge a
  -- Generate values biased towards large numbers
```

```haskell
newtype Nums = MkNums [Int]
  deriving Arbitrary
    via (NonEmptyList (Positive (Large Int)))

> sample' (arbitrary :: Gen Nums)
[[2],[2,1],[2,7,8,4],[11,13],
[8,40,17,57,16,51,88,58],[249,27],[511,642]]
```

`deriving via` lets you quickly write your type class instances with a high power-to-weight ratio.

- Allows effective use of newtypes without the awkwardness of wrapping/unwrapping them yourself
- Leverage existing tools in GHC in a way that feels natural
- Compose programming patterns by codifying them as newtypes, cheaply and cheerfully

`https://github.com/RyanGlScott/ghc/tree/deriving-via`