`DerivingVia`

or, How to Turn Hand-Written Instances into an Anti-Pattern

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

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

✉ `rgscott@indiana.edu`
`github.com/RyanGlScott`

A brief history of `deriving`

Haskell 98 Report (various authors):
Derive the Blessed Type Classes™

```haskell
data Grade = A | B | C | D | F

instance Eq Grade where
  A == A = True
  B == B = True
  ...
  _ == _ = False
```

A brief history of `deriving`

Haskell 98 Report (various authors):
Derive the Blessed Type Classes™

```
data Grade = A | B | C | D | F
  deriving Eq
```

A brief history of `deriving`

Haskell 98 Report (various authors):

Derive the Blessed Type Classes™

```
data Grade = A | B | C | D | F
  deriving ( Eq, Ord, Read, Show
           , Enum, Bounded, Ix )
```

A brief history of `deriving`

GHC extensions (various authors):
Derive these *other* blessed classes

```
{-# LANGUAGE DeriveDataTypeable #-}
{-# LANGUAGE DeriveGeneric #-}

data Grade = A | B | C | D | F
  deriving ( Data, Generic
           , ...
           )
```

A brief history of `deriving`

GHC extension (Peyton Jones, 2001):
Derive anything (through newtypes)

```
instance Num Int where ...

newtype Age = MkAge Int

instance Num Age where
  MkAge x + MkAge y = MkAge (x + y)
  ...
```

A brief history of `deriving`

GHC extension (Peyton Jones, 2001):
Derive anything (through newtypes)

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

instance Num Int where ...

newtype Age = MkAge Int
  deriving Num
```

A brief history of `deriving`

GHC extension (Magalhães, 2014):
Derive anything (through empty instances)

```
class Foo a where
  bar :: a -> String
  bar _ = “Sensible default”

data Baz = MkBaz
instance Foo Baz
```

A brief history of `deriving`

GHC extension (Magalhães, 2014):
Derive anything (through empty instances)

```
{-# LANGUAGE DeriveAnyClass #-}

class Foo a where
  bar :: a -> String
  bar _ = “Sensible default”

data Baz = MkBaz
  deriving Foo
```

A brief history of `deriving`

GHC extension (Scott, 2016):
Be explicit about how you derive things

```
{-# LANGUAGE DerivingStrategies #-}

newtype Age = MkAge Int
  deriving stock    (Eq, Ord)
  deriving newtype  Num
  deriving anyclass Bar
```

```
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
```

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

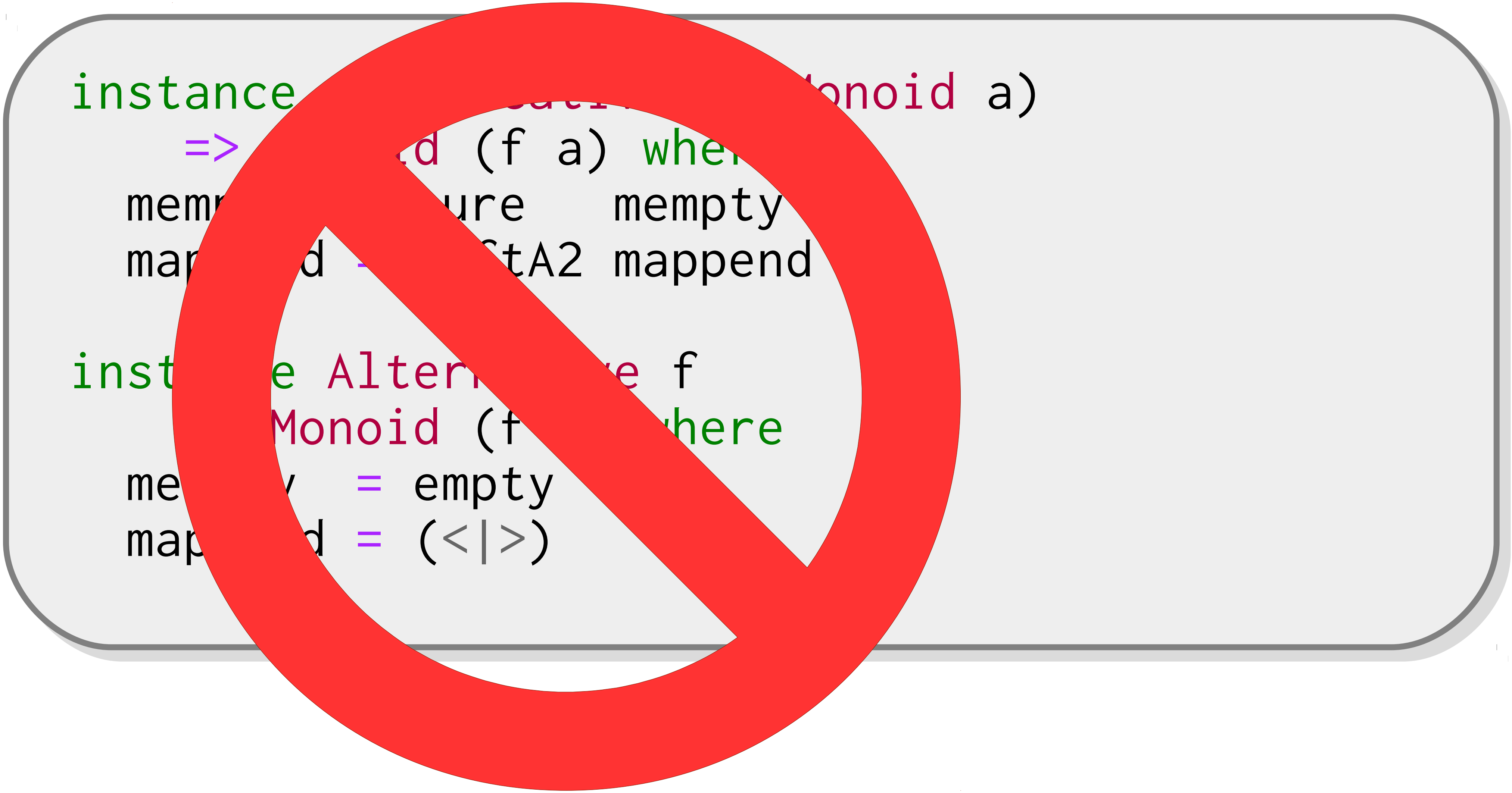
```
instance Monoid a
    => Monoid (ST s a) where
  mempty  = pure   mempty
  mappend = liftA2 mappend
```

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

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

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

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

instance Alternative f
    => Monoid (f a) where
  mempty  = empty
  mappend = (<|>)
```



```
instance                  onoid a)
    =>      d (f a) whe
  mem          ure   mempty
  map   d       tA2 mappend

inst    e Alter     e f
       Monoid (f     here
  me   y  = empty
  map   d = (<|>)
```

```haskell
instance (Applicative f, Monoid a)
    => Monoid (f a) where
  mempty  = pure   mempty
  mappend = liftA2 mappend
```

Can we abstract this pattern out without the pain of instance overlap?

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

Can we abstract this pattern out without the pain of instance overlap?

Well, sort of...

```haskell
newtype Ap f a = Ap { getAp :: f a }
```

Can we abstract this pattern out without the pain of instance overlap?

Well, sort of...

```haskell
newtype Ap f a = Ap { getAp :: f a }

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

Can we abstract this pattern out without the pain of instance overlap?

Well, sort of...

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

```haskell
instance Monoid a
    => Monoid (IO a) where
  mempty = getAp (mempty :: Ap IO a)
  mappend p1 p2
    = getAp (mappend (Ap p1) (Ap p2)
             :: Ap IO a)
```

```haskell
instance Monoid a
    => Monoid (IO a) where
  mempty = getAp (mempty :: Ap IO a)
  mappend p1 p2
    = getAp (mappend (Ap p1) (Ap p2)
                :: Ap IO a)
```

(╯°□°）╯︵ ┻━┻)

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

```
instance Monoid a
    => Monoid (IO a) where
  mempty = getAp (mempty :: Ap IO a)
  mappend p1 p2
    = getAp (mappend (Ap p1) (Ap p2)
                :: Ap IO a)
```

```
data IO a = ...
  deriving Monoid ???
```

```
data IO a = ...
  deriving Monoid via (Ap IO a)
```

Safe Zero-cost Coercions for Haskell

Joachim Breitner
Karlsruhe Institute of Technology
breitner@kit.edu

Richard A. Eisenberg
University of Pennsylvania
eir@cis.upenn.edu

Simon Peyton Jones
Microsoft Research
simonpj@microsoft.com

Stephanie Weirich
University of Pennsylvania
sweirich@cis.upenn.edu

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

instance Num Int where ...

newtype Age = MkAge Int
  deriving newtype Num
```

```haskell
instance Num Int where ...

newtype Age = MkAge Int

instance Num Age where
  MkAge x + MkAge y = MkAge (x + y)
  ...
```

```
instance Num Int where ...

newtype Age = MkAge Int

instance Num Age where
  (+) = unsafeCoerce
          ((+) :: Int -> Int -> Int)
```

`unsafeCoerce :: a -> b`

```
instance Num Int where ...

newtype Age = MkAge Int

instance Num Age where
  (+) = unsafeCoerce
          ((+) :: Int -> Int -> Int)
```

~~`unsafeCoerce :: a -> b`~~

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

```
instance Num Int where ...

newtype Age = MkAge Int

instance Num Age where
  (+) = {- ~~unsafeCoerce~~ -} coerce
          ((+) :: Int -> Int -> Int)
```

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

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

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

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

```haskell
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
```

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

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

```haskell
newtype Age = MkAge Int

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

```haskell
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
```

```
data IO a = ...
  deriving Monoid via (App IO a)
```

```
data IO a = ...
  deriving Monoid via (App IO a)

instance Monoid a => Monoid (IO a) where
  mempty  = coerce (mempty  :: Ap IO a)
  mappend = coerce (mappend :: Ap IO a
                            -> Ap IO a
                            -> Ap IO a)
```

```haskell
data IO a = ...
  deriving Monoid via (App IO a)

instance Monoid a => Monoid (IO a) where
  mempty  = coerce (mempty  :: Ap IO a)
  mappend = coerce (mappend :: Ap IO a
                             -> Ap IO a
                             -> Ap IO a)
```

Typechecks since `IO a` and `Ap IO a` have the same runtime representation.

`DerivingVia` is generalized `GeneralizedNewtypeDeriving`

`DerivingVia` is generalized `GeneralizedNewtypeDeriving`

```
newtype Age = MkAge Int
  deriving newtype Num

==>

instance Num Age where
  (+) = coerce ((+) :: Int -> Int -> Int)
  ...
```

`DerivingVia` is generalized `GeneralizedNewtypeDeriving`

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

==>

instance Num Age where
  (+) = coerce ((+) :: Int -> Int -> Int)
  ...
```

Case studies

- `QuickCheck`
- Excluding types in datatype-generic algorithms

Case study: `QuickCheck`

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

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

Case study: `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]
```

Case study: `QuickCheck`

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

```haskell
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]
```

Case study: `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!

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

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

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

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

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

```
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]]
```

```
newtype NonEmptyList a = NonEmpty [a]

newtype Positive a = MkPositive a
```

```
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]]
```

```
newtype NonEmptyList a = NonEmpty [a]

newtype Positive a = MkPositive a

newtype Large a = MkLarge a
```

```
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]]
```

Case studies

- `QuickCheck`
- **Excluding types in datatype-generic algorithms**

Case study: Excluding types

How can we derive instances for fields
with problematic types?

```
data ModIface {
  ...
, mi_fixities :: [(OccName,Fixity)]
, mi_fix_fn :: OccName -> Maybe Fixity
  -- ^ Cached lookup for 'mi_fixities'
  ...
}
```

Case study: Excluding types

How can we derive instances for fields
with problematic types?

```
data ModIface {
  ...
, mi_fixities :: [(OccName,Fixity)]
, mi_fix_fn :: OccName -> Maybe Fixity
  -- ^ Cached lookup for 'mi_fixities'
  ...
} deriving Eq
   via (Excluding ‘[OccName -> Maybe Fixity]
          ModIface)
```

Case study: Excluding types

How can we derive instances for fields
with problematic types?

```haskell
data ModIface {
  ...
, mi_fixities :: [(OccName,Fixity)]
, mi_fix_fn :: OccName -> Maybe Fixity
  -- ^ Cached lookup for 'mi_fixities'
  ...
} deriving Eq
   via (Excluding ‘[OccName -> Maybe Fixity]
          ModIface)
  deriving stock Generic
```

```haskell
import GHC.Generics

class GEq (excluded :: [Type]) f where
  geq :: f a -> f a -> Bool
```

```haskell
import GHC.Generics

class GEq (excluded :: [Type]) f where
  geq :: f a -> f a -> Bool




instance GEq excluded U1 where
  geq U1 U1 = True
```

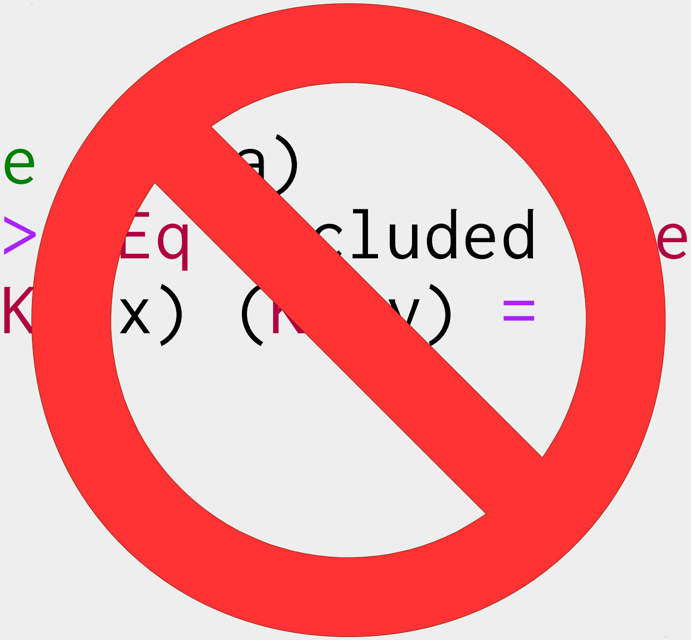
```haskell
import GHC.Generics

class GEq (excluded :: [Type]) f where
  geq :: f a -> f a -> Bool



instance (GEq excluded a, GEq excluded b)
      => GEq excluded (a :*: b) where
  geq (a1 :*: b1) (a2 :*: b2) =
    geq @excluded a1 a2 &&
    geq @excluded b1 b2
```

```haskell
import GHC.Generics

class GEq (excluded :: [Type]) f where
  geq :: f a -> f a -> Bool



instance (GEq excluded a, GEq excluded b)
       => GEq excluded (a :+: b) where
  geq (L1 a) (L1 b) = geq @excluded a b
  geq (R1 a) (R1 b) = geq @excluded a b
  geq _      _      = False
```

```
import GHC.Generics

class GEq (excluded :: [Type]) f where
  geq :: f a -> f a -> Bool



instance ???
      => GEq excluded (Rec0 a) where
  ???
```

```
import GHC.Generics

class GEq (excluded :: [Type]) f where
  geq :: f a -> f a -> Bool



instance (Eq a)
      => GEq excluded (Rec0 a) where
  geq (K1 x) (K1 y) = (x == y)
```

```
import GHC.Generics

class GEq (excluded :: [Type]) f where
  geq :: f a -> f a -> Bool



instance     a)
      =>  Eq   cluded   ec0 a) where
  geq (K  x) (K  y) =    == y)
```



```
import GHC.Generics

class GEq (excluded :: [Type]) f where
  geq :: f a -> f a -> Bool



instance ( Unless (Elem a excluded) (Eq a)
         , ??? )
      => GEq excluded (Rec0 a) where
  geq (K1 x) (K1 y) = ???
```

im

```
type family
  Unless (a :: Bool) (b :: Constraint)
    :: Constraint where
  Unless True  _ = ()
  Unless False b = b

type family
  Elem (x :: a) (xs :: [a])
    :: Bool where
  Elem _ '[]    = False
  Elem x (x:_)  = True
  Elem x (y:xs) = Elem x xs
```

```haskell
import GHC.Generics

class GEq (excluded :: [Type]) f where
  geq :: f a -> f a -> Bool



instance ( Unless (Elem a excluded) (Eq a)
         , ??? )
      => GEq excluded (Rec0 a) where
  geq (K1 x) (K1 y) = ???
```

```haskell
import GHC.Generics

class GEq (excluded :: [Type]) f where
  geq :: f a -> f a -> Bool



instance ( Unless (Elem a excluded) (Eq a)
         , SBoolI (Elem a excluded) )
      => GEq excluded (Rec0 a) where
  geq (K1 x) (K1 y) = ???
```

```
imp
```

```
data SBool :: Bool -> Type where
  SFalse :: SBool False
  STrue  :: SBool True
```

```
imp
```

```haskell
data SBool :: Bool -> Type where
  SFalse :: SBool False
  STrue  :: SBool True

class SBoolI (b :: Bool) where
  sbool :: SBool b
instance SBoolI False where
  sbool = SFalse
instance SBoolI True where
  sbool = STrue
```

```haskell
import GHC.Generics

class GEq (excluded :: [Type]) f where
  geq :: f a -> f a -> Bool



instance ( Unless (Elem a excluded) (Eq a)
         , SBoolI (Elem a excluded) )
      => GEq excluded (Rec0 a) where
  geq (K1 x) (K1 y) = ???
```

```haskell
import GHC.Generics

class GEq (excluded :: [Type]) f where
  geq :: f a -> f a -> Bool



instance ( Unless (Elem a excluded) (Eq a)
         , SBoolI (Elem a excluded) )
      => GEq excluded (Rec0 a) where
  geq (K1 x) (K1 y) =
     case sbool @(Elem a excluded) of
         SFalse -> (x == y)
         STrue  -> True
```

```haskell
import GHC.Generics

class GEq (excluded :: [Type]) f where
  geq :: f a -> f a -> Bool




newtype Excluding :: [Type] -> Type
                  -> Type where
  Excluding :: a -> Excluding excluded a
```

```haskell
import GHC.Generics

class GEq (excluded :: [Type]) f where
  geq :: f a -> f a -> Bool




instance (Generic a, GEq excluded (Rep a))
        => Eq (Excluding excluded a) where
  Excluding x == Excluding y =
    geq @excluded (from x) (from y)
```

```
data ModIface {
  ...
, mi_fixities :: [(OccName,Fixity)]
, mi_fix_fn :: OccName -> Maybe Fixity
  -- ^ Cached lookup for 'mi_fixities'
  ...
} deriving Eq
   via (Excluding ‘[OccName -> Maybe Fixity]
          ModIface)
  deriving stock Generic
```

`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

Debuts in GHC 8.6!