

DerivingVia

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

Baldur Blöndal

Andres Löh¹

Ryan Scott²

¹Well-Typed LLP 

²Indiana University 

Haskell Symposium 2018
St. Louis, MO

```
newtype Pair = Pair (Int, Int)
```

```
newtype Pair = Pair (Int, Int)
  deriving Monoid
  via (Sum Int, Product Int)
```

Use deriveVia #511

Merged phadej merged 2 commits into master from derive-via on Feb 15

Conversation 0 Commits 2 Checks 0 Files changed 65 +657 -1,048



phadej commented on Feb 13

Member +

No description provided.

phadej added some commits on Feb 13

Use deriveVia

096ebe3

Remove GHC-8.0.2 job

22e3a96

Reviewers

No reviews

Assignees

No one assigned

Labels

None yet

+657 -1,048 ■ ■ ■ ■ ■

Use deriveVia #511

Merged phadej merged 2 commits into master from derive-via on Feb 15

+657 -1,048 ■ ■ ■ ■ ■



phadej commented on Feb 13

Member + 😊

No description provided.

Reviewers

No reviews

Assignees

No one assigned

Labels

None yet

phadej added some commits on Feb 13

Use deriveVia ✗ 096ebe3

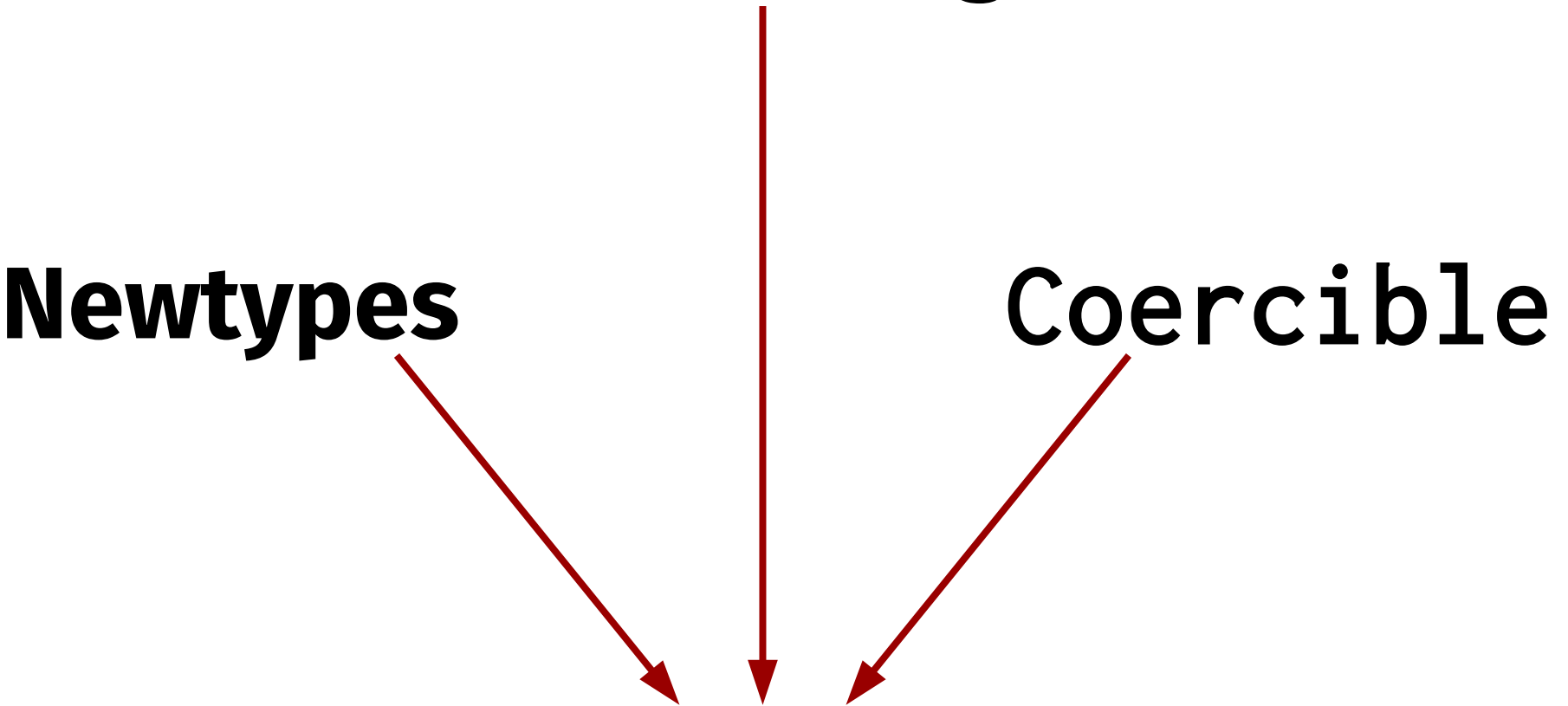
Remove GHC-8.0.2 job ✗ 22e3a96

deriving

Newtypes

Coercible

DerivingVia



deriving

deriving

```
deriving ( Eq, Ord, Read, Show, ... )
```


deriving

```
deriving ( Eq, Ord, Read, Show, ... )
```

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

deriving

```
deriving ( Eq, Ord, Read, Show, ... )
```

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

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

```
{-# LANGUAGE StandaloneDeriving #-}
```

deriving

```
deriving ( Eq, Ord, Read, Show, ... )
```

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

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

```
{-# LANGUAGE StandaloneDeriving #-}
```

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

deriving

```
deriving ( Eq, Ord, Read, Show, ... )
```

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

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

```
{-# LANGUAGE StandaloneDeriving #-}
```

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

```
{-# LANGUAGE DeriveAnyClass #-}
```

deriving

```
deriving ( Eq, Ord, Read, Show, ... )
```

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

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

```
{-# LANGUAGE StandaloneDeriving #-}
```

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

```
{-# LANGUAGE DeriveAnyClass #-}
```

```
{-# LANGUAGE DerivingStrategies #-}
```

```
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
```

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

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

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

```
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)
```

```
instance Monoid a
  => Monoid (IO a) where
  mempty = pure mempty
  mappend p1 p2
    = liftA2 mappend p1 p2
```

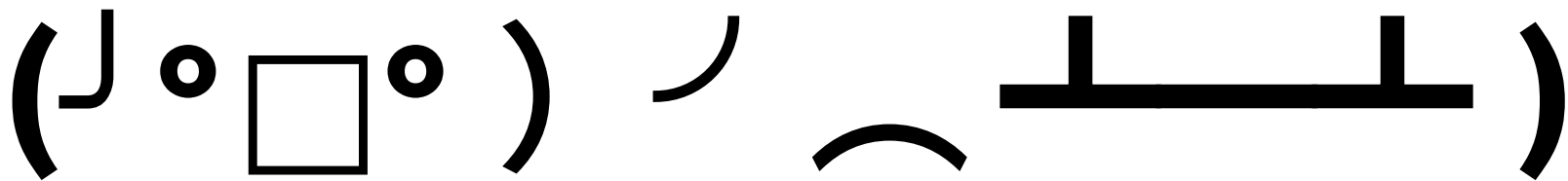


```
instance Monoid a
  => Monoid (IO a) where
  mempty = getAp (mempty :: Ap IO a)
  mappend p1 p2
    = getAp (mappend (Ap p1) (Ap p2)
              :: Ap IO a)
```

```

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)

```



```
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)
```

```
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 via (Ap IO a)
```

```
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 via (Ap IO a)
```

???

deriving

Newtypes

Coercible

DerivingVia

deriving

Newtypes

Coercible

DerivingVia

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

Abstract

Generative type abstractions – present in Haskell, OCaml, and other languages – are useful concepts to help prevent programmer errors. They serve to create new types that are distinct at compile time but share a run-time representation with some base type. We present a new mechanism that allows for zero-cost conversions between generative type abstractions and their representations, even when such types are deeply nested. We prove type safety in the presence of these conversions and have implemented our work in GHC.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features—abstract data types; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Type structure

Keywords Haskell; Coercion; Type class; Newtype deriving

```
module Html( HTML, text, unMk, ... ) where
  newtype HTML = Mk String
  unMk :: HTML → String
  unMk (Mk s) = s
  text :: String → HTML
  text s = Mk (escapeSpecialCharacters s)
```

Figure 1. An abstraction for HTML values

String will not be accepted by a function expecting an HTML. The constructor `Mk` converts a `String` to an `HTML` (see function `text`), while using `Mk` in a pattern converts in the other direction (see function `unMk`). By exporting the type `HTML`, but not its data constructor, module `Html` ensures that the type `HTML` is *abstract* – clients cannot make arbitrary strings into `HTML` – and thereby prevent cross-site scripting attacks.

Coercible

Special constraint witnessing the fact that two types have the same **runtime representation**.

Coercible

Special constraint witnessing the fact that two types have the same **runtime representation**.

Driving force: newtypes

```
newtype Age = MkAge Int
```

Coercible

Special constraint witnessing the fact that two types have the same **runtime representation**.

Driving force: newtypes

```
newtype Age = MkAge Int  
  
instance Coercible Age Int  
instance Coercible Int Age
```

Coercible

Special constraint witnessing the fact that two types have the same **runtime representation**.

Driving force: newtypes

```
newtype Age = MkAge Int
```

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

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

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

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

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

```
newtype Age = MkAge Int
```

```
succInt :: Int -> Int
```

```
succInt i = i + 1
```

```
succAge :: Age -> Age
```

```
succAge (MkAge i) = Age (succInt i)
```

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

```
newtype Age = MkAge Int
```

```
succInt :: Int -> Int
```

```
succInt i = i + 1
```

```
succAge :: Age -> Age
```

```
succAge = coerce succInt
```



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

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

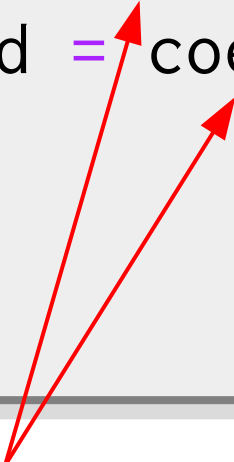
instance Monoid a => Monoid (IO a) where
  mempty   = coerce (mempty   :: Ap IO a)
  mappend = coerce (mappend  :: Ap IO a
                    -> Ap IO a
                    -> Ap IO a)
```

```

data IO a = ...
  deriving Monoid via (Ap 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!

deriving

Newtypes

Coercible

DerivingVia

deriving

Newtypes

Coercible

DerivingVia

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)
  ...
```


DerivingVia can replace DefaultSignatures

DerivingVia can replace DefaultSignatures

```
class Pretty a where  
  pPrint :: a -> Doc
```

DerivingVia can replace DefaultSignatures

```
class Pretty a where
  pPrint :: a -> Doc
  default pPrint
    :: Show a
    => a -> Doc
  pPrint = text . show
```

DerivingVia can replace DefaultSignatures

```
class Pretty a where
  pPrint :: a -> Doc
  default pPrint
    :: Show a
    => a -> Doc
  pPrint = text . show

data Foo deriving (Show)
instance Pretty Foo
```

DerivingVia can replace DefaultSignatures

```
class Pretty a where
  pPrint :: a -> Doc
  default pPrint
    :: (Generic a, GPretty (Rep a))
    => a -> Doc
  pPrint = genericPPrint

data Foo deriving (Generic)
instance Pretty Foo
```

DerivingVia can replace DefaultSignatures

```
class Pretty a where
  pPrint :: a -> Doc
  default pPrint
    :: ???
    => a -> Doc
  pPrint = -- Which is the One True Default?
```

```
data Foo deriving (Show, Generic)
instance Pretty Foo
```

DerivingVia can replace DefaultSignatures

```
class Pretty a where  
  pPrint :: a -> Doc  
  default pPrint  
  ???  
  pPrint :: a -> Doc
```

Which is the One True Default?

```
data Foo deriving (Show, Generic)  
instance Pretty Foo
```

DerivingVia can replace DefaultSignatures

```
newtype ShowPPrint a = ShowPPrint a  
newtype GenericPPrint a = GenericPPrint a
```


DerivingVia can replace DefaultSignatures

```
newtype ShowPPrint a = ShowPPrint a
newtype GenericPPrint a = GenericPPrint a

instance Show a
  => Pretty (ShowPPrint a) where
  pPrint (ShowPPrint x) = text (show x)
instance Generic a
  => Pretty (GenericPPrint a) where
  pPrint (GenericPPrint x) = genericPPrint x
```

DerivingVia can replace DefaultSignatures

```
data Foo  
  deriving (Show, Generic)
```

DerivingVia can replace DefaultSignatures

```
data Foo
  deriving (Show, Generic)
  deriving Pretty via (ShowPPrint Foo)
```

DerivingVia can replace DefaultSignatures

```
data Foo
  deriving (Show, Generic)
-- deriving Pretty via (ShowPPrint Foo)
deriving Pretty via (GenericPPrint Foo)
```

DerivingVia can replace DefaultSignatures

```
data Foo
  deriving (Show, Generic)
  deriving Pretty via (ShowPPrint Foo)
  -- deriving Pretty via (GenericPPrint Foo)
```

In the paper...

- Derive via things that aren't inter-Coercible (using datatype-generic programming)
- Interaction with StandaloneDeriving, associated type families, MultiParamTypeClasses, etc.
- DerivingVia as a way to derive asymptotically faster code
- DerivingVia as a technique for making it easier to retrofit superclass constraints

data Slides deriving Conclusion via WrapUp Slides

- Simple extension, but with powerful consequences
- Compositional, configurable, cheap, and cheerful
- Encourages code reuse and codifying patterns into named ideas that others can refer to
- Leverages existing technology in GHC

Debuts in GHC 8.6!