# Deriving Via

Haskell eXchange 2018

Andres Löh, Baldur Blöndal, Ryan Scott

2018-10-12

Well-Typed

The Haskell Consultants

```haskell
data Status = Green | Yellow | Red
```

```haskell
data Status = Green | Yellow | Red
  deriving Eq
```

Well-Typed

```haskell
data Status = Green | Yellow | Red
  deriving (Eq, Ord, Show, Enum, Bounded)
```

Well-Typed

```haskell
data Status = Green | Yellow | Red
  deriving (Eq, Ord, Show, Enum, Bounded)
  deriving Generic
```

Well-Typed

```haskell
data Status = Green | Yellow | Red
  deriving (Eq, Ord, Show, Enum, Bounded)
  deriving Generic
  deriving (FromJSON, ToJSON)
```

```haskell
data Status = Green | Yellow | Red
  deriving (Eq, Ord, Show, Enum, Bounded)
  deriving Generic
  deriving (FromJSON, ToJSON)   -- is this really what we want?
```

```haskell
data Status = Green | Yellow | Red
  deriving (Eq, Ord, Show, Enum, Bounded)
  deriving Generic
  deriving (FromJSON, ToJSON)
```

What about `Semigroup` (and `Monoid` )?

```haskell
data Status = Green | Yellow | Red
  deriving (Eq, Ord, Show, Enum, Bounded)
  deriving Generic
  deriving (FromJSON, ToJSON)
```

What about `Semigroup` (and `Monoid`)?

Several reasonable options:

- always take first,
- always take last,
- always take "worst",
- ...

```haskell
data Status = Green | Yellow | Red
  deriving (Eq, Ord, Show, Enum, Bounded)
  deriving Generic
  deriving (FromJSON, ToJSON)
  deriving Semigroup
    via First Status  -- always take first
```

Well-Typed

```haskell
data Status = Green | Yellow | Red
  deriving (Eq, Ord, Show, Enum, Bounded)
  deriving Generic
  deriving (FromJSON, ToJSON)
  deriving Semigroup
    via Last Status  -- always take last
```

```haskell
data Status = Green | Yellow | Red
  deriving (Eq, Ord, Show, Enum, Bounded)
  deriving Generic
  deriving (FromJSON, ToJSON)
  deriving Semigroup
    via Max Status  -- always take “worst”
```

```haskell
data Status = Green | Yellow | Red
  deriving (Eq, Ord, Show, Enum, Bounded)
  deriving Generic
  deriving (FromJSON, ToJSON)
  deriving Semigroup
    via Max Status  -- always take "worst"
```

Rule `Max a` :

```haskell
Ord a => Semigroup a
```

Well-Typed

```haskell
data Status = Green | Yellow | Red
  deriving (Eq, Ord, Show, Enum, Bounded)
  deriving Generic
  deriving (FromJSON, ToJSON)
  deriving (Semigroup, Monoid)
    via Max Status   -- always take “worst”, default to “best”
```

```haskell
data Status = Green | Yellow | Red
  deriving (Eq, Ord, Show, Enum, Bounded)
  deriving Generic
  deriving (FromJSON, ToJSON)
  deriving (Semigroup, Monoid)
    via Max Status   -- always take “worst”, default to “best”
```

Rule `Max a` :

```haskell
(Ord a, Bounded a) => Monoid a
```

Well-Typed

# Deriving Via

# Deriving Via

- Give **names** to **instance rules**.

# Deriving Via

- Give **names** to **instance rules**.

- **Apply** named instance rules in `via` clauses to derive instances.

Well-Typed

# Deriving Via

- Give **names** to **instance rules**.

  Use `newtype` s and `instance` s on them for named rules.

- **Apply** named instance rules in `via` clauses to derive instances.

# Deriving Via

- Give **names** to **instance rules**.

  Use `newtype`s and `instance`s on them for named rules.

- **Apply** named instance rules in `via` clauses to derive instances.

  Compiler applies **(safe) coercions** between representationally equal types to get the instances.

Rules `Max a` :

```
Ord a => Semigroup a
(Ord a, Bounded a) => Monoid a
```

Rules `Max a` :

```
Ord a => Semigroup a
(Ord a, Bounded a) => Monoid a
```

```
newtype Max a = Max {getMax :: a}
    -- already in Data.Semigroup
```

Rules `Max a` :

```
Ord a => Semigroup a
(Ord a, Bounded a) => Monoid a
```

```
newtype Max a = Max {getMax :: a}
    -- already in Data.Semigroup
```

```
instance Ord a => Semigroup (Max a) where
  Max a1 <> Max a2 = Max (a1 `max` a2)
```

Well-Typed

Rules `Max a`:

```
Ord a => Semigroup a
(Ord a, Bounded a) => Monoid a
```

```
newtype Max a = Max {getMax :: a}
    -- already in Data.Semigroup
```

```
instance Ord a => Semigroup (Max a) where
  Max a1 <> Max a2 = Max (a1 `max` a2)
```

```
instance (Ord a, Bounded a) => Monoid (Max a) where
  mempty = Max minBound
```

```
data Status = Green | Yellow | Red
  deriving (Eq, Ord)
  deriving Semigroup
    via Max Status
```

```haskell
data Status = Green | Yellow | Red
  deriving (Eq, Ord)
  deriving Semigroup
    via Max Status
```

Derived instance:

```haskell
instance Semigroup Status where
  (<>) =
    coerce
      @(Max Status -> Max Status -> Max Status)
      @(Status -> Status -> Status)
      (<>)
```

```haskell
coerce :: Coercible a b => a -> b  -- from Data.Coerce
```

# What has changed?

- **Scrap your boilerplate**

# What has changed?

- **Scrap your boilerplate**

  In particular if:

  - classes have many methods,
  - you can derive several instances via one rule.

# What has changed?

- **Scrap your boilerplate**

  In particular if:

  - classes have many methods,
  - you can derive several instances via one rule.

- **Write down instance rules**

  (This is already happening.)

# What has changed?

- **Scrap your boilerplate**

  In particular if:

  - classes have many methods,
  - you can derive several instances via one rule.

- **Write down instance rules**

  (This is already happening.)

- **Justify your instances**

# Examples

# Generalised newtype deriving

```
newtype Amount = MkAmount Rational
  deriving (Num, Fractional, Eq, Enum, Ord, Show)
    via Rational
```

# Monads are applicative functors

Rules `FromMonad m`:

```
Monad m => Functor m
Monad m => Applicative m
```

# Monads are applicative functors

Rules `FromMonad m`:

```
Monad m => Functor m
Monad m => Applicative m
```

```
newtype FromMonad m a = FM (m a)
```

Well-Typed

# Monads are applicative functors

Rules `FromMonad m`:

```
Monad m => Functor m
Monad m => Applicative m
```

```
newtype FromMonad m a = FM (m a)
```

```
instance Monad m => Functor (FromMonad m) where
  fmap f (FM m) = FM (m >>= return . f)
```

```
instance Monad m => Applicative (FromMonad m) where
  pure a        = FM (return a)
  FM f <*> FM x =
    FM (f >>= \ rf -> x >>= \ rx -> return (rf rx))
```

# Monads are applicative functors

```
data Maybe a = Nothing | Just a
  deriving (Functor, Applicative)
    via (FromMonad Maybe)
```

```
instance Monad Maybe where
  return        = Just
  Just m  >>= k = k m
  Nothing >>= _ = Nothing
```

# Getters

```haskell
data Event =
  MkEvent
    { status  :: Status
    , handler :: IO ()
    }
```

# Getters

```
data Event =
  MkEvent
    { status  :: Status
    , handler :: IO ()
    }
```

# Getters

```haskell
data Event =
  MkEvent
    { status  :: Status
    , handler :: IO ()
    }
  deriving Generic
  deriving Eq
    via Field "status" Event
```

# Getters

```haskell
newtype Field (n :: Symbol) (a :: Type) =
  Field {unField :: a}
```

```haskell
instance (HasField' n a b, Eq b) => Eq (Field n a) where
  (==) = (==) `on` getField @n . unField
```

# Custom enumeration types

```
data Status = Green | Yellow | Red
  deriving (Eq, Ord, Show, Enum, Bounded)
  deriving Generic
  deriving (FromJSON, ToJSON)   -- is this really what we want?
```

# Custom enumeration types

```
data Status = Green | Yellow | Red
  deriving (Eq, Ord, Show, Enum, Bounded)
  deriving (Generic)
  deriving (FromJSON, ToJSON)
    via CustomEnum '["green", "yellow", "yed"] Status
```

# Custom enumeration types

```
newtype CustomEnum (ls :: [Symbol]) (a :: Type) =
  MkCustomEnum a
```

```
instance ModifiedGeneric ls a => FromJSON a
instance ModifiedGeneric ls a => ToJSON a
```

# Conclusions

- Available now as `-XDerivingVia` in GHC 8.6.1.
- Lightweight feature, reusing existing language concepts.
- Generalises generalised newtype deriving (and, to some extent, default signatures).
- The real fun starts once you consider that instance rules can have parameters and be conposed.