The Great Power of `newtypes`

@mr_konn

https://konn-san.com

Slides are available at: `http://bit.ly/derivia`

Example codes are on GitHub: `konn/newtype-talk-five`

Self Introduction

- Hiromi ISHII（@mr_konn)
- Doctoral Candidate in Mathematics
  - Research Area: Mathematical Logic, Computer Science
- Writing and teaching Haskell for 12 years…

Great!

The Great Power of `newtype`s

Roles, Safe zero-cost coercions, and DerivingVia

〜Monoid & Foldable included〜

newtype

newtype

```
newtype Foo α = Bar α
newtype Id = MkId Word
```

newtype

```
newtype Foo α = Bar α
newtype Id = MkId Word
```

- A type with a single constructor and field.

newtype

```
newtype Foo α = Bar α
newtype Id = MkId Word
```

- A type with a single constructor and field.
- Has the **<u>Same representation</u>** as its only field

newtype

```haskell
newtype Foo α = Bar α
newtype Id = MkId Word
```

- A type with a single constructor and field.
- Has the **Same representation** as its only field
    - Distinguished from the original type at type-level, but has the same memory representation as the original, and evaluated strictly.

Typical Newbie Question

Typical Newbie Question

- 🤔 "What is the difference from **`data`**?"

Typical Newbie Question

- 🤔 "What is the difference from **`data`**?"
- 🧐 "It's efficient thanks to its representation."

Typical Newbie Question

- 🤔 "What is the difference from **`data`**?"
- 🧐 "It's efficient thanks to its representation."
- 🤔 "It doesn't matter much to me... I'd rather use **`data`**."

Typical Newbie Question

- 🤔 "What is the difference from **`data`**?"
- 🧐 "It's efficient thanks to its representation."
- 🤔 "It doesn't matter much to me... I'd rather use **`data`**."
- 😌 "Well, we have `-funpack-strict-fields` anyhow..."

Really?

3 Roles of `newtype`s

3 Roles of **newtypes**

- Implementation Hiding

```
module Data.Id (Id ()) where
newtype Id = MkId Word
```

3 Roles of **newtypes**

- Implementation Hiding

```haskell
module Data.Id (Id ()) where
newtype Id = MkId Word
```

Distinguished at type-level, but just a `Word` `internally`

3 Roles of **newtypes**

- Implementation Hiding

```
module Data.Id (Id ()) where
newtype Id = MkId Word
```

Hide data cons `MkId` `outside the module`

Distinguished at type-level, but just a `Word internally`

3 Roles of **newtypes**

- Implementation Hiding

Hide data cons MkId outside the module

```
module Data.Id (Id ()) where
newtype Id = MkId Word
```

Distinguished at type-level, but just a Word internally

- Implementation Sharing

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
newtype Id = MkId Word deriving (Num, Eq)
```

3 Roles of **newtypes**

- Implementation Hiding

```haskell
module Data.Id (Id ()) where
newtype Id = MkId Word
```

Hide data cons `MkId` `outside the module`

Distinguished at type-level, but just a `Word internally`

- Implementation Sharing

```haskell
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
newtype Id = MkId Word deriving (Num, Eq)
```

Underivable in Haskell 10, but we can **share** `Word`'s impl!

3 Roles of **newtypes**

- Implementation Hiding

```
module Data.Id (Id ()) where
newtype Id = MkId Word
```

Hide data cons `MkId` `outside the module`

Distinguished at type-level, but just a `Word` `internally`

- Implementation Sharing

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
newtype Id = MkId Word deriving (Num, Eq)
```

Underivable in Haskell 10, but we can **share** `Word`'s impl!

Evolves more in DerivingVia

3 Roles of **newtypes**

- Implementation Hiding

```haskell
module Data.Id (Id ()) where
newtype Id = MkId Word
```

Hide data cons `MkId` outside the module

Distinguished at type-level, but just a `Word` internally

- Implementation Sharing

```haskell
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
newtype Id = MkId Word deriving (Num, Eq)
```

- Implementation Selection

Underivable in Haskell 10, but we can **share** `Word`'s impl!

Evolves more in DerivingVia

3 Roles of **newtypes**

- Implementation Hiding

Hide data cons MkId outside the module

```
module Data.Id (Id ()) where
newtype Id = MkId Word
```

- Implementation Sharing

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
newtype Id = MkId Word deriving (Num, Eq)
```

- **Implementation Selection**

Underivable in Haskell 10, but we can **share** Word's impl!

Evolves more in DerivingVia

3 Roles of **newtypes**

- Implementation Hiding

Hide data cons `MkId` `outside the module`

```
module Data.Id (Id ()) where
newtype Id = MkId Word
```

Distinguished at type-level, but just a `Word internally`

- Implementation Sharing

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
newtype Id = MkId Word deriving (Num, Eq)
```

- **Implementation Selection**

Underivable in Haskell 10, but we can **share** `Word`'s impl!

Evolves more in DerivingVia

Implementation Selection

Implementation Selection

Monoid & Foldable as Examples

Exercise: List Scanning

Q. Given a list of integers, calculates its maximum and total sum by scanning list **exactly once**.

※ Do not use `foldl` or `folds` packages…

Typical Answer

Typical Answer

- 😁 Folds!

Typical Answer

😁 Folds!

```
aggregate :: [ℕ] → (Maybe ℕ, ℕ)
aggregate = foldr
  (λ a (m, s) → (Just a `max` m, a + s))
  (Nothing, 0)
```

※ ℕ is short for Integer

Typical Answer

😁 Folds!

```
aggregate :: [ℕ] → (Maybe ℕ, ℕ)
aggregate = foldr
   (λ a (m, s) → (Just a `max` m, a + s))
   (Nothing, 0)
```

※ ℕ is short for Integer

🤔 We have similar operations on both sides…

Typical Answer

😁 Folds!

```
aggregate :: [ℕ] → (Maybe ℕ, ℕ)
aggregate = foldr
   (λ a (m, s) → (Just a `max` m, a + s))
   (Nothing, 0)
```

※ ℕ is short for Integer

🤔 We have similar operations on both sides…

Typical Answer

😁 Folds!

```haskell
aggregate :: [ℕ] → (Maybe ℕ, ℕ)
aggregate = foldr
  (λ a (m, s) → (Just a `max` m, a + s))
  (Nothing, 0)
```

Map + Binary Operation

※ ℕ is short for Integer

🤔 We have similar operations on both sides…

Typical Answer

- 😁 Folds!

```
aggregate :: [ℕ] → (Maybe ℕ, ℕ)
aggregate = foldr
  (λ a (m, s) → (Just a `max` m, a + s))
  (Nothing, 0)
```

Map + Binary Operation

※ ℕ is short for Integer

- 🤔 We have similar operations on both sides…

Typical Answer

- 😁 Folds!

```
aggregate :: [ℕ] → (Maybe ℕ, ℕ)
aggregate = foldr
  (λ a (m, s) → (Just a `max` m, a + s))
  (Nothing, 0)
```

Units

Map + Binary Operation

※ ℕ is short for Integer

- 🤔 We have similar operations on both sides…

Typical Answer

😁 Folds!

```
aggregate :: [ℕ] → (Maybe ℕ, ℕ)
aggregate = foldr
   (λ a (m, s) → (Just a `max` m, a + s))
   (Nothing, 0)
```

Units

Map + Binary Operation

※ ℕ is short for Integer

🤔 We have similar operations on both sides…

😂 Monoids!

Monoids

- An operation which can be **<u>computed both from left and right</u>**, with **<u>unit element</u>**:

$$x \bullet (y \bullet z) = (x \bullet y) \bullet z$$

$$x \bullet \varepsilon = x = \varepsilon \bullet x$$

- Both `` `max` `` and `(+)`
  - … can be computed from either left or right,
  - … has `Nothing` (no max) and `0` as units.
- Mapping to monoid + folding $\leadsto$ **<u>`Foldable`</u>**!

Foldable class

```
class Foldable t where
 foldMap :: Monoid m ⇒ (a → m) → t a → m

 …
```

Foldable class

Map to Monoid
+
Fold left-to-right

```
class Foldable t where
  foldMap :: Monoid m ⇒ (a → m) → t a → m

  …
```

Foldable class

Map to Monoid
+
Fold left-to-right

```
class Foldable t where
  foldMap :: Monoid m ⇒ (a → m) → t a → m

  …
```

😁 It suffices to make max, (+) fit in this framework

Foldable class

Map to Monoid
+
Fold left-to-right

```
class Foldable t where
  foldMap :: Monoid m ⇒ (a → m) → t a → m

  …
```

- 😁 It suffices to make max, (+) fit in this framework
- 🤔 We can have at most one instance for `Monoid Word`…

Foldable class

Map to Monoid
+
Fold left-to-right

```
class Foldable t where
  foldMap :: Monoid m ⇒ (a → m) → t a → m

  …
```

- 😁 It suffices to make max, (+) fit in this framework
- 🤔 We can have at most one instance for `Monoid Word`…
- 😁 Implementation Selection using **`newtype`**s!

Example instances

```haskell
newtype Sum a = Sum { getSum :: a}
  deriving (Num, Integral)

instance Num a ⇒ Monoid (Sum a) where

  (<>) = (+); ε = 0

newtype Max a = Max { getMax :: a }
  deriving (Num, Integral, Ord)

instance Ord a ⇒ Semigroup (Max a) where

  (<>) = max
```

Example instances

```haskell
newtype Sum a = Sum { getSum :: a}
  deriving (Num, Integral)

instance Num a ⇒ Monoid (Sum a) where
  (<>) = (+); ε = 0

newtype Max a = Max { getMax :: a }
  deriving (Num, Integral, Ord)

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

Monoid of numeric addition

Example instances

```haskell
newtype Sum a = Sum { getSum :: a}
  deriving (Num, Integral)

instance Num a ⇒ Monoid (Sum a) where
  (<>) = (+); ε = 0

newtype Max a = Max { getMax :: a }
  deriving (Num, Integral, Ord)

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

Monoid of numeric addition

Semigroup by taking max

Example instances

```haskell
newtype Sum a = Sum { getSum :: a}
  deriving (Num, Integral)

instance Num a ⇒ Monoid (Sum a) where
  (<>) = (+); ε = 0

newtype Max a = Max { getMax :: a }
  deriving (Num, Integral, Ord)

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

Monoid of numeric addition

Impl. sharing by GND

Semigroup by taking max

Practical Remark: 1

```haskell
newtype Max a = Max a
instance Ord a ⇒ Semigroup (Max a)
instance Bounded a ⇒ Monoid (Max a)
```

- Only **bounded types** can be monoids! (We need maximum element to have the unit)
- We still have a **Semiring**, which lacks units.
  - We have to convert it to monoid to use with Foldable…

Practical Remark: 2

```
newtype Option a = Option (Maybe a)
instance Semigroup a ⇒ Monoid (Option a)
```

Practical Remark: 2

```haskell
newtype Option a = Option (Maybe a)
instance Semigroup a ⇒ Monoid (Option a)
```

`Option` adjoins unit, turning semigroups into monoids.

Practical Remark: 2

```haskell
newtype Option a = Option (Maybe a)
instance Semigroup a ⇒ Monoid (Option a)
```

`Option` adjoins unit, turning semigroups into monoids.

😵 ~ GHC 8.2 : Maybe **requires Monoid a** as a constraint!

```haskell
instance Monoid a     ⇒ Monoid (Maybe a)
```

Practical Remark: 2

```haskell
newtype Option a = Option (Maybe a)
instance Semigroup a ⇒ Monoid (Option a)
```

`Option` adjoins unit, turning semigroups into monoids.

- 😵 ~ GHC 8.2 : Maybe **requires Monoid a** as a constraint!

```haskell
instance Monoid a     ⇒ Monoid (Maybe a)
```

- 😁 GHC 8.4~ : Requires only `Semigroup a`, **no need of Option**!

```haskell
instance Semigroup a ⇒ Monoid (Maybe a)
```

Practical Remark: 2

```
newtype Option a = Option (Maybe a)
instance Semigroup a ⇒ Monoid (Option a)
```

`Option` adjoins unit, turning semigroups into monoids.

- 😵 ~ GHC 8.2 : Maybe **requires Monoid a** as a constraint!

```
instance Monoid a      ⇒ Monoid (Maybe a)
```

- 😁 GHC 8.4~ : Requires only `Semigroup a`, **`no need of`** **Option**!

```
instance Semigroup a ⇒ Monoid (Maybe a)
```

- 😓 We still have to use Option to write a portable codes though…

Foldable / newtype version

```
import Control.Arrow
aggregate :: [ℕ] → (Maybe ℕ, ℕ)
aggregate =
    fmap getMax *** getSum
  ∘ foldMap (Just . Max &&& Sum)
```

Foldable / newtype version

```
import Control.Arrow
aggregate :: [ℕ] → (Maybe ℕ, ℕ)
aggregate =
    fmap getMax *** getSum
  ∘ foldMap (Just . Max &&& Sum)
```

😁 So concise!

Foldable / newtype version

```
import Control.Arrow
aggregate :: [ℕ] → (Maybe ℕ, ℕ)
aggregate =
    fmap getMax *** getSum
  ∘ foldMap (Just . Max &&& Sum)
```

- 😁 So concise!
- 🤔 It's still tedious to unwrap Max and Sum... they sits in nested types...

Foldable / newtype version

```
import Control.Arrow
aggregate :: [ℕ] → (Maybe ℕ, ℕ)
aggregate =
    fmap getMax *** getSum
  ∘ foldMap (Just . Max &&& Sum)
```

- 😁 So concise!
- 🤔 It's still tedious to unwrap Max and Sum... they sits in nested types...

➡ **Zero-Cost Coercions!**

Safe Zero-Cost Coercions and Roles[1]

Safe Zero-Cost Coercions and Roles[1]

A great invention
opening up the new era of **newtypes**

[1] Breitner, Eisenberg, Peyton Jones and Weirich, 2014

Before 2014

Haskellers' complaint:

Indeed, newtypes are convinent for impl. selection...

But we have to **unwrap them one by one...**

Doing so is not so efficient…

Since we know it's safe,

we can use

unsafeCoerce ...

It's not quite smart…

But

In 2014,
The Revolution took place
to newtypes.

Zero-Cost Coercion

Zero-Cost Coercion

```
import Data.Coerce (coerce)
coerce :: Coercible a b ⇒ a → b
```

Zero-Cost Coercion

```
import Data.Coerce (coerce)
coerce :: Coercible a b ⇒ a → b
```

- **Coercible** relates two types with the same memory repr.

Zero-Cost Coercion

```haskell
import Data.Coerce (coerce)
coerce :: Coercible a b ⇒ a → b
```

- **Coercible** relates two types with the same memory repr.
  - It seems like a type-class, but **GHC generates an information at compile-time**, and user **cannot add custom instance**

Zero-Cost Coercion

```
import Data.Coerce (coerce)
coerce :: Coercible a b ⇒ a → b
```

- **Coercible** relates two types with the same memory repr.
  - It seems like a type-class, but **GHC generates an information at compile-time**, and user **cannot add custom instance**
- With `coerce` from `Data.Coerce`, we can do **zero-cost casts!**

Zero-Cost Coercion

```haskell
import Data.Coerce (coerce)
coerce :: Coercible a b ⇒ a → b
```

- **Coercible** relates two types with the same memory repr.
  - It seems like a type-class, but **GHC generates an information at compile-time**, and user **cannot add custom instance**
- With `coerce` from `Data.Coerce`, we can do **zero-cost casts!**
  - Inferred **per-module**, we need the **info of data constructor** to call `coerce`.

With coerce…

```
import Control.Arrow

aggregate :: [ℕ] → (Maybe ℕ, ℕ)
aggregate =
    fmap getMax *** getSum
  ∘ foldMap (Just . Max &&& Sum)
```

We get it!

```
import Control.Arrow
import Data.Coerce
aggregate :: [ℕ] → (Maybe ℕ, ℕ)
aggregate =
  coerce ∘ foldMap (Just . Max &&& Sum)
```

We get it!

```haskell
import Control.Arrow
import Data.Coerce
aggregate :: [ℕ] → (Maybe ℕ, ℕ)
aggregate =
  coerce ∘ foldMap (Just . Max &&&
```

Casting nested types with zero-cost!

We get it!

```
import Control.Arrow
import Data.Coerce
aggregate :: [ℕ] → (Maybe ℕ, ℕ)
aggregate =
  coerce ∘ foldMap (Just . Max &&&
```

Casting nested types with zero-cost!

```
coerce :: (Maybe (Max ℕ), Sum ℕ) → (Maybe ℕ, ℕ)
```

We get it!

```
import Control.Arrow
import Data.Coerce
aggregate :: [ℕ] → (Maybe ℕ, ℕ)
aggregate =
  coerce ∘ foldMap (Just . Max &&&
```

Casting nested types with zero-cost!

```
coerce :: (Maybe (Max N), Sum N) → (Maybe N, N)
```

- No effect on the # of scanning since it's zero-cost

We get it!

```haskell
import Control.Arrow
import Data.Coerce
aggregate :: [ℕ] → (Maybe ℕ, ℕ)
aggregate =
  coerce ∘ foldMap (Just . Max &&&
```

Casting nested types with zero-cost!

```haskell
coerce :: (Maybe (Max N), Sum N) → (Maybe N, N)
```

- No effect on the # of scanning since it's zero-cost
- Just one call for coerce to make it done!

Casting b/w nested types

Casting b/w nested types

- 😁 It's convenient that we can cast **any nested types**!

Casting b/w nested types

- 😁 It's convenient that we can cast **<u>any nested types</u>**!
- 🧐 ... Really?

Casting b/w nested types

- 😁 It's convenient that we can cast **<u>any nested types</u>**!
- 🧐 … Really?

```
newtype Down a = Down a
instance Ord a ⇒ Ord (Down a) where

  a ≤ b = b ≤ a
data Heap a
minView :: Heap a → Maybe a
```

Casting b/w nested types

- 😁 It's convenient that we can cast **<u>any nested types</u>**!
- 🧐 ... Really?

```haskell
newtype Down a = Down a
instance Ord a ⇒ Ord (Down a) where
  a ≤ b = b ≤ a
data Heap a
minView :: Heap a → Maybe a
```

Reversed Order

Casting b/w nested types

- 😁 It's convenient that we can cast **<u>any nested types</u>**!
- 🧐 … Really?

```haskell
newtype Down a = Down a
instance Ord a ⇒ Ord (Down a) where
  a ≤ b = b ≤ a
data Heap a
minView :: Heap a → Maybe a
```

Reversed Order

Heap

Casting b/w nested types

- 😁 It's convenient that we can cast **<u>any nested types</u>**!
- 🧐 … Really?

```haskell
newtype Down a = Down a
instance Ord a ⇒ Ord (Down a) where
  a ≤ b = b ≤ a
data Heap a
minView :: Heap a → Maybe a
```

Reversed Order

Heap

Minimum, O(1)

Casting b/w nested types

- 😁 It's convenient that we can cast **<u>any nested types</u>**!
- 🧐 … Really?

```haskell
newtype Down a = Down a
instance Ord a ⇒ Ord (Down a) where
  a ≤ b = b ≤ a
data Heap a
minView :: Heap a → Maybe a
```

Reversed Order

Heap

Minimum, O(1)

Semantically, `Heap a` MUST NOT be casted to `Heap (Down a)`!

Roles!

Roles!

- We can cast them with coerce!

Roles!

- We can cast them with coerce!

```
ghci> h = fromList [1,2,3] :: Heap Int
ghci> minView (coerce h :: Heap (Down Int))
Just 1
```

Roles!

- We can cast them with coerce!

```
ghci> h = fromList [1,2,3] :: Heap Int
ghci> minView (coerce h :: Heap (Down Int))
Just 1
```

Must be Just 3!

Roles!

- We can cast them with coerce!

```
ghci> h = fromList [1,2,3] :: Heap Int
ghci> minView (coerce h :: Heap (Down Int))
Just 1
```

Must be `Just 3`!

- Then we specify the **Role**:

Roles!

- We can cast them with coerce!

```
ghci> h = fromList [1,2,3] :: Heap Int
ghci> minView (coerce h :: Heap (Down Int))
Just 1
```

Must be `Just 3`!

- Then we specify the **Role**:

```
type role Heap nominal
```

Roles!

- We can cast them with coerce!

```
ghci> h = fromList [1,2,3] :: Heap Int
ghci> minView (coerce h :: Heap (Down Int))
Just 1
```

Must be Just 3!

- Then we specify the **Role**:

```
type role Heap nominal
```

```
ghci> minView (coerce h :: Heap (Down Int))
error: Couldn't match type ‘Int’ with ‘Down Int’
```

Roles!

- We can cast them with coerce!

```
ghci> h = fromList [1,2,3] :: Heap Int
ghci> minView (coerce h :: Heap (Down Int))
Just 1
```

Must be Just 3!

- Then we specify the **Role**:

```
type role Heap nominal
```

```
ghci> minView (coerce h :: Heap (Down Int))
error: Couldn't match type 'Int' with 'Down Int'
```

- We cannot coerce without the info of newtype constructors

More on Roles

More on Roles

- **Role**: The type variable here behaves like this…

More on Roles

- **Role**: The type variable here behaves like this…
- Three kinds: **representational** / **nominal** / **phantom**

More on Roles

- <u>**Role**</u>: The type variable here behaves like this…
- Three kinds: **representational / nominal / phantom**
    - **repr**: equivalent if they have the same representation.

More on Roles

- **Role**: The type variable here behaves like this…
- Three kinds: **representational / nominal / phantom**
  - **repr**: equivalent if they have the same representation.
  - **nominal**: must have exactly the same type!

More on Roles

- Role: The type variable here behaves like this...
- Three kinds: **representational / nominal / phantom**
  - **repr**: equivalent if they have the same representation.
  - **nominal**: must have exactly the same type!
  - **phantom**: unrelated to its real content! anything goes!

More on Roles

- **Role**: The type variable here behaves like this...
- Three kinds: **representational** / **nominal** / **phantom**
  - **repr**: equivalent if they have the same representation.
  - **nominal**: must have exactly the same type!
  - **phantom**: unrelated to its real content! anything goes!
- GHC infers most general roles at every time.

More on Roles

- **Role**: The type variable here behaves like this...
- Three kinds: **representational / nominal / phantom**
  - **repr**: equivalent if they have the same representation.
  - **nominal**: must have exactly the same type!
  - **phantom**: unrelated to its real content! anything goes!
- GHC infers most general roles at every time.
  - Sometimes library implementor must specify roles, because GHC can't tell the semantics specific to the particular type

More on Roles

- **Role**: The type variable here behaves like this...
- Three kinds: **representational / nominal / phantom**
  - **repr**: equivalent if they have the same representation.
  - **nominal**: must have exactly the same type!
  - **phantom**: unrelated to its real content! anything goes!
- GHC infers most general roles at every time.
  - Sometimes library implementor must specify roles, because GHC can't tell the semantics specific to the particular type
  - We can't coerce types withou newtype constructor info.

Coercion & Roles Summary

- With **coerce function**, we can cast nest types with the same representation, with **zero-cost**!
  - We can use newtypes more safely and conveniently!
- We can control castability by specifying **Roles**.
  - Roles are usually inferred.
  - We have to specify roles when we want to disallow casts for the semantical reasons.

Yes, That's what we wanted!

Why we didn't have this?

Pre history of Zero-Cost Coercion: GND crisis

- We have Generalized Newtype Deriving (GND) at keast already in GHC 5.
  - At that time, GHC had only a "tame" type-system, everything was fine.
- Later, type families, GADTs and so on came into the GHC's type system and ...
  - **GND became unound**!

GND was unsound

```
newtype Id1 a = MkId1 a
newtype Id2 a = MkId2 (Id1 a)
                 deriving (UnsafeCast b)
type family Discern a b
type instance Discern (Id1 a) b = a
type instance Discern (Id2 a) b = b
class UnsafeCast to from where
  unsafe :: from → Discern from to
instance UnsafeCast b (Id1 a) where
  unsafe (MkId1 x) = x
unsafeCoerce :: a → b
unsafeCoerce x = unsafe (MkId2 (MkId1 x))
```

GND was unsound

```
newtype Id1 a = MkId1 a
newtype Id2 a = MkId2 (Id1 a)
                deriving (UnsafeCast b)
type family Discern a b
type instance Discern (Id1 a) b = a
type instance Discern (Id2 a) b = b
class UnsafeCast to from where
  unsafe :: from → Discern from to
instance UnsafeCast b (Id1 a) where
  unsafe (MkId1 x) = x
unsafeCoerce :: a → b
unsafeCoerce x = unsafe (MkId2 (MkId1 x))
```

GND!

GND was unsound

```
newtype Id1 a = MkId1 a
newtype Id2 a = MkId2 (Id1 a)
                deriving (UnsafeCast b)
type family Discern a b
type instance Discern (Id1 a) b = a
type instance Discern (Id2 a) b = b
class UnsafeCast to from where
  unsafe :: from → Discern from to
instance UnsafeCast b (Id1 a) where
  unsafe (MkId1 x) = x
unsafeCoerce :: a → b
unsafeCoerce x = unsafe (MkId2 (MkId1 x))
```

GND!

Cast b/w any types!

That's terrible...

We have to save the GND…

That's why Roles
are emerged.

GND is Unsound

```haskell
newtype Id1 a = MkId1 a
newtype Id2 a = MkId2 (Id1 a)
                deriving (UnsafeCast b)
type family Discern a b
type instance Discern (Id1 a) b = a
type instance Discern (Id2 a) b = b
class UnsafeCast to from where
  unsafe :: from → Discern from to
instance UnsafeCast b (Id1 a) where
  unsafe (MkId1 x) = x
unsafeCoerce :: a → b
unsafeCoerce x = unsafe (MkId2 (MkId1 x))
```

GND is Unsound

```haskell
newtype Id1 a = MkId1 a
newtype Id2 a = MkId2 (Id1 a)
                    deriving (UnsafeCast b)
type family Discern a b
type instance Discern (Id1 a) b = a
type instance Discern (Id2 a) b = b
class UnsafeCast to from where
  unsafe :: from → Discern from to
instance UnsafeCast b (Id1 a) where
  unsafe (MkId1 x) = x
unsafeCoerce :: a → b
unsafeCoerce x = unsafe (MkId2 (MkId1 x))
```

GND became to use coerce

GND is Unsound

```
newtype Id1 a = MkId1 a
newtype Id2 a = MkId2 (Id1 a)
                deriving (UnsafeCast b)
type family Discern a b
type instance Dis                  = a
type instance Dis                  = b
class UnsafeCast
  unsafe :: from                  to
instance UnsafeCast b (Id1 a) where
  unsafe (MkId1 x) = x
unsafeCoerce :: a → b
unsafeCoerce x = unsafe (MkId2 (MkId1 x))
```

GND became to use coerce

GHC is clever enough to infer a as nominal

GND is Unsound

```
newtype Id1 a = MkId1 a
newtype Id2 a = MkId2 (Id1 a)
                deriving (UnsafeCast b)
type family Discern a b
type instance Dis                    = a
type instance Dis                    = b
class UnsafeCast
  unsafe :: from                     to
instance UnsafeCast b (Id1 a) where
  unsafe (MkId1 x) = x
unsafeCoerce :: a → b
unsafeCoerce x = unsafe (MkId2 (MkId1 x))
```

GHC is clever enough to infer a as nominal

GND became to use coerce

Reject!

☺️

Now, **newtypes** can play their roles thanks to Roles

Roles of newtypes:

Implementation Hiding

Implementation Sharing

Implementation Selection.

This is where
newtypes
stands now.

From now on:
The future of
newtype

The future of **newtypes**,

or: Deriving Via^2

The future of **newtypes**, or: Deriving Via[2]

~When the impl. sharing and selection meet~

[2] Blöndal, Löh and Scott, 2018

Deriving Via: More Flexible Implementation Sharing

Deriving Via: More Flexible Implementation Sharing

- A new feature of GHC 8.6.

Deriving Via: More Flexible Implementation Sharing

- A new feature of GHC 8.6.
    - GHC 8.6.1-alpha2 is released at the time of this talk

Deriving Via: More Flexible Implementation Sharing

- A new feature of GHC 8.6.
  - GHC 8.6.1-alpha2 is released at the time of this talk
- We can use newtypes as a **hint** for deriving clauses.

Eg: Monoid structure of Id

```
{-# LANGUAGE DerivingVia #-}
newtype Id = MkId Word
  deriving (Semigroup, Monoid) via Max Word
```

Eg: Monoid structure of Id

```haskell
{-# LANGUAGE DerivingVia #-}
newtype Id = MkId Word
  deriving (Semigroup, Monoid) via Max Word
```

- As we have seen, <u>Word have multiple monoid impls.</u>

Eg: Monoid structure of Id

```haskell
{-# LANGUAGE DerivingVia #-}
newtype Id = MkId Word
  deriving (Semigroup, Monoid) via Max Word
```

- As we have seen, <u>Word have multiple monoid impls.</u>
- Suppose we want to use "choosing the newest Id (= Maximum Id)" as monoid operation on Ids.

Eg: Monoid structure of Id

```haskell
{-# LANGUAGE DerivingVia #-}
newtype Id = MkId Word
  deriving (Semigroup, Monoid) via Max Word
```

- As we have seen, Word have multiple monoid impls.
- Suppose we want to use "choosing the newest Id (= Maximum Id)" as monoid operation on Ids.
  - `Max Word` has the same repr as `Id`, forms a monoid with respect to (max, 0).

Eg: Monoid structure of Id

```
{-# LANGUAGE DerivingVia #-}
newtype Id = MkId Word
  deriving (Semigroup, Monoid) via Max Word
```

- As we have seen, Word have multiple monoid impls.
- Suppose we want to use "choosing the newest Id (= Maximum Id)" as monoid operation on Ids.
  - `Max Word` has the same repr as `Id`, forms a monoid with respect to (max, 0).
  - DerivingVia can **lift this impl automatically to Id!**

DerivingVia vs GND

DerivingVia vs GND

- DerivingVia is **a superset of GND**

DerivingVia vs GND

- DerivingVia is **a superset of GND**
  - **GND** just looks at **the innermost type**.

DerivingVia vs GND

- DerivingVia is <u>**a superset of GND**</u>
    - **GND** just looks at **the innermost type**.
    - **DerivingVia** let us reuse the impl. of <u>**any Coercible types**</u>, without any cost!

DerivingVia is not limited to **newtypes**

DerivingVia is not limited to **newtypes**

- In the original paper, it is proposed to use DerivingVia to share implementations **between any isomophic types**.

※This "isomorphic" means slightly stronger condition;
it means "their generic representation is the same".

DerivingVia is not limited to **newtypes**

- In the original paper, it is proposed to use DerivingVia to share implementations **between any isomophic types**.

🔑 Combination of **Generics** and Coercion.

※This "isomorphic" means slightly stronger condition;
it means "their generic representation is the same".

Demo

Static definition of JSON de/serialization instance

Impl sharing b/w isomorphic types

※Complete code is available at `http://bit.ly/derivia`

Complicated Example

```haskell
data OtherConfig = OtherConfig
                   { otrNameOfProcess :: Maybe String
                   , otrArgsToProcess :: [String]
                   }
  deriving (Read, Show, Eq, Ord, Generic)
  deriving (ToJSON, FromJSON)
       via WithOptions '[ FieldLabelModifier      '[CamelTo2 "-"]
                        , ConstructorTagModifier '[CamelTo2 "-"]
                        , OmitNothingFields      'True
                        ]
                        OtherConfig
```

- Specifies the encoding method at type-level
- Statically assures that same encoding is usedin FromJSON & ToJSON

Example of Iso

```haskell
data Blog = Blog { authors :: [Author]
                 , articles :: [Article] }
  deriving (Generic)
  deriving (Semigroup, Monoid)
       via Blog `SameRepAs` ([Author], Dual [Article])

{-
ghci>  mconcat [Blog ["1"] ["1"], Blog ["2"] ["3","4"]]
Blog {authors = ["1","2"], articles = ["3","4","1"]}
-}
```

Example of Iso (cont.)

```
newtype SameRepAs a b = SameRepAs { runSameRepAs :: a }

type Iso a b = (Generic a, Generic b,
                Coercible (Rep a ()) (Rep b ()))

instance (Semigroup b, Iso a b)
      ⇒  Semigroup (SameRepAs a b) where
  SameRepAs a <> SameRepAs b = ...

instance (Monoid b, Iso a b)
       ⇒ Monoid (SameRepAs a b) where
  mempty = SameRepAs $ toA mempty
    where
      toA :: b -> a
      toA = to . (coerce :: Rep b () -> Rep a ()) . from
```

DerivingVia: Summary

DerivingVia: Summary

- Available since GHC 8.6.

DerivingVia: Summary

- Available since GHC 8.6.
- We can use **newtype**s to specify the impl. for deriving clauses.

DerivingVia: Summary

- Available since GHC 8.6.
- We can use **newtype**s to specify the impl. for deriving clauses.
- Combined with Generics, we can even derive the instance from isomorphic type, but <u>**not necessarily representationally equal**</u>!

DerivingVia: Summary

- Available since GHC 8.6.
- We can use **newtype**s to specify the impl. for deriving clauses.
- Combined with Generics, we can even derive the instance from isomorphic type, but <u>**not necessarily representationally equa**</u>l!
    - Any other "isomorphism" expressible as a type constraint is also applicable to this technique.

Summary

Summary

Summary

- Three roles of newtypes:

Summary

- Three roles of newtypes:
  - Implementation Hiding / Sharing/ Selection.

Summary

- Three roles of newtypes:
  - Implementation Hiding / Sharing/ Selection.
- With Data.Coerce, we can **cast representationally equall types with zero-cost!**

Summary

- Three roles of newtypes:
  - Implementation Hiding / Sharing/ Selection.
- With Data.Coerce, we can **cast representationally equall types with zero-cost!**
  - The newtype Revolution started here

Summary

- Three roles of newtypes:
  - Implementation Hiding / Sharing/ Selection.
- With Data.Coerce, we can **cast representationally equall types with zero-cost!**
  - The newtype Revolution started here
  - We can treat compound types properly with **role inference and annotations**.

Summary

- Three roles of newtypes:
  - Implementation Hiding / Sharing/ Selection.
- With Data.Coerce, we can **cast representationally equall types with zero-cost!**
  - The newtype Revolution started here
  - We can treat compound types properly with **role inference and annotations**.
- Since GHC 8.6, **DerivingVia** enables us to use **newtype to customise the deriving clauses!**

References

1. J. Breitner, R. A. Eisenberg, S. P. Jones and S. Weirich, *Safe Zero-cost Coercions for Haskell*, ICFP 2014.

2. Baldur Blöndal, Andres Löh and Ryan Scott, *Deriving Via: How to Turn Hand-Written Instances into an Anti-Pattern*, ICFP18.