

# 本当はすごい newtype

@mr\_konn

<https://konn-san.com>

Slides are available at: <http://bit.ly/derivia>

Example codes are on GitHub: [konn/newtype-talk-five](#)

# 自己紹介

- ・ いし い ひろ み 石井 大海 (@mr\_konn)
- ・ 数学専攻博士課程3年
  - ・ 数理論理学と計算機科学とか
- ・ 気づけば Haskell 歴12年

すごい！



# 本当はすごい newtype

Roles, Safe zero-cost coercions, and DerivingVia  
～Monoid & Foldable もあるよ～

**newtype**

# newtype

```
newtype Foo  $\alpha$  = Bar  $\alpha$   
newtype Id = MkId Word
```

# newtype

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

- ・ ただ一つの構築子とフィールドを持つデータ型

# newtype

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

- ・ ただ一つの構築子とフィールドを持つデータ型
- ・ 内部表現が唯一のフィールドの型と同一

# newtype

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

- ・ ただ一つの構築子とフィールドを持つデータ型
- ・ 内部表現が唯一のフィールドの型と同一
- ・ 型の上では区別されるが、メモリ上での表現は同一で、フィールドも「正格」に評価される



初心者あるある

# 初心者あるある

🤔 「ふつうの **data** 宣言と何が違うの？」

# 初心者あるある

- 🤔 「ふつうの **data** 宣言と何が違うの？」
- 🧐 「内部表現が同じだから効率的だよ」

# 初心者あるある

- 🤔 「ふつうの **data** 宣言と何が違うの？」
- 🤔 「内部表現が同じだから効率的だよ」
- 🤔 「効率まだどうでもいいし **data** でいいや」

# 初心者あるある

- 🤔 「ふつうの **data** 宣言と何が違うの？」
- 🧐 「内部表現が同じだから効率的だよ」
- 🤔 「効率まだどうでもいいし **data** でいいや」
- 😌 「今は `unpack strict field` もあるし……」

そんなことはない !!!

# `newtype` の3つの用途

# newtype の3つの用途

## 実装の隠蔽

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



# newtype の3つの用途

## 🌀 実装の隠蔽

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

型の上では区別  
でも内部的には Word

# newtype の3つの用途

## 🌀 実装の隠蔽

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

外部にはデータ構築子  
MkId は隠蔽

型の上では区別  
でも内部的には Word

# newtype の3つの用途

## 実装の隠蔽

外部にはデータ構築子  
MkId は隠蔽

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

型の上では区別  
でも内部的には Word

## 実装の共有

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

# newtype の3つの用途

## 🌀 実装の隠蔽

外部にはデータ構築子  
MkId は隠蔽

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

型の上では区別  
でも内部的には Word

## 🌀 実装の共有

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

本来は導出不可でも  
Word の実装が共有できる

# newtype の3つの用途

## 実装の隠蔽

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

外部にはデータ構築子  
MkId は隠蔽

型の上では区別  
でも内部的には Word

## 実装の共有

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

本来は導出不可でも  
Word の実装が共有できる

DerivingVia  
で更に進化

# newtype の3つの用途

## 実装の隠蔽

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

外部にはデータ構築子  
MkId は隠蔽

型の上では区別  
でも内部的には Word

## 実装の共有

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

## 実装の選択

本来は導出不可でも  
Word の実装が共有できる

DerivingVia  
で更に進化

# newtype の3つの用途

## 実装の隠蔽

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

外部にはデータ構築子  
MkId は隠蔽

型の上では区別  
でも内部的には Word

## 実装の共有

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

## 実装の選択

本来は導出不可でも  
Word の実装が共有できる

DerivingVia  
で更に進化

# 実装の選択



# 実装の選択

Monoid と Foldable を例に

# 例題：リストの走査

問. 整数のリストが与えられた時、その最大値と総和を一回の走査で求めよ。

※ foldl および folds パッケージはみなかった事にしてください

# 解答例

# 解答例

😊 畳み込みだ！

# 解答例

😄 畳み込みだ！

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

※ N は Integer の略だと思ってください

# 解答例

😄 畳み込みだ！

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

※ N は Integer の略だと思ってください

🤔 何か似たようなことやってんな……

# 解答例

😄 畳み込みだ！

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

※ N は Integer の略だと思ってください

🤔 何か似たようなことやってんな……

# 解答例

😄 畳み込みだ！

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

変換＋二項演算

※ N は Integer の略だと思ってください

🤔 何か似たようなことやってんな……



# 解答例

😄 畳み込みだ！

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

変換＋二項演算

※ N は Integer の略だと思ってください

🤔 何か似たようなことやってんな……

# 解答例

😊 畳み込みだ！

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

単位元

変換＋二項演算

※ N は Integer の略だと思ってください

🤔 何か似たようなことやってんな……

# 解答例

😊 畳み込みだ！

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

単位元

変換＋二項演算

※ N は Integer の略だと思ってください

🤔 何か似たようなことやってんな……

😂 Monoid だ！

# Monoid

- 左右の別なく計算できて、単位元があるやつ

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

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

- max も + も.....
  - 左からやっても右からやっても同じ
  - Nothing（最大値無し）と 0 が単位元（ $\varepsilon$ ）
- モノイド変換 + 畳み込み  $\leadsto$  Foldable!

# Foldable 型クラス

```
class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m
  ...
```

# Foldable 型クラス

モノイドに変換  
+  
左から畳み込み

```
class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m
  ...
```

# Foldable 型クラス

モノイドに変換  
+  
左から畳み込み

```
class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m
  ...
```

😊 この枠組みに max, (+) を落とせばよさそう

# Foldable 型クラス

モノイドに変換  
+  
左から畳み込み

```
class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m
  ...
```

😊 この枠組みに max, (+) を落とせばよさそう

🤔 Monoid Word インスタンスは高々一つしか書けない……



# Foldable 型クラス

モノイドに変換  
+  
左から畳み込み

```
class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m
  ...
```

- 😊 この枠組みに `max`, `(+)` を落とせばよさそう
- 🤔 `Monoid Word` インスタンスは高々一つしか書けない……
- 😊 そこで `newtype` による実装の選択！

# インスタンス例

```
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
```

# インスタンス例

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

数の加法に関する  
モノイド

```
instance Num a  $\Rightarrow$  Monoid (Sum a) where
```

```
    (<>) = (+);  $\epsilon$  = 0
```

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

```
instance Ord a  $\Rightarrow$  Semigroup (Max a) where
```

```
    (<>) = max
```

# インスタンス例

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

数の加法に関する  
モノイド

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

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

最大限に関する  
半群

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

# インスタンス例

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

数の加法に関する  
モノイド

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

GND による  
実装の共有

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

最大限に関する  
半群

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

# 実際上の注意 1

```
newtype Max a = Max a
instance Ord a  $\Rightarrow$  Semigroup (Max a)
instance Bounded a  $\Rightarrow$  Monoid (Max a)
```

- ・ 有界な型しかモノイドにならない！（最大小元がないと単位元にならないので）
- ・ 単位元のない半群にはなる
  - ・ 半群からモノイドに変換しないとFoldableは使えない！

# 実際上の注意 2

```
newtype Option a = Option (Maybe a)  
instance Semigroup a  $\Rightarrow$  Monoid (Option a)
```

# 実際上の注意 2

```
newtype Option a = Option (Maybe a)  
instance Semigroup a  $\Rightarrow$  Monoid (Option a)
```

Optionは 半群に単位元を付け足してモノイドにする



# 実際上の注意 2

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

Optionは 半群に単位元を付け足してモノイドにする

😓 ～ GHC 8.2 : Maybe は制約として Monoid a を要求 !

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

# 実際上の注意 2

```
newtype Option a = Option (Maybe a)  
instance Semigroup a  $\Rightarrow$  Monoid (Option a)
```

Optionは 半群に単位元を付け足してモノイドにする

😵 ～ GHC 8.2 : Maybe は制約として Monoid a を要求 !

```
instance Monoid a  $\Rightarrow$  Monoid (Maybe a)
```

😁 GHC 8.4～ : Semigroup a になり、Option は不要に !

```
instance Semigroup a  $\Rightarrow$  Monoid (Maybe a)
```

# 実際上の注意 2

```
newtype Option a = Option (Maybe a)  
instance Semigroup a  $\Rightarrow$  Monoid (Option a)
```

Optionは 半群に単位元を付け足してモノイドにする

😓 ～ GHC 8.2 : Maybe は制約として Monoid a を要求 !

```
instance Monoid a  $\Rightarrow$  Monoid (Maybe a)
```

😄 GHC 8.4～ : Semigroup a になり、Option は不要に !

```
instance Semigroup a  $\Rightarrow$  Monoid (Maybe a)
```

😓 とはいえ、ポータブルなコードを書こうと思ったら Option に頼る必要あり……。

# Foldable / Newtype 版

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

# Foldable / Newtype 版

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

😊 随分すっきり！

# Foldable / Newtype 版

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

😊 随分すっきり！

🤔 でも一々 Max, Sum を剥すのはダルいなあ……ネストしてるし……

# Foldable / Newtype 版

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

😊 随分すっきり！

😞 でも一々 Max, Sum を剥すのはダルいなあ……ネストしてるし……

➔ Zero-Cost Coercions!

# Safe Zero-Cost Coercions and Roles<sup>1</sup>



# Safe Zero-Cost Coercions and Roles<sup>1</sup>

～**newtype** の未来を切り拓いた大発明～

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

2014年以前の  
Haskeller のぼやき：

たしかに実装の選択に  
newtype は便利だけどさ……

結局一枚一枚剥して  
いかなないといけなないし

そうすると効率は  
悪いじゃん

安全だから unsafeCoerce

使う手もあるけど

あんま綺麗じゃない……

しかし



2014年、  
newtype に  
革命が起きた

# 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** は内部表現が同じ型同士を関連づける型制約

# Zero-Cost Coercion

```
import Data.Coerce (coerce)
coerce :: Coercible a b => a -> b
```

- **Coercible** は内部表現が同じ型同士を関連づける型制約
  - 一見型クラスだが、実際には GHC がコンパイル時に情報を生成し、勝手にインスタンスを追加できない

# Zero-Cost Coercion

```
import Data.Coerce (coerce)
coerce :: Coercible a b => a -> b
```

- **Coercible** は内部表現が同じ型同士を関連づける型制約
  - 一見型クラスだが、実際には GHC が コンパイル時に情報を生成し、勝手にインスタンスを追加できない
- Data.Coerce の coerce 関数で ゼロコストのキャストができる！

# Zero-Cost Coercion

```
import Data.Coerce (coerce)
coerce :: Coercible a b => a -> b
```

- **Coercible** は内部表現が同じ型同士を関連づける型制約
  - 一見型クラスだが、実際には GHC が コンパイル時に情報を生成し、勝手にインスタンスを追加できない
- Data.Coerce の coerce 関数で ゼロコストのキャストができる！
  - モジュール毎に推論、coerce には データ構築子の情報が必要

# これを使えば.....

```
import Control.Arrow
```

```
aggregate :: [N] → (Maybe N, N)
```

```
aggregate =
```

```
    fmap getMax *** getSum
```

```
○ foldMap (Just . Max &&& Sum)
```



# こうなります

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

# こうなります

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

ネスト型も  
ゼロコストで変換！

# こうなります

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

ネスト型も  
ゼロコストで変換!

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

# こうなります

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

ネスト型も  
ゼロコストで変換!

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

- ・ ゼロコストなので、走査は一回だけ

# こうなります

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

ネスト型も  
ゼロコストで変換!

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

- ・ ゼロコストなので、走査は一回だけ
- ・ 前後の型がしっかり決まっているので coerce 一発で終了

# ネスト型の変換

# ネスト型の変換

😊 どんなコンテナ中の型も 変換できて便利！

# ネスト型の変換

😊 どんなコンテナ中の型も 変換できて便利！

🙄 本当に？



# ネスト型の変換

😊 どんなコンテナ中の型も 変換できて便利！

🤔 本当に？

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

# ネスト型の変換

😊 どんなコンテナ中の型も 変換できて便利！

🤔 本当に？

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

逆順を入れる

# ネスト型の変換

😊 どんなコンテナ中の型も 変換できて便利！

🤔 本当に？

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

逆順を入れる

ヒープ

# ネスト型の変換

😊 どんなコンテナ中の型も 変換できて便利！

🤔 本当に？

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

逆順を入れる

ヒープ

最小元 O(1)

# ネスト型の変換

😊 どんなコンテナ中の型も 変換できて便利！

🤔 本当に？

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

逆順を入れる

ヒープ

最小元 O(1)

意味論的に `Heap a` と `Heap (Down a)` はキャスト出来ないべき

そこで Roles!

# そこで Roles!

- ・ このままでは coerce で変換出来てしまう！

# そこで Roles!

- ・ このままでは coerce で変換出来てしまう！

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



# そこで Roles!

- このままでは coerce で変換出来てしまう！

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

Just 3 であるべき！

# そこで Roles!

- このままでは coerce で変換出来てしまう！

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

Just 3 であるべき！

- そこで Role を指定：

# そこで Roles!

- このままでは coerce で変換出来てしまう！

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

Just 3 であるべき！

- そこで Role を指定：

```
type role Heap nominal
```

# そこで Roles!

- このままでは coerce で変換出来てしまう！

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

Just 3 であるべき！

- そこで Role を指定：

```
type role Heap nominal
```

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

# そこで Roles!

- このままでは coerce で変換出来てしまう！

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

Just 3 であるべき！

- そこで Role を指定：

```
type role Heap nominal
```

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

- データ構築子を露出しなければ、他のモジュールでは nominal に推論される筈

# そこで Roles!

- このままでは coerce で変換出来てしまう！

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

Just 3 であるべき！

- そこで Role を指定：

```
type role Heap nominal
```

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

- データ構築子を露出しなければ、他のモジュールでは nominal に推論される筈

- newtype 構築子の情報を import しないと coerce できない

# Roles 詳細

# Roles 詳細

- ・ Role: 「この型変数はこういう奴です」



# Roles 詳細

- ・ Role: 「この型変数はこういう奴です」
- ・ 三種: **representational / nominal / phantom**

# Roles 詳細

- ・ **Role**: 「この型変数はこういう奴です」
- ・ 三種: **representational / nominal / phantom**
  - ・ **repr**: 「ここは内部表現が同じなら同値」

# Roles 詳細

- ・ **Role**: 「この型変数はこういう奴です」
- ・ 三種: **representational / nominal / phantom**
  - ・ **repr**: 「ここは内部表現が同じなら同値」
  - ・ **nominal**: 「完全に同じ型じゃないとだめ」

# Roles 詳細

- ・ **Role**: 「この型変数はこういう奴です」
- ・ 三種: **representational / nominal / phantom**
  - ・ **repr**: 「ここは内部表現が同じなら同値」
  - ・ **nominal**: 「完全に同じ型じゃないとだめ」
  - ・ **phantom**: 「タダの飾り！中身に関係なし」

# Roles 詳細

- ・ **Role**: 「この型変数はこういう奴です」
- ・ 三種: **representational** / **nominal** / **phantom**
  - ・ **repr**: 「ここは内部表現が同じなら同値」
  - ・ **nominal**: 「完全に同じ型じゃないとだめ」
  - ・ **phantom**: 「タダの飾り！中身に関係なし」
- ・ GHC は適宜な role を推論してくれる

# Roles 詳細

- ・ **Role**: 「この型変数はこういう奴です」
- ・ 三種: **representational** / **nominal** / **phantom**
  - ・ **repr**: 「ここは内部表現が同じなら同値」
  - ・ **nominal**: 「完全に同じ型じゃないとだめ」
  - ・ **phantom**: 「タダの飾り！中身に関係なし」
- ・ GHC は適宜な role を推論してくれる
  - ・ データ型固有の意味論はユーザしか知らないなので、今回のようなケースではライブラリ実装者が指定

# Roles 詳細

- ・ **Role**: 「この型変数はこういう奴です」
- ・ 三種: **representational** / **nominal** / **phantom**
  - ・ **repr**: 「ここは内部表現が同じなら同値」
  - ・ **nominal**: 「完全に同じ型じゃないとだめ」
  - ・ **phantom**: 「タダの飾り！中身に関係なし」
- ・ GHC は適宜な role を推論してくれる
  - ・ データ型固有の意味論はユーザしか知らないなので、今回のようなケースではライブラリ実装者が指定
  - ・ newtype 構築子の情報がないと coerce は不可能

# Coercion & Roles まとめ

- ・ coerce 関数を使うと内部表現の同じ型はネストしていてもゼロコストで変換可能
  - ・ newtype の利用がより安全・簡単に！
- ・ 細かな変換可能性は Role を指定して制御
  - ・ 普段は Role は推論されるので問題なし
  - ・ 意味論上変換されると困る／変換出来てほしいものは自分で指定



これこれ、  
こういうの欲しかった

てかなんでなかったの？

# Zero-Cost Coercion 前史： GND の受難

- ・ Generalized Newtype Deriving (GND) はGHC 5 の頃には既にあった
  - ・ この頃は GHC は型システムが控え目で、全ては上手く回っていた
- ・ その後、型族や GADTs などが入り……
  - ・ 気付くと GND は不健全になっていた！

# GND は不健全

```
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 は不健全

```
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 は不健全

```
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!

任意の型間のキャスト！

これはまずい……

GND を救いたい.....



.....という経緯で  
Role が現れた

# GND は不健全

```
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 は不健全

```
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 は coerce  
を経由するように

# GND は不健全

```
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 a b
  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 は coerce  
を経由するように

GHC は賢いので  
a を nominal に推論

# GND は不健全

```
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 a (Id2 b) = b
class UnsafeCast a b
  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 は coerce  
を経由するように

GHC は賢いので  
a を nominal に推論

Reject!



Roles の登場で  
**newtype** は  
役割を果せるように

役割：



実装の隠蔽

実装の共有

# 実装の選択

いま  
これが現在の  
newtype

ここからは  
あした  
未来の newtype  
の話

**newtype** の<sup>あした</sup>未来

または、Deriving Via<sup>2</sup>

# newtype の<sup>あした</sup>未来

## または、Deriving Via<sup>2</sup>

～実装の共有と選択がであうとき～

[2] Blöndal, Löh and Scott, 2018

# Deriving Via: より柔軟な 実装の共有



# Deriving Via: より柔軟な 実装の共有

- ・ GHC 8.6 から入る新機能

# Deriving Via: より柔軟な 実装の共有

- ・ GHC 8.6 から入る新機能
  - ・ 現時点で GHC 8.6.1-alpha2 が出ている

# Deriving Via: より柔軟な 実装の共有

- ・ GHC 8.6 から入る新機能
  - ・ 現時点で GHC 8.6.1-alpha2 が出ている
- ・ newtype を deriving 節のヒントとして用いることができるようになる

# 例：Idのモノイド構造

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

# 例：Idのモノイド構造

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

- ・ 前に見たように、Word のモノイド実装は複数ある

# 例：Idのモノイド構造

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

- ・ 前に見たように、Word のモノイド実装は複数ある
- ・ Id では「一番新しいId (= 最大のId) を選ぶ」演算をモノイド演算として使いたかったとする

# 例：Idのモノイド構造

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

- ・ 前に見たように、Word のモノイド実装は複数ある
- ・ Id では「一番新しいId (= 最大のId) を選ぶ」演算をモノイド演算として使いたかったとする
  - ・ Max Word は Id と同一表現、(max, 0) に関しモノイド

# 例：Idのモノイド構造

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

- ・ 前に見たように、Word のモノイド実装は複数ある
- ・ Id では「一番新しいId (= 最大のId) を選ぶ」演算をモノイド演算として使いたかったとする
  - ・ Max Word は Id と同一表現、(max, 0) に関しモノイド
  - ・ DerivingVia はこの実装を自動的に Id に持ち上げてくれる！



# DerivingVia vs GND

# DerivingVia vs GND

- DerivingVia は GND のスーパーセット

# DerivingVia vs GND

- DerivingVia は GND のスーパーセット
- GND は包まれる一番内側の型の実装を見る

# DerivingVia vs GND

- DerivingVia は GND のスーパーセット
- GND は包まれる 一番内側の型 の実装を見る
- DerivingVia は Coercible な任意の型 の実装を、コストゼロで再利用することができる！

DerivingVia の射程は  
newtype に限らない

# DerivingVia の射程は newtype に限らない

- ・ 原論文では任意の同型な型の間で実装を共有する方法も提案されている

※この“同型”は正確には「総称的な表現がCoercible」というやや強い条件

# DerivingVia の射程は newtype に限らない

- ・ 原論文では任意の同型な型の間で実装を共有する方法も提案されている

 総称プログラミングとCoercionの合わせ技

※この“同型”は正確には「総称的な表現がCoercible」というやや強い条件

# Demo

JSON 変換インスタンスの静的定義  
同型な型の間での実装の共有

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



# 複雑な例

```
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
```

- ・ 型レベルでエンコードの仕様を指定
- ・ FromJSON / ToJSON で仕様が共有されることを静的に保証

# 同型の例

```
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"]}
-}
```

# 同型の例 (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 まとめ

# DerivingVia まとめ

- ・ GHC 8.6 から利用可能

# DerivingVia まとめ

- ・ GHC 8.6 から利用可能
- ・ **newtype** を使って deriving に使う実装を選択出来る

# DerivingVia まとめ

- ・ GHC 8.6 から利用可能
- ・ **newtype** を使って deriving に使う実装を選択出来る
- ・ coerce と Generics と組み合わせれば、表現が同じとは限らないが同型な型の実装も導出可能

# DerivingVia まとめ

- ・ GHC 8.6 から利用可能
- ・ **newtype** を使って deriving に使う実装を選択出来る
- ・ coerce と Generics と組み合わせれば、表現が同じとは限らないが同型な型の実装も導出可能
  - ・ 制約で表現できる同型なら他にも適用可能



まとめ

まとめ

# まとめ

- ・ newtype の用途は三つ

# まとめ

- ・ newtype の用途は三つ
  - ・ 実装の**隠蔽**／実装の**共有**／実装の**選択**

# まとめ

- ・ newtype の用途は三つ
  - ・ 実装の**隠蔽**／実装の**共有**／実装の**選択**
- ・ Data.Coerce を使うと**内部表現が同じ型同士をゼロコストでキャスト可能**

# まとめ

- ・ newtype の用途は三つ
  - ・ 実装の**隠蔽**／実装の**共有**／実装の**選択**
- ・ Data.Coerce を使うと**内部表現が同じ型同士をゼロコストでキャスト可能**
  - ・ newtype の出世はここから始まった

# まとめ

- ・ newtype の用途は三つ
  - ・ 実装の**隠蔽**／実装の**共有**／実装の**選択**
- ・ Data.Coerce を使うと**内部表現が同じ型同士をゼロコストでキャスト可能**
  - ・ newtype の出世はここから始まった
  - ・ 複合型も **Role 推論**・**註釈**で適切に扱える

# まとめ

- ・ newtype の用途は三つ
  - ・ 実装の**隠蔽**／実装の**共有**／実装の**選択**
- ・ Data.Coerce を使うと**内部表現が同じ型同士をゼロコストでキャスト可能**
  - ・ newtype の出世はここから始まった
  - ・ 複合型も **Role 推論**・**註釈**で適切に扱える
- ・ GHC 8.6 から **DerivingVia** で newtype により**導出節をカスタマイズ可能に！**



# 参考文献

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.

御清聴

ありがとうございました

# Any Question?

- ・ newtype の用途は三つ
  - ・ 実装の**隠蔽**／実装の**共有**／実装の**選択**
- ・ Data.Coerce を使うと**内部表現が同じ型同士をゼロコストでキャスト可能**
  - ・ newtype の出世はここから始まった
  - ・ 複合型も **Role 推論**・**註釈**で適切に扱える
- ・ GHC 8.6 から **DerivingVia** で newtype により**導出節をカスタマイズ可能に！**

おまけ

# なぜ Monoid と Foldable は 仲が いい の か ？

- Foldable の最小定義

```
class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m
```

- 実はこれは自由モノイドの普遍性の一部が由来

# 自由モナドの普遍性

- 自由モノイド関手の定義から
  - $\alpha$  から  $f \alpha$  への入射
  - $f$  が関手であること
  - $f \alpha$  自身がモノイドである事
- ……を除くと Foldable になる
  - 直接作る方法がなくてもいい
  - Hask** 圏の関手とは限らない
  - モノイドとは限らない、単なる木構造でも畳み込みたい

$$\begin{array}{ccc} f \alpha & \xrightarrow{\exists! \text{ foldMap } \varphi} & M \\ \uparrow [\cdot] & \nearrow \varphi & \\ \alpha & & \end{array}$$

# Foldable と Traversable

- ・ Traversable は自由モノイドとFoldableの中間
- ・ traverse は自由モノイドなら簡単に書ける

```
class    (Pointed f, Foldable f, ∀a. Monoid (f a)) ⇒ FreeMonoid f
instance (Pointed f, Foldable f, ∀a. Monoid (f a)) ⇒ FreeMonoid f
```

```
fold :: (FreeMonoid f) ⇒ (a → b → b) → b → f a → b
fold g n xs = appEndo (foldMap (Endo . g) xs) n
```

```
traverseF :: (FreeMonoid f, Applicative t)
           ⇒ (a → t b) → f a → t (f b)
```

```
traverseF f = fold (λa tb → (<>) <$> (iota <$> f a) <*> tb)
                (pure mempty)
```

8.6 から入る  
QuantifiedConstraints

Complete code : [src/Data/Foldable/Monoid.hs](#)