Data Type Generics

Type Safety Without Boilerplate

Juhana Laurinharju

Emblica Oy

October 21, 2018

Algebraic Data Types are Cool

```
data UserType     = Regular | Admin

newtype UserName = UserName String

data User         = User UserType UserName
```

Type classes give polymorphism

```
instance showUserType :: Show UserType where
  show Regular = "Regular"
  show Admin   = "Admin"
```

Type classes give polymorphism

```
instance showUserType :: Show UserType where
  show Regular = "Regular"
  show Admin   = "Admin"

instance showUserName :: Show UserName where
  show (UserName name) = "(UserName " <> name <> ")"
```

Type classes give polymorphism

```
instance showUserType :: Show UserType where
  show Regular = "Regular"
  show Admin   = "Admin"

instance showUserName :: Show UserName where
  show (UserName name) = "(UserName " <> name <> ")"

instance showUser :: Show User where
  show (User userType userName) =
    "(User " <> show userType <> " " <> show userName <> ")"
```

But that's a lot of work

- Could we somehow look at the structure of our types?

Anatomy of Algebraic Data Types

```
data Example = Example1 String String
             | Example2 String Int
```

Anatomy of Algebraic Data Types

```
data Example = Example1 String String
             | Example2 String Int
```

- Sums

Anatomy of Algebraic Data Types

```
data Example = Example1 String String
             | Example2 String Int
```

- Sums
- Contructors

Anatomy of Algebraic Data Types

```
data Example = Example1 String String
             | Example2 String Int
```

- Sums
- Contructors
- Products

Encoded as Types

Encoded as Types

- Sum

```
data Sum a b = Inl a | Inr b
```

Encoded as Types

- Sum

```
data Sum a b = Inl a | Inr b
```

- Constructors

```
newtype Constructor (name :: Symbol) a = Constructor a
```

Encoded as Types

- Sum

```
data Sum a b = Inl a | Inr b
```

- Constructors

```
newtype Constructor (name :: Symbol) a = Constructor a
```

- No Arguments

```
data NoArguments = NoArguments
```

Encoded as Types

- Sum

```
data Sum a b = Inl a | Inr b
```

- Constructors

```
newtype Constructor (name :: Symbol) a = Constructor a
```

- No Arguments

```
data NoArguments = NoArguments
```

- Argument

```
newtype Argument a = Argument a
```

Encoded as Types

- Sum

```
data Sum a b = Inl a | Inr b
```

- Constructors

```
newtype Constructor (name :: Symbol) a = Constructor a
```

- No Arguments

```
data NoArguments = NoArguments
```

- Argument

```
newtype Argument a = Argument a
```

- Products

```
data Product a b = Product a b
```

Encoded as Types

- Sum

```
data Sum a b = Inl a | Inr b
```

- Constructors

```
newtype Constructor (name :: Symbol) a = Constructor a
```

- No Arguments

```
data NoArguments = NoArguments
```

- Argument

```
newtype Argument a = Argument a
```

- Products

```
data Product a b = Product a b
```

- Empty types

```
data NoConstructors
```

To There and Back

```
class Generic a rep | a -> rep where
  to   :: rep -> a
  from :: a   -> rep
```

To There and Back

```
class Generic a rep | a -> rep where
  to   :: rep -> a
  from :: a   -> rep
```

- Compiler can generate you instances

To There and Back

```
class Generic a rep | a -> rep where
  to   :: rep -> a
  from :: a   -> rep
```

- Compiler can generate you instances

```
derive instance genericMyType :: Generic MyType _
```

To There and Back

```
class Generic a rep | a -> rep where
  to   :: rep -> a
  from :: a   -> rep
```

- Compiler can generate you instances

  ```
  derive instance genericMyType :: Generic MyType _
  ```

- But we should check out some examples

Extremely Simple Type

```
data Simple = Simple
```

Extremely Simple Type

```
data Simple = Simple

instance genericSimple
  :: Generic Simple (Constructor "Simple" NoArguments) where
```

Extremely Simple Type

```
data Simple = Simple

instance genericSimple
  :: Generic Simple (Constructor "Simple" NoArguments) where

  from Simple = Constructor NoArguments
```

Extremely Simple Type

```
data Simple = Simple

instance genericSimple
  :: Generic Simple (Constructor "Simple" NoArguments) where

  from Simple = Constructor NoArguments

  to (Constructor NoArguments) = Simple
```

Simple Sum Type

```
data Vote = Positive | Negative
```

Simple Sum Type

```
data Vote = Positive | Negative

instance genericVote
  :: Generic Vote (Sum (Constructor "Positive" NoArguments)
                       (Constructor "Negative" NoArguments)) where
```

Simple Sum Type

```
data Vote = Positive | Negative

instance genericVote
  :: Generic Vote (Sum (Constructor "Positive" NoArguments)
                       (Constructor "Negative" NoArguments)) where

  from Positive = Inl (Constructor NoArguments)
  from Negative = Inr (Constructor NoArguments)
```

Simple Sum Type

```
data Vote = Positive | Negative

instance genericVote
  :: Generic Vote (Sum (Constructor "Positive" NoArguments)
                       (Constructor "Negative" NoArguments)) where

  from Positive = Inl (Constructor NoArguments)
  from Negative = Inr (Constructor NoArguments)

  to (Inl (Constructor NoArguments)) = Positive
  to (Inr (Constructor NoArguments)) = Negative
```

Simple Product Type

```
data Vec3 = Vec3 Number Number Number
```

Simple Product Type

```
data Vec3 = Vec3 Number Number Number

instance genericVec3
  :: Generic Vec3 (Constructor "Vec3"
                     (Product
                       (Argument Number)
                       (Product
                         (Argument Number)
                         (Argument Number)))) where
```

Simple Product Type

```
data Vec3 = Vec3 Number Number Number

instance genericVec3
  :: Generic Vec3 (Constructor "Vec3"
                     (Product
                       (Argument Number)
                       (Product
                         (Argument Number)
                         (Argument Number)))) where

  from (Vec3 a b c)
    = Constructor
        (Product (Argument a)
                 (Product (Argument b)
                          (Argument c)))
```

Simple Product Type

```
data Vec3 = Vec3 Number Number Number

instance genericVec3
  :: Generic Vec3 (Constructor "Vec3"
                     (Product
                       (Argument Number)
                       (Product
                         (Argument Number)
                         (Argument Number)))) where

  from (Vec3 a b c)
    = Constructor
        (Product (Argument a)
                 (Product (Argument b)
                          (Argument c)))

  to (Constructor
        (Product (Argument a)
                 (Product (Argument b)
                          (Argument c)))) = Vec3 a b c
```

Sum of Products

```
data Example = Example1 String String
             | Example2 String Int
```

Sum of Products

```
data Example = Example1 String String
             | Example2 String Int

instance genericExample
  :: Generic Example (Sum (Constructor "Example1"
                             (Product (Argument String)
                                      (Argument String)))
                          (Constructor "Example2"
                             (Product (Argument String)
                                      (Argument Int)))) where
```

Sum of Products

```
data Example = Example1 String String
             | Example2 String Int

instance genericExample
  :: Generic Example (Sum (Constructor "Example1"
                            (Product (Argument String)
                                     (Argument String)))
                          (Constructor "Example2"
                            (Product (Argument String)
                                     (Argument Int)))) where

  from (Example1 str1 str2)
    = (Inl (Constructor (Product (Argument str1) (Argument str2))))
  from (Example2 str int)
    = (Inr (Constructor (Product (Argument str)  (Argument int))))
```

Sum of Products

```
data Example = Example1 String String
             | Example2 String Int

instance genericExample
  :: Generic Example (Sum (Constructor "Example1"
                            (Product (Argument String)
                                     (Argument String)))
                          (Constructor "Example2"
                            (Product (Argument String)
                                     (Argument Int)))) where

  from (Example1 str1 str2)
    = (Inl (Constructor (Product (Argument str1) (Argument str2))))
  from (Example2 str int)
    = (Inr (Constructor (Product (Argument str)  (Argument int))))

  to (Inl (Constructor (Product (Argument str1) (Argument str2))))
    = Example1 str1 str2
  to (Inr (Constructor (Product (Argument str)  (Argument int))))
    = Example2 str int
```

Using generic functions

- There are functions that work with these generic representations

Using generic functions

- There are functions that work with these generic representations

```
> log $ genericShow' $ from $ Example1 "Hello" "Tampere"
```

Using generic functions

- There are functions that work with these generic representations

```
> log $ genericShow' $ from $ Example1 "Hello" "Tampere"
(Example1 "Hello" "Tampere")
```

Using generic functions

- There are functions that work with these generic representations

```
> log $ genericShow' $ from $ Example1 "Hello" "Tampere"
(Example1 "Hello" "Tampere")
> log $ genericShow' $ from $ Just "Tampere"
```

Using generic functions

- There are functions that work with these generic representations

```
> log $ genericShow' $ from $ Example1 "Hello" "Tampere"
(Example1 "Hello" "Tampere")
> log $ genericShow' $ from $ Just "Tampere"
(Just "Tampere")
```

Using generic functions

- There are functions that work with these generic representations

```
> log $ genericShow' $ from $ Example1 "Hello" "Tampere"
(Example1 "Hello" "Tampere")
> log $ genericShow' $ from $ Just "Tampere"
(Just "Tampere")
```

- And they are type checked

Using generic functions

- There are functions that work with these generic representations

```
> log $ genericShow' $ from $ Example1 "Hello" "Tampere"
(Example1 "Hello" "Tampere")
> log $ genericShow' $ from $ Just "Tampere"
(Just "Tampere")
```

- And they are type checked

```
newtype NoShow a = NoShow a
```

Using generic functions

- There are functions that work with these generic representations

```
> log $ genericShow' $ from $ Example1 "Hello" "Tampere"
(Example1 "Hello" "Tampere")
> log $ genericShow' $ from $ Just "Tampere"
(Just "Tampere")
```

- And they are type checked

```
newtype NoShow a = NoShow a
> log $ genericShow' $ from $ Just (NoShow "Tampere")
```

Using generic functions

- There are functions that work with these generic representations

```
> log $ genericShow' $ from $ Example1 "Hello" "Tampere"
(Example1 "Hello" "Tampere")
> log $ genericShow' $ from $ Just "Tampere"
(Just "Tampere")
```

- And they are type checked

```
newtype NoShow a = NoShow a
> log $ genericShow' $ from $ Just (NoShow "Tampere")
  No type class instance was found for

    Data.Show.Show (NoShow String)
```

Shape of ADTs

```
Sum       (Constructor "Name1"            NoArguments)
     (Sum (Constructor "Name2"            (Argument Int))
          (Constructor "Name3" (Product (Argument String)
                                        (Argument Number))))
```

Shape of ADTs

```
Sum       (Constructor "Name1"            NoArguments)
    (Sum  (Constructor "Name2"            (Argument Int))
          (Constructor "Name3" (Product   (Argument String)
                                          (Argument Number))))
```

- Sum of
  - Constructor of
    - NoArguments
    - a single Argument
    - Product of Arguments
- NoConstructors

How to Write Such a Function?

- `Constructor`, `Sum`, `Product`, `Argument`, `NoArguments` and `NoConstructors` are all different types

How to Write Such a Function?

- `Constructor`, `Sum`, `Product`, `Argument`, `NoArguments` and `NoConstructors` are all different types
- How to write a function that possibly works on all of them?

How to Write Such a Function?

- `Constructor`, `Sum`, `Product`, `Argument`, `NoArguments` and `NoConstructors` are all different types
- How to write a function that possibly works on all of them?
- Typeclasses

Eq

```
class GenericEq a where
  genericEq' :: a -> a -> Boolean
```

Eq

```
class GenericEq a where
  genericEq' :: a -> a -> Boolean

instance genericEqNoArguments
  :: GenericEq NoArguments where
  genericEq' NoArguments NoArguments = true
```

Eq

```
class GenericEq a where
  genericEq' :: a -> a -> Boolean

instance genericEqNoArguments
  :: GenericEq NoArguments where
  genericEq' NoArguments NoArguments = true

instance genericEqConstructor
  :: GenericEq args
  => GenericEq (Constructor name args) where
  genericEq' (Constructor x) (Constructor y)
    = genericEq' x y
```

Eq

```
class GenericEq a where
  genericEq' :: a -> a -> Boolean

instance genericEqNoArguments
  :: GenericEq NoArguments where
  genericEq' NoArguments NoArguments = true

instance genericEqConstructor
  :: GenericEq args
  => GenericEq (Constructor name args) where
  genericEq' (Constructor x) (Constructor y)
    = genericEq' x y
```

```
> genericEq' (from Simple) (from Simple)
true
```

Cannot Handle Sums yet

```
> genericEq' (from Negative) (from Negative)
  No type class instance was found for
    Main.GenericEq (Sum (Constructor "Positive" NoArguments)
                        (Constructor "Negative" NoArguments))
```

Simple Sums

```
instance genericEqSum
  :: (GenericEq a, GenericEq b)
  => GenericEq (Sum a b) where
  genericEq' (Inl x) (Inl y) = genericEq' x y
  genericEq' (Inr x) (Inr y) = genericEq' x y
  genericEq' _       _       = false
```

Simple Sums

```
instance genericEqSum
  :: (GenericEq a, GenericEq b)
  => GenericEq (Sum a b) where
  genericEq' (Inl x) (Inl y) = genericEq' x y
  genericEq' (Inr x) (Inr y) = genericEq' x y
  genericEq' _       _       = false
```

```
> genericEq' (from Positive) (from Positive)
true
```

Simple Sums

```
instance genericEqSum
  :: (GenericEq a, GenericEq b)
  => GenericEq (Sum a b) where
  genericEq' (Inl x) (Inl y) = genericEq' x y
  genericEq' (Inr x) (Inr y) = genericEq' x y
  genericEq' _       _       = false
```

```
> genericEq' (from Positive) (from Positive)
true

> genericEq' (from Positive) (from Negative)
false
```

Argument

```
> genericEq' (from (UserName "KEKKONEN_3000"))
             (from (UserName "KEKKONEN_3000"))
  No type class instance was found for

    Main.GenericEq (Argument String)
```

Argument

```
instance genericEqArgument
  :: Eq arg
  => GenericEq (Argument arg) where
  genericEq' (Argument x) (Argument y) = x == y
```

Argument

```
instance genericEqArgument
  :: Eq arg
  => GenericEq (Argument arg) where
  genericEq' (Argument x) (Argument y) = x == y
```

```
> genericEq' (from (UserName "KEKKONEN_3000"))
             (from (UserName "KEKKONEN_3000"))
true
```

Argument

```
instance genericEqArgument
  :: Eq arg
  => GenericEq (Argument arg) where
  genericEq' (Argument x) (Argument y) = x == y
```

```
> genericEq' (from (UserName "KEKKONEN_3000"))
             (from (UserName "KEKKONEN_3000"))
true
```

```
> genericEq' (from (UserName "Alice"))
             (from (UserName "Bob"))
false
```

Product of Arguments

```
> genericEq' (from (Example2 "Alice" 1))
             (from (Example2 "Bob" 1))
  No type class instance was found for
    Main.GenericEq (Product (Argument String)
                            (Argument String))
```

Product of Arguments

```
instance genericEqProduct
  :: (GenericEq a, GenericEq b)
  => GenericEq (Product a b) where
```

Product of Arguments

```
instance genericEqProduct
  :: (GenericEq a, GenericEq b)
  => GenericEq (Product a b) where

  genericEq' (Product x1 y1) (Product x2 y2)
    = genericEq' x1 x2 && genericEq' y1 y2
```

Product of Arguments

```
instance genericEqProduct
  :: (GenericEq a, GenericEq b)
  => GenericEq (Product a b) where

  genericEq' (Product x1 y1) (Product x2 y2)
    = genericEq' x1 x2 && genericEq' y1 y2
```

```
> genericEq' (from (Example2 "Alice" 1))
             (from (Example2 "Bob" 1))
false
```

Product of Arguments

```
instance genericEqProduct
  :: (GenericEq a, GenericEq b)
  => GenericEq (Product a b) where

  genericEq' (Product x1 y1) (Product x2 y2)
    = genericEq' x1 x2 && genericEq' y1 y2
```

```
> genericEq' (from (Example2 "Alice" 1))
             (from (Example2 "Bob" 1))
false

> genericEq' (from (Example2 "Alice" 1))
              (from (Example2 "Alice" 1))
true
```

Stare in to the Void

- Should also cover empty types

```
instance genericEqNoConstructors
  :: GenericEq NoConstructors where
  genericEq' _ _ = true
```

Helper function

- To actually use this, write a helper function to do the generic conversion

```
genericEq :: forall a rep
           . Generic a rep
          => GenericEq rep
          => a -> a -> Boolean
genericEq x y = genericEq' (from x) (from y)
```

User code

- Now with all of this done, all that a user needs to do is write an instance that calls `genericEq`

```
instance eqExample :: Eq Example where
  eq = genericEq
```

User code

- Now with all of this done, all that a user needs to do is write an instance that calls `genericEq`

```
instance eqExample :: Eq Example where
  eq = genericEq
```

```
> (Example2 "KEKKONEN_3000" 10) == (Example2 "KEKKONEN_3000" 10)
true
```

User code

- Now with all of this done, all that a user needs to do is write an instance that calls `genericEq`

```
instance eqExample :: Eq Example where
  eq = genericEq
```

```
> (Example2 "KEKKONEN_3000" 10) == (Example2 "KEKKONEN_3000" 10)
true

> (Example2 "KEKKONEN_3000" 10) == (Example1 "KEKKONEN_3000" "10")
false
```

Show must go on

```
class GenericShow a where
  genericShow' :: a -> String
```

Show must go on

```
class GenericShow a where
  genericShow' :: a -> String

instance genericShowConstructorNoArguments
  :: IsSymbol name
  => GenericShow (Constructor name NoArguments) where
```

Show must go on

```
class GenericShow a where
  genericShow' :: a -> String

instance genericShowConstructorNoArguments
  :: IsSymbol name
  => GenericShow (Constructor name NoArguments) where

  genericShow' (Constructor NoArguments)
    = let constructorNameProxy = SProxy :: SProxy name
          constructorName = reflectSymbol constructorNameProxy
      in constructorName
```

Show must go on

```
class GenericShow a where
  genericShow' :: a -> String

instance genericShowConstructorNoArguments
  :: IsSymbol name
  => GenericShow (Constructor name NoArguments) where

  genericShow' (Constructor NoArguments)
    = let constructorNameProxy = SProxy :: SProxy name
          constructorName = reflectSymbol constructorNameProxy
      in constructorName

> log $ genericShow' $ from Simple
Simple
```

Sums

```
> log $ genericShow' $ from Positive
  No type class instance was found for
    Main.GenericShow (Sum (Constructor "Positive" NoArguments)
                          (Constructor "Negative" NoArguments))
```

Sums

```
> log $ genericShow' $ from Positive
  No type class instance was found for
    Main.GenericShow (Sum (Constructor "Positive" NoArguments)
                          (Constructor "Negative" NoArguments))

instance genericShowSum
  :: (GenericShow a, GenericShow b)
  => GenericShow (Sum a b) where
```

Sums

```
> log $ genericShow' $ from Positive
  No type class instance was found for
    Main.GenericShow (Sum (Constructor "Positive" NoArguments)
                          (Constructor "Negative" NoArguments))

instance genericShowSum
  :: (GenericShow a, GenericShow b)
  => GenericShow (Sum a b) where

  genericShow' (Inl x) = genericShow' x
  genericShow' (Inr y) = genericShow' y
```

Sums

```
> log $ genericShow' $ from Positive
  No type class instance was found for
    Main.GenericShow (Sum (Constructor "Positive" NoArguments)
                          (Constructor "Negative" NoArguments))

instance genericShowSum
  :: (GenericShow a, GenericShow b)
  => GenericShow (Sum a b) where

  genericShow' (Inl x) = genericShow' x
  genericShow' (Inr y) = genericShow' y

> log $ genericShow' $ from Positive
Positive
```

Single Argument

```
> log $ genericShow' $ from (Just "Tampere")
  No type class instance was found for
    Main.GenericShow (Constructor "Just" (Argument String))
```

Single Argument

```
> log $ genericShow' $ from (Just "Tampere")
  No type class instance was found for
    Main.GenericShow (Constructor "Just" (Argument String))

instance genericShowConstructorArgument
  :: (Show arg, IsSymbol name)
  => GenericShow (Constructor name (Argument arg)) where
```

Single Argument

```
> log $ genericShow' $ from (Just "Tampere")
  No type class instance was found for
    Main.GenericShow (Constructor "Just" (Argument String))

instance genericShowConstructorArgument
  :: (Show arg, IsSymbol name)
  => GenericShow (Constructor name (Argument arg)) where

  genericShow' (Constructor (Argument arg))
    = let constructorName = reflectSymbol (SProxy :: SProxy name)
          argStr = show arg
      in "(" <> constructorName <> " " <> argStr <> ")"
```

Single Argument

```
> log $ genericShow' $ from (Just "Tampere")
  No type class instance was found for
    Main.GenericShow (Constructor "Just" (Argument String))

instance genericShowConstructorArgument
  :: (Show arg, IsSymbol name)
  => GenericShow (Constructor name (Argument arg)) where

  genericShow' (Constructor (Argument arg))
    = let constructorName = reflectSymbol (SProxy :: SProxy name)
          argStr = show arg
      in "(" <> constructorName <> " " <> argStr <> ")"

> log $ genericShow' $ from (Just "Tampere")
(Just "Tampere")
```

Multiple Arguments

```
instance genericShowConstructorProduct
  :: (IsSymbol name, ??? a, ??? b)
  => GenericShow (Constructor name (Product a b))
```

Helper Function to the Rescue

```
class GenericShowArgs a where
  genericShowArgs :: a -> Array String
```

Helper Function to the Rescue

```
class GenericShowArgs a where
  genericShowArgs :: a -> Array String
```

- Turn a product of arguments to an array of string

Helper Function to the Rescue

```
class GenericShowArgs a where
  genericShowArgs :: a -> Array String
```

- Turn a product of arguments to an array of string

```
instance genericShowArgsArgument
  :: Show a
  => GenericShowArgs (Argument a) where
```

Helper Function to the Rescue

```
class GenericShowArgs a where
  genericShowArgs :: a -> Array String
```

- Turn a product of arguments to an array of string

```
instance genericShowArgsArgument
  :: Show a
  => GenericShowArgs (Argument a) where

  genericShowArgs (Argument x) = [show x]
```

Helper Function to the Rescue

```
class GenericShowArgs a where
  genericShowArgs :: a -> Array String
```

- Turn a product of arguments to an array of string

```
instance genericShowArgsArgument
  :: Show a
  => GenericShowArgs (Argument a) where

  genericShowArgs (Argument x) = [show x]

instance genericShowArgsProduct
  :: (GenericShowArgs a, GenericShowArgs b)
  => GenericShowArgs (Product a b) where
```

Helper Function to the Rescue

```
class GenericShowArgs a where
  genericShowArgs :: a -> Array String
```

- Turn a product of arguments to an array of string

```
instance genericShowArgsArgument
  :: Show a
  => GenericShowArgs (Argument a) where

  genericShowArgs (Argument x) = [show x]

instance genericShowArgsProduct
  :: (GenericShowArgs a, GenericShowArgs b)
  => GenericShowArgs (Product a b) where

  genericShowArgs (Product x y)
    = genericShowArgs x <> genericShowArgs y
```

Multiple Arguments

```
instance genericShowConstructorProduct
  :: (IsSymbol name, GenericShowArgs (Product a b))
  => GenericShow (Constructor name (Product a b)) where
```

Multiple Arguments

```
instance genericShowConstructorProduct
  :: (IsSymbol name, GenericShowArgs (Product a b))
  => GenericShow (Constructor name (Product a b)) where

  genericShow' (Constructor prod)
    = let constructorName = reflectSymbol (SProxy :: SProxy name)
          args = genericShowArgs prod
          argStr = intercalate " " args
      in "(" <> constructorName <> " " <> argStr <> ")"
```

Multiple Arguments

```
instance genericShowConstructorProduct
  :: (IsSymbol name, GenericShowArgs (Product a b))
  => GenericShow (Constructor name (Product a b)) where

  genericShow' (Constructor prod)
    = let constructorName = reflectSymbol (SProxy :: SProxy name)
          args = genericShowArgs prod
          argStr = intercalate " " args
      in "(" <> constructorName <> " " <> argStr <> ")"
```

```
> log $ genericShow' $ from (Example2 "hep" 2)
(Example2 "hep" 2)
```

Simplified

```
instance genericShowArgsNoArguments
  :: GenericShowArgs NoArguments where
  genericShowArgs NoArguments = []
```

Simplified

```
instance genericShowArgsNoArguments
  :: GenericShowArgs NoArguments where
  genericShowArgs NoArguments = []

instance genericShowConstructor
  :: (GenericShowArgs params, IsSymbol name)
  => GenericShow (Constructor name params) where
```

Simplified

```
instance genericShowArgsNoArguments
  :: GenericShowArgs NoArguments where
  genericShowArgs NoArguments = []

instance genericShowConstructor
  :: (GenericShowArgs params, IsSymbol name)
  => GenericShow (Constructor name params) where

  genericShow' (Constructor args) =
    let constructorNameProxy = SProxy :: SProxy name
        constructorName = reflectSymbol constructorNameProxy
        paramsStringArray = genericShowArgs args
    in case paramsStringArray of
      []       -> constructorName
      someArgs ->
        "(" <> constructorName <> " "
            <> (intercalate " " someArgs)
            <> ")"
```

Helpers

```
genericShow :: forall a rep
             . Generic a rep
            => GenericShow rep
            => a
            -> String
genericShow x = genericShow' (from x)
```

Helpers

```
genericShow :: forall a rep
             . Generic a rep
            => GenericShow rep
            => a
            -> String
genericShow x = genericShow' (from x)

instance showExample :: Show Example where
  show = genericShow
```

Helpers

```
genericShow :: forall a rep
             . Generic a rep
            => GenericShow rep
            => a
            -> String
genericShow x = genericShow' (from x)

instance showExample :: Show Example where
  show = genericShow
```

```
> log $ show $ (Example2 "hep" 2)
(Example2 "hep" 2)
```

Recap

- `Sum` of
  - `Constructor` of
    - `NoArguments`
    - a single `Argument`
    - `Product` of `Arguments`
- `NoContructors`