Generalized Abstract

GHC.Generics

Ryan Scott
Indiana University

Haskell Implementors Workshop 2018
St. Louis, MO
GHC.Generics

GHC’s most popular datatype-generic programming library.
GHC.Generics

GHC’s most popular datatype-generic programming library.

data ADT a
    = MkADT1 a
    | MkADT2 a
deriving Generic
GHC.Generics

GHC’s most popular datatype-generic programming library.

data ADT a
  = MkADT1 a
  | MkADT2 a
  deriving Generic
GHC.Generics

GHC’s most popular datatype-generic programming library.

data ADT a
    = MkADT1 a
    | MkADT2 a
deriving Generic

data GADT a where
    MkGADT1 :: GADT Int
    MkGADT2 :: GADT Bool
deriving Generic
GHC.Generics

GHC’s most popular datatype-generic programming library.

| data ADT a                        | data GADT a where                  |
|                                  | MkGADT1 :: GADT Int                |
|                                  | MkGADT2 :: GADT Bool               |
| = MkADT1 a                       | deriving Generic                  |
| MkADT2 a                         | deriving Generic                  |
| deriving Generic                 |                                  |
GHC.Generics

GHC’s most popular datatype-generic programming library.

class Generic a where -- Can be derived
  type Rep a
  from :: a -> Rep a
  to   :: Rep a -> a
GHC.Generics

GHC’s most popular datatype-generic programming library.

class NFData a where
  rnf :: a -> ()

instance NFData a => NFData [a] where
  rnf [] = ()
  rnf (x:xs) = rnf x `seq` rnf xs

We’ll continue to use NFData as a running example.
What can GHC.Generics do?

Generically represent any (simple) algebraic data type as a composition of *representation types*.

```haskell
data U1 = U1 -- No fields
newtype K1 c = K1 c -- One field
data a :*: b = a :*: b -- Products
data a :+: b = L1 a | R1 b -- Sums
```
What can GHC.Generics do?

Generically represent any (simple) algebraic data type as a composition of *representation types*.

-- Example instance

```haskell
instance Generic [a] where
  type Rep [a] =
    U1               -- [] constructor
  :+: (K1 a :+:: K1 [a]) -- (:) constructor

from [] = L1 U1
from (x:xs) = R1 (K1 x :+:: K1 xs)
to (L1 U1) = []
to (R1 (K1 x :+:: K1 xs)) = x:xs
```
What can GHC.Generics do?

Generically represent any (simple) algebraic data type as a composition of representation types.

```haskell
instance NFData U1 where
  rnf U1 = ()

instance NFData c => NFData (K1 c) where
  rnf (K1 c) = rnf c
```
What can GHC.Generics do?

Generically represent any (simple) algebraic data type as a composition of representation types.

```haskell
instance (NFData a, NFData b) => NFData (a :+: b) where
  rnf (L1 x) = rnf x
  rnf (R1 y) = rnf y
```

```haskell
instance (NFData a, NFData b) => NFData (a :*: b) where
  rnf (x :*: y) = rnf x `seq` rnf y
```
What can GHC.Generics do?

Generically represent any (simple) algebraic data type as a composition of representation types.

```haskell
instance NFData a => NFData [a] where
  rnf [] = ()
  rnf (x:xs) = rnf x `seq` rnf xs
```
What can GHC.Generics do?

Generically represent any (simple) algebraic data type as a composition of *representation types*.

```haskell
instance NFData a => NFData [a] where
  rnf [] = ()
  rnf (x:xs) = rnf x `seq` rnf xs

genericRNF :: (Generic a, NFData (Rep a)) => a -> ()
genericRNF x = rnf (from x)
```
What can GHC.Generics do?

Generically represent any (simple) algebraic data type as a composition of representation types.

```haskell
instance NFData a => NFData [a] where
  rnf = genericRNF

genericRNF :: (Generic a, NFData (Rep a)) => a -> ()
genericRNF x = rnf (from x)
```
What can’t GHC.Generics do?

It can’t represent GADTs... but why not?
What can’t GHC.Generics do?

data GADTEx :: Type -> Type -> Type where
    GADTEx1 :: NFData a => a -> GADTEx a b
    GADTEx2 :: NFData b => b -> GADTEx a b
What can’t GHC.Generics do?

```haskell
data GADTEx :: Type -> Type -> Type
where
  GADTEx1 :: NFData a => a -> GADTEx a b
  GADTEx2 :: NFData b => b -> GADTEx a b
```
What can’t GHC.Generics do?

data GADTEEx :: Type -> Type -> Type where
GADTEEx1 :: NFData a => a -> GADTEEx a b
GADTEEx2 :: NFData b => b -> GADTEEx a b

-- Like to be able to derive this:
instance NFData (GADTEEx a b) where
  rnf (GADTEEx1 x) = rnf x
  rnf (GADTEEx2 y) = rnf y
What can’t GHC.Generics do?

data GADTEx :: Type -> Type -> Type where
  GADTEx1 :: NFData a => a -> GADTEx a b
  GADTEx2 :: NFData b => b -> GADTEx a b

-- Attempt 1
instance (NFData a, NFData b)
  => Generic (GADTEx a b) where
  type Rep (GADTEx a b) = K1 a :+: K1 b

  from (GADTEx1 x) = L1 (K1 x)
  from (GADTEx2 y) = R1 (K1 y)
  to (L1 (K1 x)) = GADTEx1 x
  to (R1 (K1 y)) = GADTEx2 y
What can’t GHC.Generics do?

data GADTEx :: Type -> Type -> Type where
  GADTEx1 :: NFData a => a -> GADTEx a b
  GADTEx2 :: NFData b => b -> GADTEx a b

-- Attempt 1 (continued)
instance (NFData a, NFData b)
  => NFData (GADTEx a b) where
  rnf = genericRNF
What can’t GHC.Generics do?

data GADTEEx :: Type -> Type -> Type where
  GADTEEx1 :: NFData a => a -> GADTEEx a b
  GADTEEx2 :: NFData b => b -> GADTEEx a b

-- Attempt 1 (continued)
instance (NFData a, NFData b) => NFData (GADTEEx a b) where
  rnf = genericRNF
What can’t GHC.Generics do?

data GADTEEx :: Type -> Type -> Type where
  GADTEEx1 :: NFData a => a -> GADTEEx a b
  GADTEEx2 :: NFData b => b -> GADTEEx a b

-- Attempt 1 (continued)
instance (NFData a, NFData b) => NFData (GADTEEx a b) where
  rnf = genericRNF
What *can’t* GHC.Generics do?

data GADTEx :: Type -> Type -> Type where
  GADTEx1 :: NFData a => a -> GADTEx a b
  GADTEx2 :: NFData b => b -> GADTEx a b

-- Attempt 1
instance (NFData a, NFData b)
  => Generic (GADTEx a b) where
  type Rep (GADTEx a b) = K1 a :+: K1 b

  from (GADTEx1 x) = L1 (K1 x)
  from (GADTEx2 y) = R1 (K1 y)
  to (L1 (K1 x)) = GADTEx1 x
  to (R1 (K1 y)) = GADTEx2 y
What can’t GHC.Generics do?

data GADTEEx :: Type -> Type -> Type where
  GADTEEx1 :: NFData a => a -> GADTEEx a b
  GADTEEx2 :: NFData b => b -> GADTEEx a b

-- Attempt 1
instance (NFData a, NFData b)
  => Generic (GADTEEx a b) where
  type Rep (GADTEEx a b) = K1 a :+: K1 b

  from (GADTEEx1 x) = L1 (K1 x)
  from (GADTEEx2 y) = R1 (K1 y)
  to (L1 (K1 x)) = GADTEEx1 x
  to (R1 (K1 y)) = GADTEEx2 y
Solution: invent a new representation type

We need to generically represent constructors with existential contexts.
Solution: invent a new representation type

We need to generically represent constructors with existential contexts.

```haskell
data ExConstr :: Constraint -> Type -> Type where
  ExConstr :: c => x -> ExConstr c x
```
Solution: invent a new representation type

We need to generically represent constructors with existential contexts.

```haskell
data ExConstr :: Constraint -> Type -> Type where
    ExConstr :: c => x -> ExConstr c x
```
Solution: invent a new representation type

We need to generically represent constructors with existential contexts.

data ExConstr :: Constraint -> Type -> Type where
    ExConstr :: c => x -> ExConstr c x

instance (c => NFData x)
    => NFData (ExConstr c x) where
    rnf (ExConstr x) = rnf x
Solution: invent a new representation type

We need to generically represent constructors with existential contexts.

```haskell
data ExConstr :: Constraint -> Type -> Type where
  ExConstr :: c => x -> ExConstr c x

instance (c => NFData x) => NFData (ExConstr c x) where
  rnf (ExConstr x) = rnf x
```

QuantifiedConstraints! (New in GHC 8.6)
What can’t GHC.Generics do?

data GADTEx :: Type -> Type -> Type where
  GADTEx1 :: NFData a => a -> GADTEx a b
  GADTEx2 :: NFData b => b -> GADTEx a b

-- Attempt 2
instance Generic (GADTEx a b) where
  ...
What can’t GHC.Generics do?

data GADTEx :: Type -> Type -> Type where
  GADTEx1 :: NFData a => a -> GADTEx a b
  GADTEx2 :: NFData b => b -> GADTEx a b

-- Attempt 2
instance Generic (GADTEx a b) where
  type Rep (GADTEx a b) =
    ExConstr (NFData a) (K1 a) :+: ExConstr (NFData b) (K1 b)
  from (GADTEx1 x) = L1 (ExConstr (K1 x))
  from (GADTEx2 y) = R1 (ExConstr (K1 y))
  to (L1 (ExConstr (K1 x))) = GADTEx1 x
  to (R1 (ExConstr (K1 y))) = GADTEx2 y
What can’t GHC.Generics do?

data GADTEx :: Type -> Type -> Type where
  GADTEx1 :: NFData a => a -> GADTEx a b
  GADTEx2 :: NFData b => b -> GADTEx a b

-- Attempt 2
instance Generic (GADTEx a b) where
  type Rep (GADTEx a b) =
    ExConstr (NFData a) (K1 a)
    :+: ExConstr (NFData b) (K1 b)
  from (GADTEx1 x) = L1 (ExConstr (K1 x))
  from (GADTEx2 y) = R1 (ExConstr (K1 y))
  to (L1 (ExConstr (K1 x))) = GADTEx1 x
  to (R1 (ExConstr (K1 y))) = GADTEx2 y
What can’t GHC.Generics do?

data GADTEEx :: Type -> Type -> Type where
  GADTEEx1 :: NFData a => a -> GADTEEx a b
  GADTEEx2 :: NFData b => b -> GADTEEx a b

-- Attempt 2
instance Generic (GADTEEx a b) where
  type Rep (GADTEEx a b) =
    ExConstr (NFData a) (K1 a)
    :+: ExConstr (NFData b) (K1 b)
  from (GADTEEx1 x) = L1 (ExConstr (K1 x))
  from (GADTEEx2 y) = R1 (ExConstr (K1 y))
  to (L1 (ExConstr (K1 x))) = GADTEEx1 x
  to (R1 (ExConstr (K1 y))) = GADTEEx2 y
data GADTEx :: Type -> Type -> Type where
  GADTEx1 :: NFData a => a -> GADTEx a b
  GADTEx2 :: NFData b => b -> GADTEx a b

-- Attempt 2 (continued)
instance NFData (GADTEx a b) where
  rnf = genericRNF
What can’t can GHC.Generics do?

data GADTEx :: Type -> Type -> Type where
    GADTEx1 :: NFData a => a -> GADTEx a b
    GADTEx2 :: NFData b => b -> GADTEx a b

-- Attempt 2 (continued)
instance NFData (GADTEx a b) where
    rnf = genericRNF
What about type indices?

```haskell
data GADTEx :: Type -> Type -> Type where
    -- ...
    GADTEx3 :: Int -> Bool -> GADTEx Int Bool
```
What about type indices?

data GADTEx :: Type -> Type -> Type where
  -- This...
GADTEx3 :: Int -> Bool -> GADTEx Int Bool

  -- ...is wholly equivalent to this:
GADTEx3 :: (a ~ Int, b ~ Bool)
  => Int -> Bool -> GADTEx a b
What about type indices?

data GADTE\text{Ex} :: Type \to Type \to Type \text{ where}

-- This...
GADTE\text{Ex}3 :: Int \to \text{Bool} \to GADTE\text{Ex} \text{ Int Bool}

-- ...is wholly equivalent to this:
GADTE\text{Ex}3 :: (a \sim \text{Int}, b \sim \text{Bool}) \to \text{Int} \to \text{Bool} \to GADTE\text{Ex} \ a \ b
What about type indices?

data GADTEx :: Type -> Type -> Type where
    -- This...
    GADTEx3 :: Int -> Bool -> GADTEx Int Bool

    -- ...is wholly equivalent to this:
    GADTEx3 :: (a ~ Int, b ~ Bool) => Int -> Bool -> GADTEx a b
What about existentially quantified type variables?

data SomeNFThing :: Type where
  SomeNFThing :: forall a. NFData a
  => a -> SomeNFThing
What about existentially quantified type variables?

data SomeNFThing :: Type where
  SomeNFThing :: forall a. NFData a => a -> SomeNFThing
What about existentially quantified type variables?

data SomeNFThing :: Type where
    SomeNFThing :: forall a. NFData a => a -> SomeNFThing

We’d like to be able to write something like this:

instance Generic SomeNFThing where
    type Rep SomeNFThing =
        ExQuant (\a -> ExConstr (NFData a) (K1 a))
        -- ^ Type-level lambda
Faking type-level lambdas

data SomeNFThing :: Type where
SomeNFThing :: forall a. NFData a => a -> SomeNFThing

instance Generic SomeNFThing where
  type Rep SomeNFThing =
    ExQuant RepAux

  type RepAux (a :: Type) =
    ExConstr (NFData a) (K1 a)
data SomeNFThing :: Type where
  SomeNFThing :: forall a. NFData a => a -> SomeNFThing

instance Generic SomeNFThing where
  type Rep SomeNFThing =
    ExQuant RepAux

  type RepAux (a :: Type) =
    ExConstr (NFData a) (K1 a)

Can’t partially apply type synonyms :(  

Solution: defunctionalization

(Reynolds 1972, Eisenberg and Stolarek 2014)
Solution: defunctionalization
(Reynolds 1972, Eisenberg and Stolarek 2014)

type a ~> b = a -> b -> Type

type family
  Apply (f :: Type ~> Type) (x :: a) :: b
Solution: defunctionalization
(Reynolds 1972, Eisenberg and Stolarek 2014)

type a ~> b = a -> b -> Type

type family
  Apply (f :: Type ~> Type) (x :: a) :: b

type RepAux (a :: Type) =
  ExConstr (NFData a) (K1 a)
Solution: defunctionalization
(Reynolds 1972, Eisenberg and Stolarek 2014)

type a \to b = a \to b \to \text{Type}
type family
  \text{Apply} (f :: \text{Type} \to \text{Type}) (x :: a) :: b

type \text{RepAux} (a :: \text{Type}) =
  \text{ExConstr} (\text{NFData} a) (\text{K1} a)
data \text{RepAuxSym} :: \text{Type} \to \text{Type}
type instance \text{Apply} \text{RepAuxSym} a = \text{RepAux} a
Solution: defunctionalization
(Reynolds 1972, Eisenberg and Stolarek 2014)

data ExQuant :: (Type ~> Type) -> Type where
  ExQuant :: forall f x.
  -- ^ x is existential
  Apply f x -> ExQuant f
Solution: defunctionalization
(Reynolds 1972, Eisenberg and Stolarek 2014)

instance (forall x. NFData (Apply f x)) => NFData (ExQuant f) where
rnf (ExQuant x) = rnf x
Solution: defunctionalization
(Reynolds 1972, Eisenberg and Stolarek 2014)

```
instance (forall x. NFData (Apply f x)) => NFData (ExQuant f) where
rnf (ExQuant x) = rnf x
```
Solution: defunctionalization
(Reynolds 1972, Eisenberg and Stolarek 2014)

instance (forall x. NFData (Apply f x))
  => NFData (ExQuant f) where
  rnf (ExQuant x) = rnf x

- Illegal type synonym family application in instance: Apply f x
Solution: defunctionalization
(Reynolds 1972, Eisenberg and Stolarek 2014)

data ExQuant :: (Type ~> Type) -> Type where
ExQuant :: forall f x.
    -- ^ x is existential
    Apply f x -> ExQuant f
Solution: defunctionalization
(Reynolds 1972, Eisenberg and Stolarek 2014)

data ExQuant :: (Type ~> Type) -> Type where ExQuant
  :: forall f x.
     -- ^ x is existential
     WrappedApply f x -> ExQuant f

newtype WrappedApply f x = WrapApply (Apply f x)
Solution: defunctionalization
(Reynolds 1972, Eisenberg and Stolarek 2014)

instance
  (forall x. NFData (Apply f x))
  => NFData (ExQuant f) where
  rnf (ExQuant x) = rnf x
Solution: defunctionalization
(Reynolds 1972, Eisenberg and Stolarek 2014)

instance
  (forall x. NFData (WrappedApply f x))
  => NFData (ExQuant f) where
  rnf (ExQuant x) = rnf x
Solution: defunctionalization
(Reynolds 1972, Eisenberg and Stolarek 2014)

instance
  (forall x. NFData (WrappedApply f x))
  => NFData (ExQuant f) where
  rnf (ExQuant x) = rnf x

instance NFData (Apply f x)
  => NFData (WrappedApply f x) where
  rnf (WrapApply x) = rnf x
Faking type-level lambdas

data SomeNFThing :: Type where
  SomeNFThing :: forall a. NFData a => a -> SomeNFThing

instance Generic SomeNFThing where
  type Rep SomeNFThing =
    ExQuant ???

  type RepAux (a :: Type) =
    ExConstr (NFData a) (K1 a)
Faking type-level lambdas

```haskell
data SomeNFThing :: Type where
    SomeNFThing :: forall a. NFData a => a -> SomeNFThing
```

```haskell
instance Generic SomeNFThing where
    type Rep SomeNFThing =
        ExQuant RepAuxSym

    type RepAux (a :: Type) =
        ExConstr (NFData a) (K1 a)
```
Faking type-level lambdas

data SomeNFThing :: Type where
SomeNFThing :: forall a. NFData a => a -> SomeNFThing

instance Generic SomeNFThing where
  type Rep SomeNFThing =
    ExQuant RepAuxSym
  from (SomeNFThing x) =
    ExQuant (WrapApply (ExConstr (K1 x)))
  to (ExQuant (WrapApply (ExConstr (K1 x)))) =
    SomeNFThing x
Faking type-level lambdas

data SomeNFThing :: Type where
  SomeNFThing :: forall a. NFData a
  => a -> SomeNFThing

instance NFData SomeNFThing where
  rnf = genericRNF
Faking type-level lambdas

data SomeNFThing :: Type where
    SomeNFThing :: forall a. NFData a
    => a -> SomeNFThing

instance NFData SomeNFThing where
    rnf = genericRNF
Can we ditch defunctionalization?
Can we ditch defunctionalization?

• Not today... but maybe tomorrow!
  • Type-level lambdas (Eisenberg 2016)

• Unsaturated type synonyms (Kiss 2018)
Related work

- *The Gentle Art of Levitation* (Chapman et al. 2010)
- *Generic Programming of All Kinds* (Serrano and Miraldo 2018)
Open questions

• Rank-\(n\) types
  • data Foo :: Type where
    MkFoo :: forall b. ((forall a. a -> a) -> b -> b) -> Foo

• Performance
Takeaways

• With these extensions, deriving Generic would work for ADTs and GADTs alike
• No breaking changes to GHC.Generics required
• Some parts are hairy... but could be made less so with the help of ongoing work in GHC

Template Haskell prototype:
https://github.com/dreixel/generic-deriving/tree/experimental