

Applying Type-Level and Generic Programming in Haskell

Summer School on Generic and Effectful Programming

Andres Löh

6–10 July 2015



Datatype-generic programming

Express algorithms that make use of the structure of datatypes

Datatype-generic programming

Express algorithms that make use of the structure of datatypes

```
eqA :: A -> A -> Bool
```

A class

```
class Generic a where  
  type Rep a  
  from :: a -> Rep a  
  to   :: Rep a -> a
```

where `from` and `to` are inverses.

A class

```
class Generic a where  
  type Rep a  
  from :: a -> Rep a  
  to   :: Rep a -> a
```

where `from` and `to` are inverses.

```
geq :: Generic a => Rep a -> Rep a -> Bool
```

A class

```
class Generic a where  
  type Rep a  
  from :: a -> Rep a  
  to   :: Rep a -> a
```

where `from` and `to` are inverses.

```
geq :: Generic a => Rep a -> Rep a -> Bool
```

```
eq :: Generic a => a -> a -> Bool  
eq x y = geq (from x) (from y)
```

Much flexibility in the details, in particular the definition of Rep .

Much flexibility in the details, in particular the definition of `Rep`.

The choice of `Rep` determines expressive power and flavour of generic programs.

Much flexibility in the details, in particular the definition of `Rep`.

The choice of `Rep` determines expressive power and flavour of generic programs.

In this lecture series: `generics-sop`.

Applications

- ▶ (De-)serialization
- ▶ Data generation
- ▶ Data traversals
- ▶ Data navigation
- ▶ ...

The generics-sop view on data, informally

Sample datatypes

```
data Maybe a    = Nothing | Just a
data Either a b = Left a  | Right b
data Group      = Group Char Bool Int
data Expr       = NumL Int
                  | BoolL Bool
                  | Add Expr Expr
                  | If Expr Expr Expr
```

Sample datatypes

```
data Maybe a      = Nothing | Just a
data Either a b    = Left a | Right b
data Group         = Group Char Bool Int
data Expr          = NumL Int
                   | BoolL Bool
                   | Add Expr Expr
                   | If Expr Expr Expr
```

- ▶ **Choice** between constructors,
- ▶ each with a **sequence** of arguments.

Sample datatypes

```
data Maybe a      = Nothing | Just a
data Either a b    = Left a | Right b
data Group         = Group Char Bool Int
data Expr          = NumL Int
                   | BoolL Bool
                   | Add Expr Expr
                   | If Expr Expr Expr
```

- ▶ **Choice** between constructors,
- ▶ each with a **sequence** of arguments.

$C_i \ x_0 \dots x_{n_i-1}$

The plan

$C_i \ x_0 \dots x_{n_i-1}$

- ▶ Choice between constructors modelled as an n -ary sum.
- ▶ Sequence of fields modelled as an n -ary product.

We'll need Haskell type-level programming concepts along the way.

Extensions, extensions

DataKinds
GADTs
TypeOperators
TypeFamilies
RankNTypes
ConstraintKinds
MultiParamTypeClasses
UndecidableInstances
StandaloneDeriving
ScopedTypeVariables
PolyKinds
FlexibleInstances
FlexibleContexts
DefaultSignatures

Plan for the week

Today:

- ▶ Learn about n -ary products.
- ▶ Along the way, discuss everything we need in terms of Haskell type-level programming features.

Wednesday:

- ▶ Introduce n -ary sums and the generics-sop view.
- ▶ Representing datatypes using generics-sop.
- ▶ Simple applications.

Friday:

- ▶ More applications.

Kinds and data kinds

Types and kinds

- ▶ Values / terms have **types**.
- ▶ Types have **kinds**.

Example:

```
GHCi> :type 'x'  
'x' :: Char  
GHCi> :kind Char  
Char :: *
```

Stars and functions

```
Int      :: *  
Double  :: *  
Bool     :: *  
Char     :: *  
( )     :: *  
Void     :: *
```

```
data () = ()  -- one value  
data Void    -- no values
```

Stars and functions – contd.

```
Maybe  :: * -> *  
[]      :: * -> *  
IO      :: * -> *  
(, )    :: * -> * -> *  
Either  :: * -> * -> *
```

Stars and functions – contd.

```
Maybe  :: * -> *  
[]      :: * -> *  
IO      :: * -> *  
(, )    :: * -> * -> *  
Either  :: * -> * -> *
```

```
Maybe Int           :: *  
IO [Bool]            :: *  
Either Char          :: * -> *  
Either Char (Maybe Int) :: *  
IO Maybe             -- kind error
```

Data kinds and promotion

```
data Bool = False | True
```

Defines a datatype with (data) constructors:

```
Bool  :: *  
False :: Bool  
True  :: Bool
```

Data kinds and promotion

```
data Bool = False | True
```

Defines a datatype with (data) constructors:

```
Bool  :: *  
False :: Bool  
True  :: Bool
```

Defines also a kind with (type) constructors:

```
Bool    :: □  
'False :: Bool  
'True  :: Bool
```

Both `False` and `True` are uninhabited.

Data kinds and promotion – contd.

Quotes are generally optional; `False` and `True` also allowed.

```
GHCi> :kind Bool
Bool :: *
GHCi> :type True
True :: Bool
GHCi> :type False
False :: Bool
GHCi> :kind 'True
'True :: Bool
GHCi> :kind 'False
'False :: Bool
GHCi> :kind True
True :: Bool
GHCi> :kind False
False :: Bool
```

Generalized algebraic data types (GADTs)

Generalizing lists in several steps

```
type T1 = [Int]
type T2 = Vec Int (Suc (Suc (Suc Zero)))
type T3 = HList '[Char, Bool, Int]
type T4 = NP Maybe '[Char, Bool, Int]
```

Generalizing lists in several steps

```
type T1 = [Int]
type T2 = Vec Int (Suc (Suc (Suc Zero)))
type T3 = HList '[Char, Bool, Int]
type T4 = NP Maybe '[Char, Bool, Int]
```

```
[1, 2, 3, 4, 5]           :: T1
[1, 2, 3]                 :: T2
['x', False, 3]          :: T3
[Just 'x', Nothing, Just 3] :: T4
```

Vectors – promoting natural numbers

```
data Nat = Zero | Suc Nat
```

As a type:

```
Nat  :: *  
Zero :: Nat  
Suc  :: Nat -> Nat
```

As a kind:

```
Nat    :: □  
'Zero :: Nat  
'Suc   :: Nat -> Nat
```

From lists to vectors

```
data [a] = [] | a : [a]
```

From lists to vectors

```
data [a] = [] | a : [a]
```

Renaming constructors:

```
data List a = LNil | LCons a (List a)
```

From lists to vectors

```
data [a] = [] | a : [a]
```

Renaming constructors:

```
data List a = LNil | LCons a (List a)
```

GADT syntax:

```
data List (a :: *) where  
  LNil  :: List a  
  LCons :: a -> List a -> List a
```


Defining vectors

Lists:

```
data List (a :: *) where  
  LNil  :: List a  
  LCons :: a -> List a -> List a
```

Vectors:

```
data Vec (a :: *) (n :: Nat) where  
  VNil  :: Vec a Zero  
  VCons :: a -> Vec a n -> Vec a (Suc n)  
infixr 5 'VCons'  -- for infix use  
deriving instance Show a => Show (Vec a n)
```

Vector examples

```
GHCi> :type VNil
VNil :: Vec a 'Zero
GHCi> :type 'x' 'VCons' VNil
'x' 'VCons' VNil :: Vec Char ('Suc 'Zero)
GHCi> :type 'y' 'VCons' 'x' 'VCons' VNil
'y' 'VCons' 'x' 'VCons' VNil
      :: Vec Char ('Suc ('Suc 'Zero))
```

Pattern matching on GADTs

```
vtail :: Vec a (Suc n) -> Vec a n  
vtail (VCons x xs) = xs
```

No case for `VNil` needed (or possible).

Pattern matching on GADTs

```
vtail :: Vec a (Suc n) -> Vec a n  
vtail (VCons x xs) = xs
```

No case for `VNil` needed (or possible).

Example:

```
GHCi> vtail ('x' 'VCons' VNil)  
VNil
```

```
GHCi> vtail (vtail ('x' 'VCons' VNil))
```

results in a type error!

Pattern matching on GADTs – contd.

```
vmap :: (a -> b) -> Vec a n -> Vec b n  
vmap f VNil          = VNil  
vmap f (x 'VCons' xs) = f x 'VCons' vmap f xs
```

Pattern matching on GADTs – contd.

```
vmap :: (a -> b) -> Vec a n -> Vec b n  
vmap f VNil          = VNil  
vmap f (x 'VCons' xs) = f x 'VCons' vmap f xs
```

First case:

$n \sim \text{Zero}$

Pattern matching on GADTs – contd.

```
vmap :: (a -> b) -> Vec a n -> Vec b n  
vmap f VNil          = VNil  
vmap f (x 'VCons' xs) = f x 'VCons' vmap f xs
```

First case:

$n \sim \text{Zero}$

Second case:

$n \sim \text{Suc } n'$

An applicative interface for vectors

The **Applicative** class

```
class Functor f where  
  fmap  :: (a -> b) -> f a -> f b  
  
class (Functor f) => Applicative f where  
  pure  :: a -> f a  
  (<*>) :: f (a -> b) -> f a -> f b
```

We've already seen that we can map over vectors ...

Generalizing replicate

```
replicate :: Int -> a -> [a]
replicate n x
  | n <= 0      = []
  | otherwise = x : replicate (n - 1) x
```

What to do with the `Int` for vectors?

Generalizing `replicate`

```
replicate :: Int -> a -> [a]
replicate n x
  | n <= 0      = []
  | otherwise = x : replicate (n - 1) x
```

What to do with the `Int` for vectors?

```
vreplicate :: a -> Vec a n
```

won't work.

Generalizing `replicate` – using a class

```
class VReplicate (n :: Nat) where  
  vreplicate :: a -> Vec a n
```

Generalizing `replicate` – using a class

```
class VReplicate (n :: Nat) where  
  vreplicate :: a -> Vec a n
```

```
instance VReplicate Zero where  
  vreplicate _ = VNil
```

```
instance VReplicate n => VReplicate (Suc n) where  
  vreplicate x = x 'VCons' vreplicate x
```

Generalizing `replicate` – using a class

```
class VReplicate (n :: Nat) where  
  vreplicate :: a -> Vec a n
```

```
instance VReplicate Zero where  
  vreplicate _ = VNil
```

```
instance VReplicate n => VReplicate (Suc n) where  
  vreplicate x = x 'VCons' vreplicate x
```

Example:

```
type Three = Suc (Suc (Suc Zero))
```

```
GHCi> vreplicate 'x' :: Vec Char Three  
VCons 'x' (VCons 'x' (VCons 'x' VNil))
```

Singleton natural numbers

```
data SNat (n :: Nat) where
  SZero :: SNat Zero
  SSuc  :: SNatI n => SNat (Suc n)

class SNatI (n :: Nat) where
  sNat :: SNat n

instance SNatI Zero where
  sNat = SZero

instance SNatI n => SNatI (Suc n) where
  sNat = SSuc
```

Generalizing `replicate` – using singletons

```
vreplicate :: forall a n . SNatI n => a -> Vec a n
vreplicate x = case sNat :: SNat n of
  SZero -> VNil
  SSuc   -> x 'VCons' vreplicate x
```

This needs “scoped type variables”.

Generalizing `replicate` – using singletons

```
vreplicate :: forall a n . SNatI n => a -> Vec a n  
vreplicate x = case sNat :: SNat n of  
  SZero -> VNil  
  SSuc  -> x 'VCons' vreplicate x
```

This needs “scoped type variables”.

With singletons, we need only one class (`SNatI`) for all functions on natural numbers – but there is potentially more work being done at run-time.

Pointwise application

Where `vreplicate` fills the role of `pure`, we still need something corresponding to `(<*>)`:

```
vapply :: Vec (a -> b) n -> Vec a n -> Vec b n
vapply VNil          VNil          = VNil
vapply (f 'VCons' fs) (x 'VCons' xs) =
  f x 'VCons' vapply fs xs
```

Heterogeneous lists, n -ary products

Promoted lists

```
data [a] = [] | a : [a]
```

As a type:

```
[]  :: * -> *  
[]  :: forall (a :: *) . [a]  
(:) :: forall (a :: *) . a -> [a] -> [a]
```

As a kind:

```
[]    :: □ -> □  
'[]  :: forall (a :: □) . [a]  
'(:) :: forall (a :: □) . a -> [a] -> [a]
```

Both `'[]` and `'(:)` are **kind-polymorphic**.

Promoted lists – examples

```
GHCi> :kind True ': '[]  
True ': '[] :: [Bool]  
GHCi> :kind '[Zero, Three]  
'[Zero, Three] :: [Nat]  
GHCi> :kind '[Char, Bool, Int]  
'[Char, Bool, Int] :: [*]  
GHCi> :kind '[Maybe, [], IO]  
'[Maybe, [], IO] :: [* -> *]
```

Promoted lists – examples

```
GHCi> :kind True ': '[]  
True ': '[] :: [Bool]  
GHCi> :kind '[Zero, Three]  
'[Zero, Three] :: [Nat]  
GHCi> :kind '[Char, Bool, Int]  
'[Char, Bool, Int] :: [*]  
GHCi> :kind '[Maybe, [], IO]  
'[Maybe, [], IO] :: [* -> *]
```

Quotes for type-level lists are often not optional:

```
GHCi> :kind '[Bool]  
'[Bool] :: [*]  
GHCi> :kind [Bool]  
[Bool] :: *
```

Heterogeneous lists

```
data HList (xs :: [*]) where  
  HNil  :: HList '[]  
  HCons :: x -> HList xs -> HList (x ': xs)  
infixr 5 'HCons'
```

Heterogeneous lists

```
data HList (xs :: [*]) where  
  HNil  :: HList '[]  
  HCons :: x -> HList xs -> HList (x ': xs)  
infixr 5 'HCons'
```

Example:

```
GHCi> :type ('x' 'HCons' False 'HCons' HNil)  
'x' 'HCons' False 'HCons' HNil :: HList '[Char, Bool]
```


Generalizing further

We often need lists of **related** types.

```
data NP (f :: k -> *) (xs :: [k]) where  
  Nil  :: NP f '[]  
  (:*) :: f x -> NP f xs -> NP f (x ': xs)  
infixr 5 :*
```

Generalizing further

We often need lists of **related** types.

```
data NP (f :: k -> *) (xs :: [k]) where  
  Nil  :: NP f '[]  
  (:*) :: f x -> NP f xs -> NP f (x ': xs)  
infixr 5 :*
```

```
newtype I a    = I a  
newtype K a b = K a
```

```
NP I      xs ≈ HList xs  
NP (K a) xs ≈ Vec a (Length xs)
```

Collapsing an environment

```
hcollapse :: NP (K a) xs -> [a]
hcollapse Nil          = []
hcollapse (K x :* xs) = x : hcollapse xs
```

Collapsing an environment

```
hcollapse :: NP (K a) xs -> [a]
hcollapse Nil          = []
hcollapse (K x :* xs) = x : hcollapse xs
```

Next goal: generalize `vmap`, `vreplicate` and `vapply` to environments.

Higher-rank types

Compare:

```
Vec a n  
NP f xs
```

```
vmap :: (a -> b) -> Vec a n -> Vec b n
```

Generalizing `vmap`

Compare:

```
Vec a n  
NP  f xs
```

```
vmap :: (a -> b) -> Vec a n -> Vec b n
```

```
hmap :: ...      -> NP  f xs -> NP  g xs
```

Generalizing `vmap` – contd.

```
hmap m Nil      = Nil
hmap m (x :* xs) = m x :* hmap m xs
```

We apply `m` to all elements of the list.

Generalizing `vmap` – contd.

```
hmap m Nil      = Nil
hmap m (x :* xs) = m x :* hmap m xs
```

We apply `m` to all elements of the list.

```
m :: forall x . f x -> g x
```

Generalizing `vmap` – contd.

```
hmap m Nil      = Nil
hmap m (x :* xs) = m x :* hmap m xs
```

We apply `m` to all elements of the list.

```
m :: forall x . f x -> g x
```

```
hmap :: (forall x . f x -> g x) -> NP f xs -> NP g xs
```

Using `hmap` – an example

```
group :: NP I '[Char, Bool, Int]
group = I 'x' :* I False :* I 3 :* Nil

unI :: I a -> a
unI (I x) = x

example :: NP Maybe '[Char, Bool, Int]
example = hmap (Just . unI) group
```

```
GHCi> example
Just 'x' :* (Just False :* (Just 3 :* Nil))
```

Generalizing `vreplicate`

```
vreplicate :: SNatI n    => a                -> Vec a n
```

Generalizing `vreplicate`

```
vreplicate :: SNatI n    => a                -> Vec a n
```

```
hpure      :: SListI xs => (forall x . f x) -> NP  f xs
```

Singleton lists

```
data SList (xs :: [k]) where
  SNil  :: SList '[]
  SCons :: SListI xs => SList (x ': xs)

class SListI (xs :: [k]) where
  sList :: SList xs

instance SListI '[] where
  sList = SNil

instance SListI xs => SListI (x ': xs) where
  sList = SCons
```

We ignore the list elements (for now).

Completing `hpure`

```
hpure :: forall f xs . SListI xs  
      => (forall x . f x) -> NP f xs  
hpure x = case sList :: SList xs of  
  SNil   -> Nil  
  SCons  -> x :* hpure x
```

Examples:

```
GHCi> hpure Nothing :: NP Maybe '[Char, Bool, Int]  
Nothing :* (Nothing :* (Nothing :* Nil))  
GHCi> hpure (K 0)   :: NP (K Int) '[Char, Bool, Int]  
K 0 :* (K 0 :* (K 0 :* Nil))
```

Generalizing `vapply`

```
vapply :: Vec (a -> b) n -> Vec a n -> Vec b n  
hap    :: NP ... xs -> NP f xs -> NP g xs
```

What to do with the functions?

Generalizing `vapply`

```
vapply :: Vec (a -> b) n -> Vec a n -> Vec b n  
hap    :: NP ... xs -> NP f xs -> NP g xs
```

What to do with the functions?

```
NP (\x -> (f x -> g x)) xs
```

Generalizing `vapply`

```
vapply :: Vec (a -> b) n -> Vec a n -> Vec b n  
hap    :: NP ... xs -> NP f xs -> NP g xs
```

What to do with the functions?

```
NP (\x -> (f x -> g x)) xs
```

```
newtype (f -.-> g) x = Fn {apFn :: f x -> g x}  
infixr 1 -.->
```

Generalizing `vapply`

```
vapply :: Vec (a -> b) n -> Vec a n -> Vec b n  
hap    :: NP ... xs -> NP f xs -> NP g xs
```

What to do with the functions?

```
NP (\x -> (f x -> g x)) xs
```

```
newtype (f -.-> g) x = Fn {apFn :: f x -> g x}  
infixr 1 -.->
```

```
hap :: NP (f -.-> g) xs -> NP f xs -> NP g xs
```

Defining hap

```
hap :: NP (f -.-> g) xs -> NP f xs -> NP g xs
hap Nil Nil = Nil
hap (f :* fs) (x :* xs) = apFn f x :* hap fs xs
```

Using `hap` – an example

```
lists :: NP [] '[String, Int]
lists = ["foo", "bar", "baz"] :* [1..10] :* Nil

numbers :: NP (K Int) '[String, Int]
numbers = K 2 :* K 5 :* Nil

fn_2 :: (f a -> f' a -> f'' a)
      -> (f -.-> f' -.-> f'') a
fn_2 f = Fn (\x -> Fn (\y -> f x y))

take' :: (K Int -.-> [] -.-> []) a
take' = fn_2 (\(K n) xs -> take n xs)
```

```
GHCi> hpure take' 'hap' numbers 'hap' lists
["foo", "bar"] :* ([1, 2, 3, 4, 5] :* Nil)
```

Another look at `hmap`

```
hmap' :: SListI xs  
      => (forall a . f a -> g a)  
      -> NP f xs -> NP g xs  
hmap' f xs = hpure (Fn f) 'hap' xs
```

Another look at `hmap`

```
hmap' :: SListI xs  
      => (forall a . f a -> g a)  
      -> NP f xs -> NP g xs  
hmap' f xs = hpure (Fn f) 'hap' xs
```

```
hzipWith :: SListI xs  
         => (forall a . f a -> g a -> h a)  
         -> NP f xs -> NP g xs -> NP h xs  
hzipWith f xs ys = hpure (fn_2 f) 'hap' xs 'hap' ys
```

Abstracting from classes, type functions

Mapping constrained functions?

```
hmap (K . show . unI) group
```

fails, because

```
K . show . unI :: forall x . Show x => I x -> K String x
```

does not match

```
forall x . f x -> g x
```

Constraints are types of kind **Constraint**

```
GHCi> :kind Eq
Eq :: * -> Constraint
GHCi> :kind Functor
Functor :: (* -> *) -> Constraint
GHCi> :kind MonadReader
MonadReader :: * -> (* -> *) -> Constraint
```

Constraints are types of kind **Constraint**

```
GHCi> :kind Eq
Eq :: * -> Constraint
GHCi> :kind Functor
Functor :: (* -> *) -> Constraint
GHCi> :kind MonadReader
MonadReader :: * -> (* -> *) -> Constraint
```

Overloaded tuple syntax:

```
type NoConstraint          = (() :: Constraint)
type SomeConstraints a     = (Eq a, Show a)
type MoreConstraints f a = (Monad f, SomeConstraints a)
```

The `All` type family

```
type family All (c :: k -> Constraint)
               (xs :: [k])
               :: Constraint where
  All c '[]      = ()
  All c (x ' : xs) = (c x, All c xs)
```

The `All` type family

```
type family All (c :: k -> Constraint)
               (xs :: [k])
               :: Constraint where
  All c '[]      = ()
  All c (x ' : xs) = (c x, All c xs)
```

Example:

```
GHCi> :kind! All Eq '[Int, Bool]
All Eq '[Int, Bool] :: Constraint
= (Eq Int, (Eq Bool, ()))
```

(Constraints are flattened.)

Trying to define `hcpure`

We want:

```
hcpure  :: SListI xs          => (forall a .      f a) -> NP f xs  
hcpure :: (SListI xs, All c xs) => (forall a . c a => f a) -> NP f xs
```

Then:

```
hcmmap :: (SListI xs, All c xs)  
        => (forall a . c a => f a -> g a) -> NP f xs -> NP g xs  
hcmmap f xs = hcpure (Fn f) 'hap' xs
```

However, this does not work.

Limitations in GHC's type inference

Assume:

```
hcpure :: (SListI xs, All c xs) => (forall a . c a => f a) .  
hcpure = undefined
```

Then

```
minBound    :: Bounded a => a  
I minBound :: Bounded a => I a
```

```
GHCi> hcpure (I minBound) :: NP I '[Char, Bool]
```

is a type error.

Proxies

```
data Proxy (a :: k) = Proxy
```

Examples:

```
pBounded :: Proxy Bounded
```

```
pBounded = Proxy
```

```
pShow :: Proxy Show
```

```
pShow = Proxy
```


Using proxies to define `hcpure`

```
hcpure :: forall c f xs . (SListI xs, All c xs)
      => Proxy c -> (forall a . c a => f a) -> NP f xs
hcpure p x = case sList :: SList xs of
  SNil   -> Nil
  SCons  -> x :* hcpure p x
```

Using proxies to define `hcpure`

```
hcpure :: forall c f xs . (SListI xs, All c xs)
      => Proxy c -> (forall a . c a => f a) -> NP f xs
hcpure p x = case sList :: SList xs of
  SNil   -> Nil
  SCons  -> x :* hcpure p x
```

Example:

```
GHCI> hcpure pBounded (I minBound) :: NP I '[Char, Bool]
I '\NUL' :* (I False :* Nil)
GHCI> hcpure pShow (Fn (K . show . unI)) 'hap' group
K "'x'" :* (K "False" :* (K "3" :* Nil))
```

```
hmap :: (SListI xs, All c xs)
      => Proxy c
      -> (forall a . c a => f a -> g a)
      -> NP f xs -> NP g xs
hmap p f xs = hcpure p (Fn f) 'hap' xs
```

Summary

Lots of type system extensions and concepts in action:

- ▶ GADTs,
- ▶ promoted lists,
- ▶ singleton lists,
- ▶ higher-rank types,
- ▶ constraint kinds,
- ▶ proxies.

Summary

Lots of type system extensions and concepts in action:

- ▶ GADTs,
- ▶ promoted lists,
- ▶ singleton lists,
- ▶ higher-rank types,
- ▶ constraint kinds,
- ▶ proxies.

The type `NP` and a number of useful functions:

- ▶ `hpure`, `hap`, `hmap`, `hzipWith`
- ▶ `hcpure`, `hcmmap`, `(hzipWith)`
- ▶ `hcollapse`

With `NP`, we can already express one-constructor datatypes nicely.

We'll add `NS` to express the choice between constructors.

Exercises

1. Define `heq :: ... => NP I xs -> NP I xs -> Bool` via pattern matching.
2. Define `hzipWith`, i.e., `hzipWith` with a constrained function.
3. Define `heq` using `hzipWith`.
4. Can you generalize
`heq :: ... => NP f xs -> NP f xs -> Bool` ?
5. Define
`hsequence :: Applicative f => NP f xs -> f (NP I xs)`.
6. Try to generalize
`foldr :: (a -> r -> r) -> r -> [a] -> r` from lists to `Vec` and `NP`. Try to redefine e.g. `hmap` using the generalized `foldr`.