Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Generic Programming for Indexed Datatypes

José Pedro Magalhães
joint work with Johan Jeuring

Dept. of Information and Computing Sciences, Utrecht University
`http://dreixel.net`

September 18, 2011

The problem

```
data Ze
data Su ν

data Vec α ν where
  Nil  ::                      Vec α Ze
  Cons :: α → Vec α ν → Vec α (Su ν)

deriving instance (Read α) ⇒ Read (Vec α ν)
```

The problem

```
data Ze
data Su ν

data Vec α ν where
  Nil  ::                        Vec α Ze
  Cons :: α → Vec α ν → Vec α (Su ν)

deriving instance (Read α) ⇒ Read (Vec α ν)
```

Couldn't match type Ze with Su ν
Expected type: Vec α ν
Actual type: Vec α Ze
In the expression: return Nil
When typechecking the code for GHC.Read.readPrec in a standalone derived instance for Read (Vec α ν)

Outline

Generic view

Type-level datatype representations:

$$\begin{aligned}
&\textbf{data}\ \alpha + \beta = \mathsf{L}\ \alpha \mid \mathsf{R}\ \beta \\
&\textbf{data}\ \alpha \times \beta = \alpha \times \beta \\
&\textbf{data}\ \mathsf{C}\ \gamma\ \alpha = \mathsf{C}\ \alpha \\
&\textbf{data}\ \mathsf{U} \quad\quad = \mathsf{U} \\
&\textbf{data}\ \mathsf{Rec}\ \alpha = \mathsf{Rec}\ \alpha
\end{aligned}$$

Generic view

Type-level datatype representations:

```
data α + β = L α | R β
data α × β = α × β
data C γ α = C α
data U     = U
data Rec α = Rec α
```

Type class mediating the conversion between types and their generic representation:

```
class Representable α where
  type Rep α
  to   :: Rep α → α
  from :: α     → Rep α
```

Example representation

Generic representation of the standard Haskell lists:

```
instance Representable [α] where
  type Rep [α] = C List_[] U + C List_: (Rec α × Rec [α])

  from []       = L (C U)
  from (a : as) = R (C (Rec a × Rec as))

  to (L (C U))                = []
  to (R (C (Rec a × Rec as))) = (a : as)
```

Example representation

Generic representation of the standard Haskell lists:

```
instance Representable [α] where
  type Rep [α] = C List[] U + C List: (Rec α × Rec [α])

  from []       = L (C U)
  from (a : as) = R (C (Rec a × Rec as))

  to (L (C U))                 = []
  to (R (C (Rec a × Rec as))) = (a : as)

data List[]
instance Constructor List[] where conName _ = "[]"

data List:
instance Constructor List: where
  conName _ = ":"
  conFixity _ = Infix RightAssociative 5
```

A generic consumer I

Equality for representation types:

```
class GEq' α where
  geq' :: α → α → Bool
```

A generic consumer I

Equality for representation types:

```
class GEq' α where
  geq' :: α → α → Bool

instance GEq' U where geq' U U = True
```

A generic consumer I

Equality for representation types:

```
class GEq' α where
  geq' :: α → α → Bool

instance GEq' U where geq' U U = True

instance (GEq' α, GEq' β) ⇒ GEq' (α + β) where
  geq' (L a) (L b) = geq' a b
  geq' (R a) (R b) = geq' a b
  geq' _     _     = False
```

A generic consumer I

Equality for representation types:

```
class GEq' α where
  geq' :: α → α → Bool

instance GEq' U where geq' U U = True

instance (GEq' α, GEq' β) ⇒ GEq' (α + β) where
  geq' (L a) (L b) = geq' a b
  geq' (R a) (R b) = geq' a b
  geq' _     _     = False

instance (GEq' α, GEq' β) ⇒ GEq' (α × β) where
  geq' (a × b) (a' × b') = geq' a a' ∧ geq' b b'
```

A generic consumer I

Equality for representation types:

```
class GEq' α where
  geq' :: α → α → Bool

instance GEq' U where geq' U U = True

instance (GEq' α, GEq' β) ⇒ GEq' (α + β) where
  geq' (L a) (L b) = geq' a b
  geq' (R a) (R b) = geq' a b
  geq' _     _     = False

instance (GEq' α, GEq' β) ⇒ GEq' (α × β) where
  geq' (a × b) (a' × b') = geq' a a' ∧ geq' b b'

instance (GEq' α) ⇒ GEq' (C γ α) where
  geq' (C a) (C b) = geq' a b
```

A generic consumer I

Equality for representation types:

```
class GEq' α where
  geq' :: α → α → Bool

instance GEq' U where geq' U U = True

instance (GEq' α, GEq' β) ⇒ GEq' (α + β) where
  geq' (L a) (L b) = geq' a b
  geq' (R a) (R b) = geq' a b
  geq' _     _     = False

instance (GEq' α, GEq' β) ⇒ GEq' (α × β) where
  geq' (a × b) (a' × b') = geq' a a' ∧ geq' b b'

instance (GEq' α) ⇒ GEq' (C γ α) where
  geq' (C a) (C b) = geq' a b

instance (GEq α) ⇒ GEq' (Rec α) where
  geq' (Rec a) (Rec b) = geq a b
```

A generic consumer II

Equality for representable types and adhoc equality:

```
class GEq α where
  geq :: α → α → Bool
```

A generic consumer II

Equality for representable types and adhoc equality:

```
class GEq α where
  geq :: α → α → Bool

geqDefault :: (Representable α, GEq' (Rep α))
           ⇒ α → α → Bool
geqDefault x y = geq' (from x) (from y)
```

A generic consumer II

Equality for representable types and adhoc equality:

```
class GEq α where
  geq :: α → α → Bool

geqDefault :: (Representable α, GEq' (Rep α))
           ⇒ α → α → Bool
geqDefault x y = geq' (from x) (from y)

instance (GEq α) ⇒ GEq [α] where
  geq = geqDefault
```

A generic consumer II

Equality for representable types and adhoc equality:

```
class GEq α where
  geq :: α → α → Bool

geqDefault :: (Representable α, GEq' (Rep α))
           ⇒ α → α → Bool
geqDefault x y = geq' (from x) (from y)

instance (GEq α) ⇒ GEq [α] where
  geq = geqDefault

instance GEq Char where geq = (≡)
instance GEq Int  where geq = (≡)
```

A generic producer I

Enumeration for representation types:

```
class GEnum' α where genum' :: [α]
```

A generic producer I

Enumeration for representation types:

```
class GEnum' α where genum' :: [α]

instance GEnum' U where genum' = [U]
```

A generic producer I

Enumeration for representation types:

```
class GEnum' α where genum' :: [α]

instance GEnum' U where genum' = [U]

instance (GEnum' α, GEnum' β) ⇒ GEnum' (α + β) where
  genum' = map L genum' ||| map R genum'
```

A generic producer I

Enumeration for representation types:

```
class GEnum' α where genum' :: [α]

instance GEnum' U where genum' = [U]

instance (GEnum' α, GEnum' β) ⇒ GEnum' (α + β) where
  genum' = map L genum' ||| map R genum'

instance (GEnum' α, GEnum' β) ⇒ GEnum' (α × β) where
  genum' = diag [[x × y | x ← genum'] | y ← genum']
```

A generic producer I

Enumeration for representation types:

```
class GEnum' α where genum' :: [α]

instance GEnum' U where genum' = [U]

instance (GEnum' α, GEnum' β) ⇒ GEnum' (α + β) where
  genum' = map L genum' ||| map R genum'

instance (GEnum' α, GEnum' β) ⇒ GEnum' (α × β) where
  genum' = diag [[x × y | x ← genum'] | y ← genum']

instance (GEnum' α) ⇒ GEnum' (C γ α) where
  genum' = map C genum'
```

A generic producer I

Enumeration for representation types:

```
class GEnum' α where genum' :: [α]

instance GEnum' U where genum' = [U]

instance (GEnum' α, GEnum' β) ⇒ GEnum' (α + β) where
  genum' = map L genum' ||| map R genum'

instance (GEnum' α, GEnum' β) ⇒ GEnum' (α × β) where
  genum' = diag [[x × y | x ← genum'] | y ← genum']

instance (GEnum' α) ⇒ GEnum' (C γ α) where
  genum' = map C genum'

instance (GEnum α) ⇒ GEnum' (Rec α) where
  genum' = map Rec genum
```

A generic producer II

Enumeration for representable types and adhoc enumeration:

```
class GEnum α where
  genum :: [α]
```

A generic producer II

Enumeration for representable types and adhoc enumeration:

```
class GEnum α where
  genum :: [α]

genumDefault :: (Representable α, GEnum′ (Rep α)) ⇒ [α]
genumDefault = map to genum′
```

A generic producer II

Enumeration for representable types and adhoc enumeration:

```
class GEnum α where
  genum :: [α]

genumDefault :: (Representable α, GEnum′ (Rep α)) ⇒ [α]
genumDefault = map to genum′

instance GEnum α ⇒ GEnum [α] where
  genum = genumDefault
```

A generic producer II

Enumeration for representable types and adhoc enumeration:

```
class GEnum α where
  genum :: [α]

genumDefault :: (Representable α, GEnum′ (Rep α)) ⇒ [α]
genumDefault = map to genum′

instance GEnum α ⇒ GEnum [α] where
  genum = genumDefault

instance GEnum Int  where
  genum = [0 . .] ||| map negate [1 . .]
instance GEnum Bool where
  genum = [True, False]
```

It all works

Testing generic equality:

geq `"ab"` `"ab"` $\rightsquigarrow$ True

geq `"ab"` `"abc"` $\rightsquigarrow$ False

Testing generic enumeration:

genum :: [Bool] $\rightsquigarrow$ [True, False]

take 6 (genum :: [[Int]]) $\rightsquigarrow$ $[[], [0], [-1], [0, 0], [1], [-1, 0]]$

Outline

Indexed datatypes: Vec

```
data Ze
data Su ν

data Vec α ν where
  Nil  ::                          Vec α Ze
  Cons :: α → Vec α ν → Vec α (Su ν)

type 0 = Ze
type 1 = Su 0
type 2 = Su 1

exampleVec :: Vec Char 2
exampleVec = Cons 'p' (Cons 'q' Nil)
```

Indexed datatypes: Term

```
data Term α where
  Lit    :: Int                          → Term Int
  IsZero :: Term Int                     → Term Bool
  Pair   :: Term α → Term β              → Term (α, β)
  If     :: Term Bool → Term α → Term α  → Term α
```

Indexed datatypes: Term

```
data Term α where
  Lit    :: Int                          → Term Int
  IsZero :: Term Int                     → Term Bool
  Pair   :: Term α → Term β              → Term (α, β)
  If     :: Term Bool → Term α → Term α  → Term α
```

```
eval :: Term α → α
eval (Lit i)      = i
eval (IsZero t)   = eval t ≡ 0
eval (Pair a b)   = (eval a, eval b)
eval (If p a b)   = if eval p then eval a else eval b
```

Functions on indexed datatypes I

We cannot take the definition of enumeration on lists . . .

$$\begin{aligned}&\mathsf{enum}_{[]} :: [\alpha] \to [[\alpha]]\\&\mathsf{enum}_{[]}\ \mathsf{ea} = [\,] : [\mathsf{x} : \mathsf{xs} \mid \mathsf{x} \leftarrow \mathsf{ea}, \mathsf{xs} \leftarrow \mathsf{enum}_{[]}\ \mathsf{ea}]\end{aligned}$$

Functions on indexed datatypes I

We cannot take the definition of enumeration on lists . . .

$$\begin{aligned}&\mathsf{enum}_{[]} :: [\alpha] \to [[\alpha]]\\&\mathsf{enum}_{[]}\ \mathsf{ea} = [\,] : [\mathsf{x} : \mathsf{xs} \mid \mathsf{x} \leftarrow \mathsf{ea}, \mathsf{xs} \leftarrow \mathsf{enum}_{[]}\ \mathsf{ea}]\end{aligned}$$

. . . and trivially convert it to vectors:

$$\begin{aligned}&\mathsf{enum}_{\mathsf{Vec}} :: [\alpha] \to [\mathsf{Vec}\ \alpha\ \nu]\\&\mathsf{enum}_{\mathsf{Vec}}\ \mathsf{ea} = \mathsf{Nil} : [\mathsf{Cons}\ \mathsf{x}\ \mathsf{xs} \mid \mathsf{x} \leftarrow \mathsf{ea}, \mathsf{xs} \leftarrow \mathsf{enum}_{\mathsf{Vec}}\ \mathsf{ea}]\end{aligned}$$

Functions on indexed datatypes I

We cannot take the definition of enumeration on lists . . .

$$\begin{aligned}&\mathsf{enum}_{[]} :: [\alpha] \to [[\alpha]]\\&\mathsf{enum}_{[]}\ \mathsf{ea} = [\,] : [\mathsf{x} : \mathsf{xs} \mid \mathsf{x} \leftarrow \mathsf{ea}, \mathsf{xs} \leftarrow \mathsf{enum}_{[]}\ \mathsf{ea}]\end{aligned}$$

. . . and trivially convert it to vectors:

$$\begin{aligned}&\mathsf{enum}_{\mathsf{Vec}} :: [\alpha] \to [\mathsf{Vec}\ \alpha\ \nu]\\&\mathsf{enum}_{\mathsf{Vec}}\ \mathsf{ea} = \mathsf{Nil} : [\mathsf{Cons}\ \mathsf{x}\ \mathsf{xs} \mid \mathsf{x} \leftarrow \mathsf{ea}, \mathsf{xs} \leftarrow \mathsf{enum}_{\mathsf{Vec}}\ \mathsf{ea}]\end{aligned}$$

Couldn't match type Ze with Su ν
Expected type: Vec α ν
Actual type: Vec α Ze
In the first argument of (:), namely Nil

Functions on indexed datatypes II

Instead:

```
instance GEnum (Vec α Ze) where
  genum = [Nil]

instance  ( GEnum α, GEnum (Vec α ν))
          ⇒ GEnum (Vec α (Su ν)) where
  genum = [Cons a t | a ← genum, t ← genum]
```

Now we see why GHC has trouble deriving Read instances for indexed datatypes.

Outline

Indexed datatypes in primitive terms

Constructor constraints, existential quantification, and type equalities:

$$\begin{aligned}\textbf{data}\ \mathsf{Vec}\ \alpha\ \nu = \quad & \nu{\sim}\mathsf{Ze} \Rightarrow \mathsf{Nil} \\ \mid\ & \forall\mu.\nu{\sim}\mathsf{Su}\ \mu \Rightarrow \mathsf{Cons}\ \alpha\ (\mathsf{Vec}\ \alpha\ \mu)\end{aligned}$$

Indexed datatypes in primitive terms

Constructor constraints, existential quantification, and type equalities:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Vec}\ \alpha\ \nu = \quad \nu{\sim}\mathsf{Ze} \quad \Rightarrow \mathsf{Nil} \\
\qquad\qquad\qquad\qquad \mid\ \forall \mu.\nu{\sim}\mathsf{Su}\ \mu \Rightarrow \mathsf{Cons}\ \alpha\ (\mathsf{Vec}\ \alpha\ \mu)
\end{array}
$$

$$
\begin{array}{lll}
\textbf{data}\ \mathsf{Term}\ \alpha = & & \\
\qquad\qquad \alpha{\sim}\mathsf{Int} & \Rightarrow \mathsf{Lit} & \mathsf{Int} \\
\mid \qquad \alpha{\sim}\mathsf{Bool} & \Rightarrow \mathsf{IsZero} & (\mathsf{Term}\ \mathsf{Int}) \\
\mid \forall \beta\ \gamma.\alpha{\sim}(\beta,\gamma) & \Rightarrow \mathsf{Pair} & (\mathsf{Term}\ \beta)\ (\mathsf{Term}\ \gamma) \\
\mid & \quad\ \mathsf{If} & (\mathsf{Term}\ \mathsf{Bool})\ (\mathsf{Term}\ \alpha)\ (\mathsf{Term}\ \alpha)
\end{array}
$$

Encoding type equality

A general type equality $\alpha \sim \beta$ can be encoded in a simple GADT:

data $\alpha \simeq \beta$ **where**
  Refl :: $\alpha \simeq \alpha$

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Encoding type equality

A general type equality $\alpha \sim \beta$ can be encoded in a simple GADT:

$$\begin{aligned}&\textbf{data}\ \alpha \simeq \beta\ \textbf{where}\\ &\quad \mathsf{Refl} :: \alpha \simeq \alpha\end{aligned}$$

We embed equalities directly into the representation for constructors:

$$\begin{aligned}&\textbf{data}\ \mathsf{C}_{\mathsf{Eq}}\ \gamma\ \phi\ \psi\ \alpha\ \textbf{where}\\ &\quad \mathsf{C}_{\mathsf{Eq}} :: \alpha \to \mathsf{C}_{\mathsf{Eq}}\ \gamma\ \phi\ \phi\ \alpha\end{aligned}$$

Standard constructors have no equality constraints:

$$\textbf{type}\ \mathsf{C}\ \gamma\ \alpha = \mathsf{C}_{\mathsf{Eq}}\ \gamma\ ()\ ()\ \alpha$$

Existentially-quantified indices

Recall the shape of the Cons constructor of the Vec datatype:

$$\forall \mu.\nu{\sim}\mathsf{Su}\ \mu \Rightarrow \mathsf{Cons}\ \alpha\ (\mathsf{Vec}\ \alpha\ \mu)$$

Existentially-quantified indices

Recall the shape of the Cons constructor of the Vec datatype:

$$\forall \mu.\nu{\sim}\mathsf{Su}\ \mu \Rightarrow \mathsf{Cons}\ \alpha\ (\mathsf{Vec}\ \alpha\ \mu)$$

Trying to introduce variables in the representation:

$$\textbf{type instance}\ \mathsf{Rep}\ (\mathsf{Vec}\ \alpha\ \nu) = \forall \mu.\mathsf{C}_{\mathsf{Eq}}\ \mathsf{Vec}_{\mathsf{Cons}}\ \nu\ (\mathsf{Su}\ \mu)\ \ldots$$

Existentially-quantified indices

Recall the shape of the Cons constructor of the Vec datatype:

$$\forall \mu.\nu{\sim}\mathsf{Su}\ \mu \Rightarrow \mathsf{Cons}\ \alpha\ (\mathsf{Vec}\ \alpha\ \mu)$$

Trying to introduce variables in the representation:

$$\textbf{type instance}\ \mathsf{Rep}\ (\mathsf{Vec}\ \alpha\ \nu) = \forall \mu.\mathsf{C}_{\mathsf{Eq}}\ \mathsf{Vec}_{\mathsf{Cons}}\ \nu\ (\mathsf{Su}\ \mu)\ \ldots$$

Not accepted by GHC.

Existentially-quantified indices

Recall the shape of the Cons constructor of the Vec datatype:

$$\forall \mu.\nu{\sim}\mathsf{Su}\ \mu \Rightarrow \mathsf{Cons}\ \alpha\ (\mathsf{Vec}\ \alpha\ \mu)$$

Trying to introduce variables in the representation:

$$\textbf{type instance}\ \mathsf{Rep}\ (\mathsf{Vec}\ \alpha\ \nu) = \forall \mu.\mathsf{C}_{\mathsf{Eq}}\ \mathsf{Vec}_{\mathsf{Cons}}\ \nu\ (\mathsf{Su}\ \mu)\ \ldots$$

Not accepted by GHC. With data families:

$$\textbf{data instance}\ \mathsf{Rep}\ (\mathsf{Vec}\ \alpha\ \nu) = \forall \mu.\mathsf{Rep}_{\mathsf{Vec}}\ (\mathsf{C}_{\mathsf{Eq}}\ \mathsf{Vec}_{\mathsf{Cons}}\ \nu\ (\mathsf{Su}\ \mu)\ \ldots)$$

Existentially-quantified indices

Recall the shape of the Cons constructor of the Vec datatype:

$$\forall \mu.\nu{\sim}\mathsf{Su}\ \mu \Rightarrow \mathsf{Cons}\ \alpha\ (\mathsf{Vec}\ \alpha\ \mu)$$

Trying to introduce variables in the representation:

$$\textbf{type instance}\ \mathsf{Rep}\ (\mathsf{Vec}\ \alpha\ \nu) = \forall \mu.\mathsf{C}_{\mathsf{Eq}}\ \mathsf{Vec}_{\mathsf{Cons}}\ \nu\ (\mathsf{Su}\ \mu)\ \ldots$$

Not accepted by GHC. With data families:

$$\textbf{data instance}\ \mathsf{Rep}\ (\mathsf{Vec}\ \alpha\ \nu) = \forall \mu.\mathsf{Rep}_{\mathsf{Vec}}\ (\mathsf{C}_{\mathsf{Eq}}\ \mathsf{Vec}_{\mathsf{Cons}}\ \nu\ (\mathsf{Su}\ \mu)\ \ldots)$$

Introduces new, type-specific constructors: unsatisfactory.

Faking existentials

Introduce a type family to encode existentials:

```
type family Ex ν
```

Use it in the representation type:

```
instance Representable (Vec α ν) where
  type Rep (Vec α ν) = C_Eq Vec_Nil  ν Ze U
                     + C_Eq Vec_Cons ν (Su (Ex ν))
                            (Rec α × Rec (Vec α (Ex ν)))
  ...
```

Provide mobility rules:

```
type instance Ex (Su μ) = μ
```

Representing Term

First, generalise Ex:

type family X γ ι α

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Representing Term

First, generalise Ex:

type family X γ ι α

Representation for Term:

type $\mathsf{Rep}_{\mathsf{Term}}$ α =
C_{Eq} $\mathsf{Term}_{\mathsf{Lit}}$ α Int (Rec Int)
$+$ C_{Eq} $\mathsf{Term}_{\mathsf{IsZero}}$ α Bool (Rec (Term Int))
$+$ C_{Eq} $\mathsf{Term}_{\mathsf{Pair}}$ α (X $\mathsf{Term}_{\mathsf{Pair}}$ 0 α, X $\mathsf{Term}_{\mathsf{Pair}}$ 1 α)
(Rec (Term (X $\mathsf{Term}_{\mathsf{Pair}}$ 0 α))
$\times$ Rec (Term (X $\mathsf{Term}_{\mathsf{Pair}}$ 1 α)))
$+$ C $\mathsf{Term}_{\mathsf{If}}$
(Rec (Term Bool) $\times$ Rec (Term α) $\times$ Rec (Term α))

Mobility rules:

type instance X $\mathsf{Term}_{\mathsf{Pair}}$ 0 $(\beta, \gamma) = \beta$
type instance X $\mathsf{Term}_{\mathsf{Pair}}$ 1 $(\beta, \gamma) = \gamma$

Outline

Instantiating generic consumers

Generic consumers, like equality, are not problematic:

```
instance (GEq α) ⇒ GEq (Vec α ν) where
  geq = geqDefault

instance GEq (Term α) where
  geq = geqDefault
```

Instantiating generic producers

Producers are more complicated:

```
instance (GEnum α) ⇒ GEnum (Vec α Ze) where
  genum = genumDefault
instance  ( GEnum α, GEnum (Vec α ν))
          ⇒ GEnum (Vec α (Su ν)) where
  genum = genumDefault
```

Instantiating generic producers

Producers are more complicated:

```
instance (GEnum α) ⇒ GEnum (Vec α Ze) where
  genum = genumDefault
instance  ( GEnum α, GEnum (Vec α ν))
          ⇒ GEnum (Vec α (Su ν)) where
  genum = genumDefault

instance       GEnum (Term Int)     where
  genum = genumDefault
instance       GEnum (Term Bool)    where
  genum = genumDefault
instance (GEnum (Term α), GEnum (Term β))
             ⇒ GEnum (Term (α, β)) where
  genum = genumDefault
```

Outline

Conclusion

What we have done:

- Automatic derivation of generic representation for indexed datatypes
- Automatic instantiation of generic functions to indexed datatypes

Thus enabling generic programming for a whole new class of datatypes.

Future work

- Unrestricted indices:

$$\begin{aligned}\textbf{data}\ \mathsf{Tag}\ \alpha = \quad & \alpha{\sim}\mathsf{Int} \Rightarrow \mathsf{TagI} \\ \mid\ & \forall\beta.\alpha{\sim}\mathsf{Bool} \Rightarrow \mathsf{TagB}\ (\mathsf{Tag}\ \beta)\end{aligned}$$

Future work

- Unrestricted indices:

$$\begin{aligned}\textbf{data}\ \mathsf{Tag}\ \alpha = \quad & \alpha{\sim}\mathsf{Int} \Rightarrow \mathsf{TagI} \\ \mid\ & \forall\beta.\alpha{\sim}\mathsf{Bool} \Rightarrow \mathsf{TagB}\ (\mathsf{Tag}\ \beta)\end{aligned}$$

- Explore the benefits of user-defined kinds

Future work

- Unrestricted indices:

$$\begin{aligned}\textbf{data}\ \mathsf{Tag}\ \alpha = \quad & \alpha{\sim}\mathsf{Int} \quad \Rightarrow \mathsf{TagI} \\ \mid\ & \forall\beta.\alpha{\sim}\mathsf{Bool} \Rightarrow \mathsf{TagB}\ (\mathsf{Tag}\ \beta)\end{aligned}$$

- Explore the benefits of user-defined kinds
- Existentials as data

$$\textbf{data}\ \mathsf{Dynamic} = \forall\alpha.\mathsf{Typeable}\ \alpha \Rightarrow \mathsf{Dyn}\ \alpha$$