Haskell: A Universe of Types

Tom Schrijvers

Leuven Haskell
User Group

Data Genericity	Recursion Schemes	GADTs	DSLs
Expression Problem	Monads	Type Families	Type Classes
Rank-N Poly-morphism	Effect Handlers	Free Theorems	...

Data Genericity	Recursion Schemes	GADTs	DSLs
Expression Problem	Monads	Type Families	Type Classes
Rank-N Poly-morphism	Effect Handlers	Free Theorems	...

The Structure of Types

Quick Refresher: Algebraic Datatypes

```
data Tree a
  = Empty
  | Leaf a
  | Node (Tree a) (Tree a)
```

Quick Refresher: Algebraic Datatypes

type constructor

```
data Tree a
   = Empty
   | Leaf a
   | Node (Tree a) (Tree a)
```

Quick Refresher: Algebraic Datatypes

type constructor

type parameter

```
data Tree a
   = Empty
   | Leaf a
   | Node (Tree a) (Tree a)
```

Quick Refresher: Algebraic Datatypes

type constructor

type parameter

```
data Tree a
  = Empty
  | Leaf a
  | Node (Tree a) (Tree a)
```

alternatives

Quick Refresher: Algebraic Datatypes

type constructor

type parameter

```
data Tree a
  = Empty
  | Leaf a
  | Node (Tree a) (Tree a)
```

alternatives

(data) constructor

Quick Refresher: Algebraic Datatypes

type constructor

type parameter

```
data Tree a
  = Empty
  | Leaf a
  | Node (Tree a) (Tree a)
```

alternatives

fields

(data) constructor

Today we learn…

```
data Tree a
  = Empty
  | Leaf a
  | Node (Tree a) (Tree a)
  deriving Eq
```

…what happens when you derive a type class.

Modelling Exercise

Create a datatype for people that have a name and an age.

Solutions

Solution 1

```
data Person = P String Int
```

Solutions

Solution 1

```
data Person = P String Int
```

Solution 2

```
data Human = H Int String
```

Solutions

Solution 1

```
data Person = P String Int
```

Solution 2

```
data Human = H Int String
```

are these the same?

The same in Haskell?

```
data Person = P String Int
```

```
data Human = H Int String
```

```
p :: Person
p = H 42 “Haskell Curry”
```

The same in Haskell?

```
data Person = P String Int
```

```
data Human = H Int String
```

```
p :: Person
p = H 42 “Haskell Curry”
```

```
Couldn't match expected type ‘Person’
with actual type ‘Human’
```

What kind of sameness?

What kind of sameness?

Same

Structure

What kind of sameness?

Same Structure

ισος
μορφη

What kind of sameness?

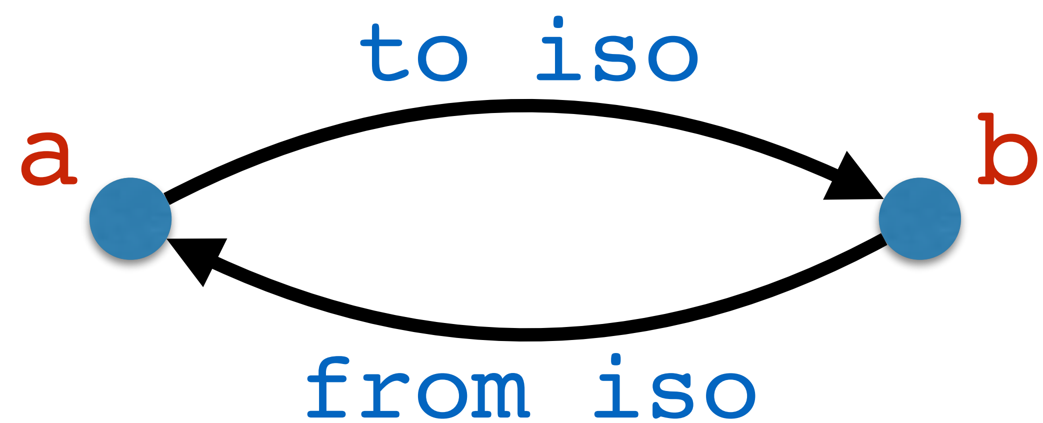
Same Structure

Iso-morphism

ισος μορφη

Isomorphism

```haskell
data a <~> b = Iso { from :: a -> b
                   , to   :: b -> a }
```




```haskell
from iso . to iso = id
to iso . from iso = id
```

Isomorphism

```haskell
data a <~> b = Iso { to   :: a -> b
                   , from :: b -> a }
```

```haskell
isoPH :: Person <~> Human
```

Isomorphism

```
data a <~> b = Iso { to   :: a -> b
                   , from :: b -> a }


isoPH :: Person <~> Human
isoPH = Iso (\(P n a) -> H a n)
            (\(H a n) -> P n a)
```

Isomorphism

```
data a <~> b = Iso { to   :: a -> b
                   , from :: b -> a }


isoPH :: Person <~> Human
isoPH = Iso (\(P n a) -> H a n)
            (\(H a n) -> P n a)

p :: Person
p = from isoPH (H 39 “Haskell Curry”)
```

Isomorphism

```
data a <~> b = Iso { to   :: a -> b
                   , from :: b -> a }


isoPH :: Person <~> Human
isoPH = Iso (\(P n a) -> H a n)
            (\(H a n) -> P n a)

p :: Person
p = from isoPH (H 39 “Haskell Curry”)
```

coercion

The Algebra of Types

The algebra of natural numbers

Guiseppe
Peano

$$
\begin{aligned}
0 &: \mathbb{N} \\
1 &: \mathbb{N} \\
+ &: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
* &: \mathbb{N} \to \mathbb{N} \to \mathbb{N}
\end{aligned}
$$

Numbers vs. Types

```
0 : N
```

Numbers vs. Types

```
0 : N
data Zero
```

— no constructors

Numbers vs. Types

```
0 : N
data Zero
```

— no constructors

```
1 : N
```

Numbers vs. Types

```
0 : N
data Zero
```
— no constructors

```
1 : N
data One = One
```

Numbers vs. Types

```
0 : N
data Zero        — no constructors

1 : N
data One = One

+ : N → N → N
```

Numbers vs. Types

```
0 : N
data Zero        -- no constructors

1 : N
data One = One

+ : N → N → N
data x + y = Inl x | Inr y
```

Numbers vs. Types

```
0 : N
data Zero        -- no constructors

1 : N
data One = One

+ : N → N → N
data x + y = Inl x | Inr y

* : N → N → N
```

Numbers vs. Types

```
0 : N
data Zero          -- no constructors

1 : N
data One = One

+ : N → N → N
data x + y = Inl x | Inr y

* : N → N → N
data x * y = x :*: y
```

Arithmetic Laws

Commutativity of +

```
x + y = y + x
```

Arithmetic Laws

Commutativity of +

$$x + y = y + x$$

```
commPlus :: (x + y) <~> (y + x)
```

Arithmetic Laws

Commutativity of +

```
x + y = y + x
```

```
commPlus :: (x + y) <~> (y + x)
commPlus = Iso f f
  where
    f :: (a + b) -> (b + a)
    f (Inl x) = Inr x
    f (Inr y) = Inl y
```

More Arithmetic Laws?

Have we got them all?

```
0 + x = x
x + y = y + x
x+(y+z) = (x+y)+z
1 * x = x
x * y = y * x
x*(y*z) = (x*y)*z
x*(y+z) = (x*y)+(x*z)
```

What’s behind all this?

Type = Set of values

What’s behind all this?

Type = Set of values

$[\![\texttt{Zero}]\!] = \{\}$

What's behind all this?

Type = Set of values

$$[\![\texttt{Zero}]\!] = \{\}$$
$$[\![\texttt{One}]\!] = \{\texttt{One}\}$$

What's behind all this?

Type = Set of values

$$
\begin{aligned}
[\![\texttt{Zero}]\!] &= \texttt{\{\}} \\
[\![\texttt{One}]\!] &= \texttt{\{One\}} \\
[\![\texttt{x+y}]\!] &= \texttt{\{Inl v | v} \in [\![\texttt{x}]\!]\ \texttt{\}} \\
&\quad \cup\ \texttt{\{Inr w | w} \in [\![\texttt{y}]\!]\ \texttt{\}}
\end{aligned}
$$

What's behind all this?

Type = Set of values

```
⟦Zero⟧ = {}
⟦One⟧  = {One}
⟦x+y⟧  = {Inl v | v ∈ ⟦x⟧ }
         U {Inr w | w ∈ ⟦y⟧ }
⟦x*y⟧  = {v :*: w |v ∈ ⟦x⟧ , w∈ ⟦y⟧ }
```

What's behind all this?

Size of set = natural number

```
# ⟦Zero⟧ =  0
# ⟦One⟧  =  1
# ⟦x+y⟧  =  # ⟦x⟧ + # ⟦y⟧
# ⟦x*y⟧  =  # ⟦x⟧ * # ⟦y⟧
```

Challenge

```
# ⟦Zero⟧  =  0
# ⟦One⟧   =  1
# ⟦x+y⟧   =  # ⟦x⟧ + # ⟦y⟧
# ⟦x*y⟧   =  # ⟦x⟧ * # ⟦y⟧
# ⟦x->y⟧  =  ???
```

Solution

```
# ⟦Zero⟧   =  0
# ⟦One⟧    =  1
# ⟦x+y⟧    =  # ⟦x⟧ + # ⟦y⟧
# ⟦x*y⟧    =  # ⟦x⟧ * # ⟦y⟧
# ⟦x->y⟧   =  # ⟦y⟧ ^ # ⟦x⟧
```

Arithmetic Laws

```
0^x = 0
x^0 = 1
1^x = 1
x^1 = x

x^(y+z) = (x^y)*(x^z)
x^(y*z) = (x^y)^z
(x*y)^z = (x^z)*(y^z)
```

curry/
uncurry

A Universe of Types

Datatype-Generic Approach

User-Friendly Representation

Haskell types

Datatype-Generic Approach

User-Friendly
Representation

Haskell
types

Bool

Datatype-Generic Approach

User-Friendly Representation

Generic Representation

Haskell types

Generic Universe

Bool

Datatype-Generic Approach

User-Friendly Representation

Generic Representation

Haskell types

Generic Universe

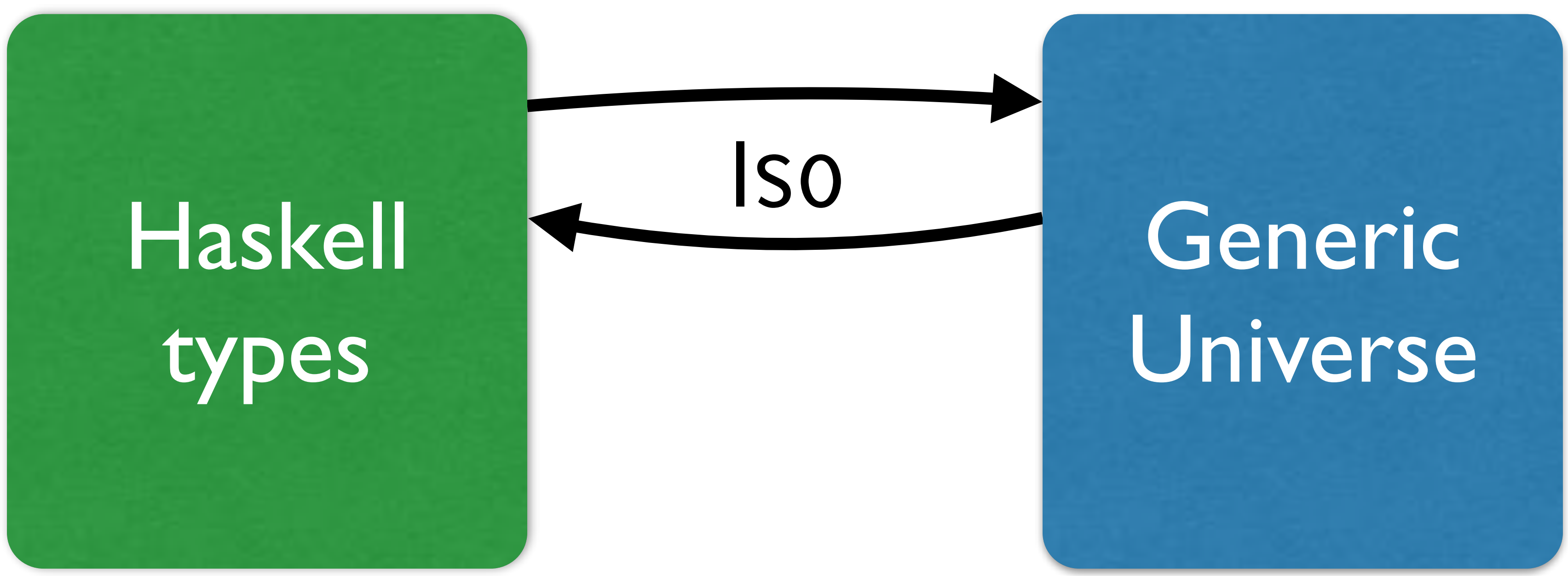
Bool

One + One

Datatype-Generic Approach

User-Friendly Representation

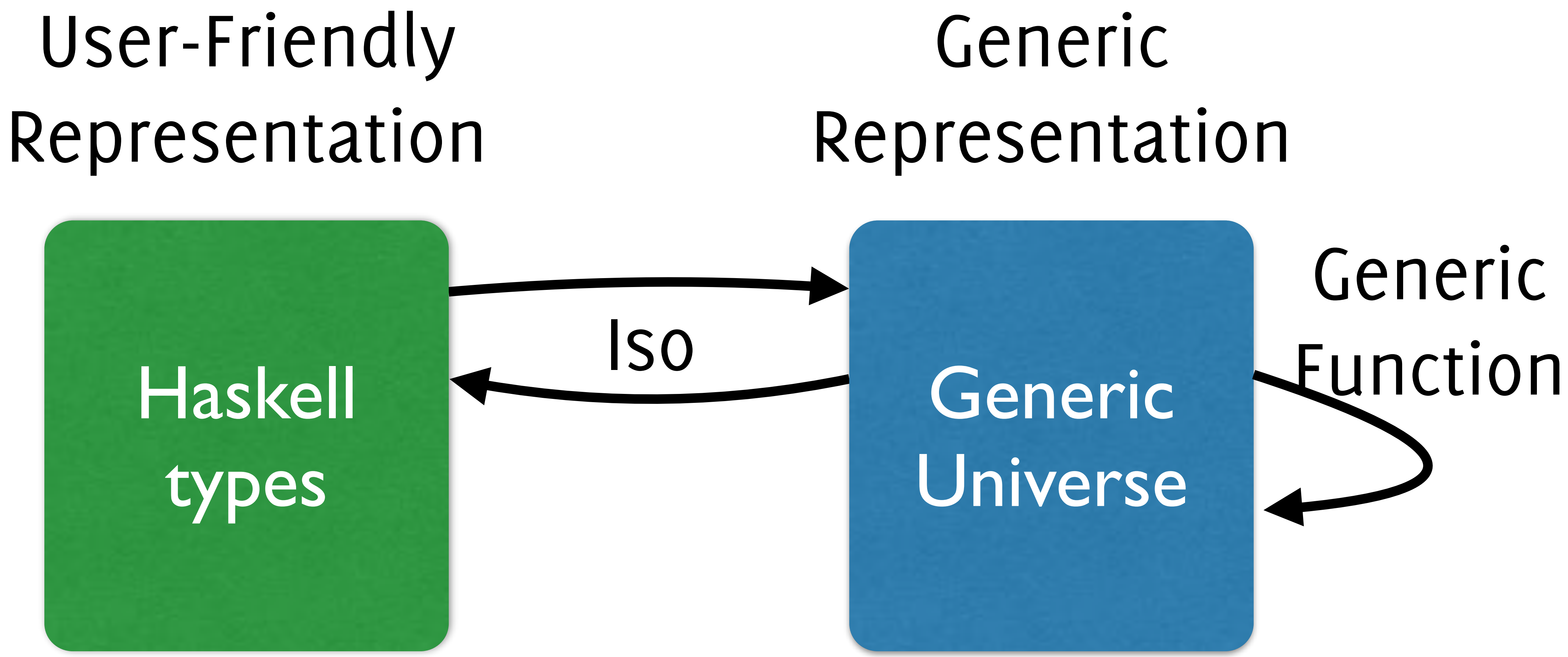
Generic Representation



Bool

One + One

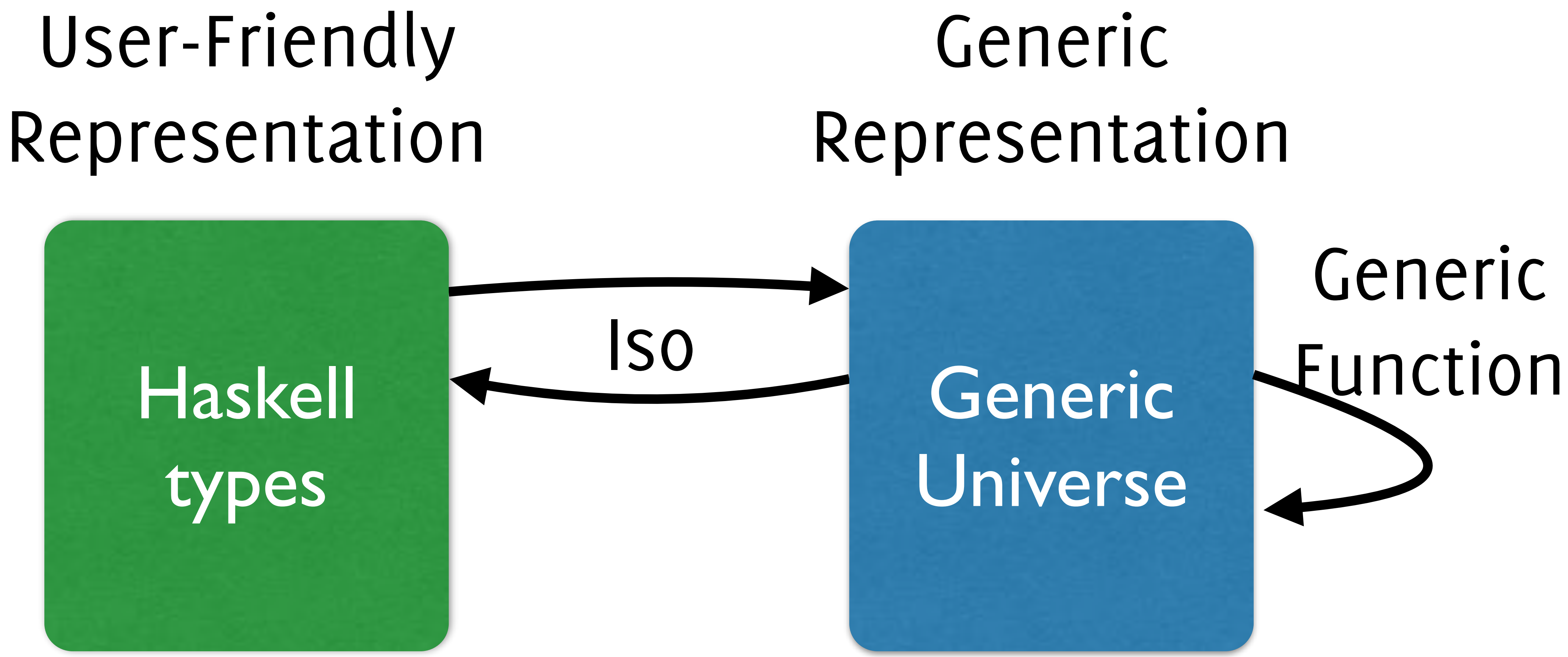
Datatype-Generic Approach



Bool

One + One

Datatype-Generic Approach

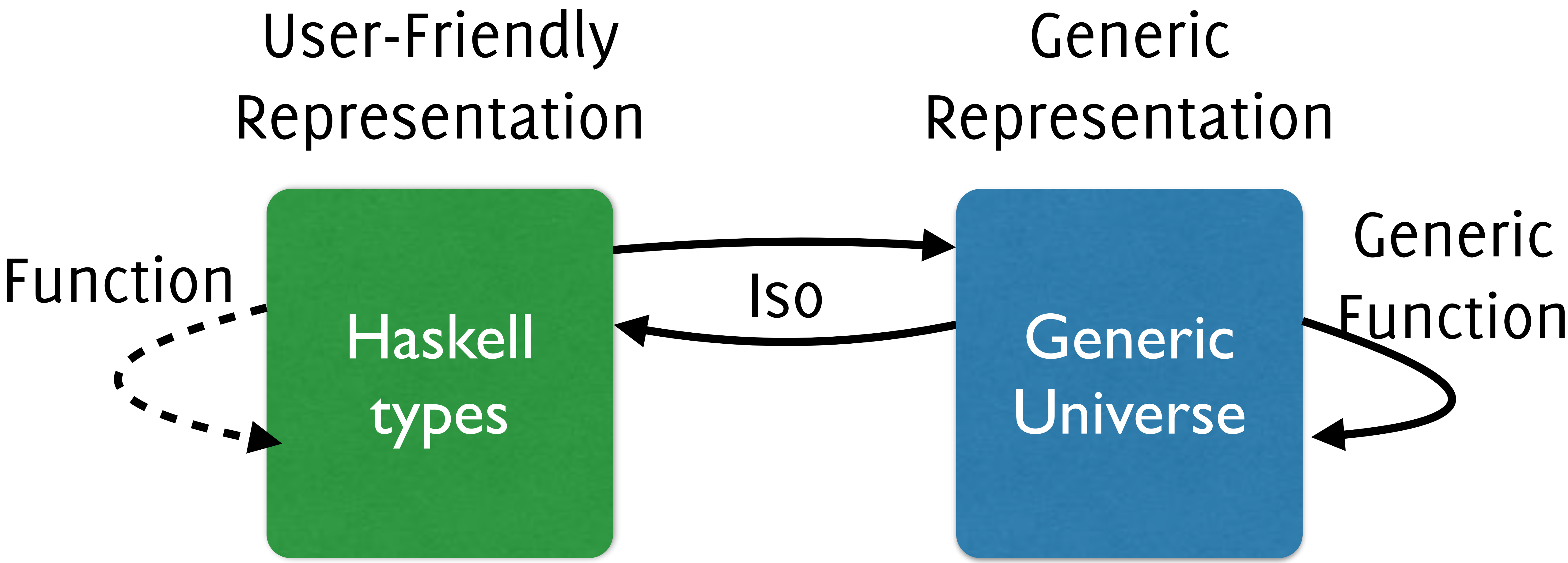


Bool

One + One

geq

Datatype-Generic Approach

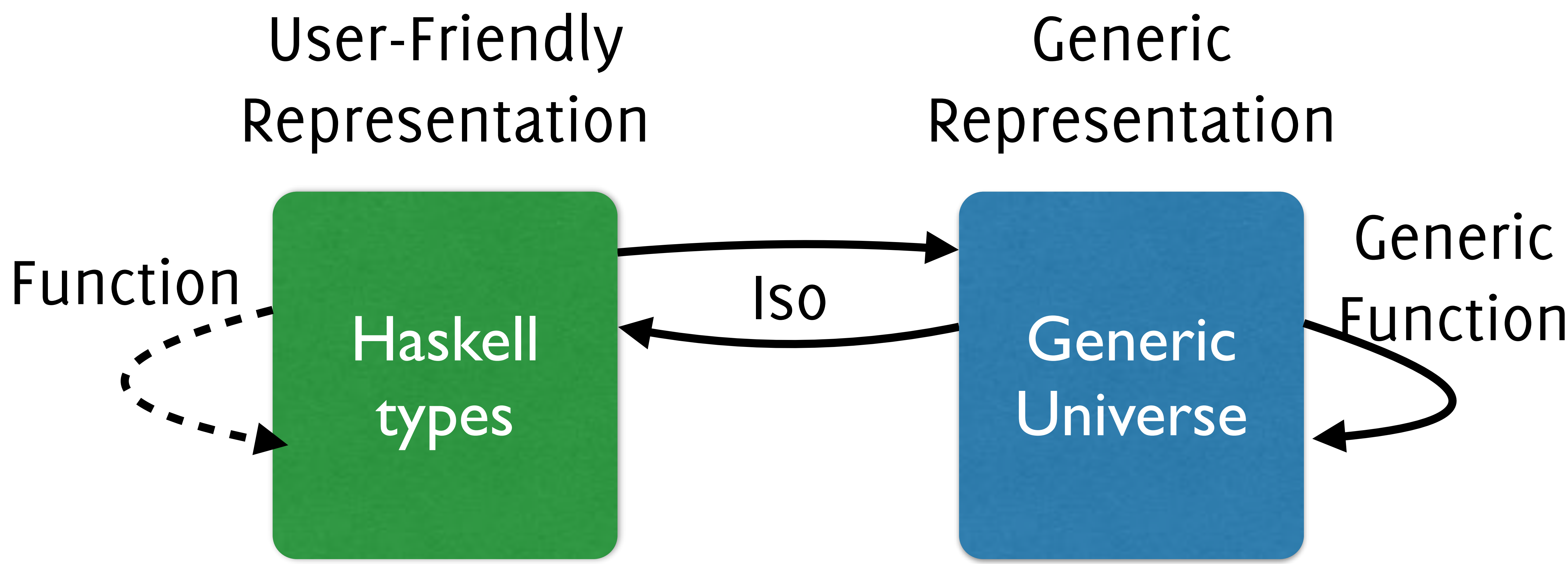


Bool

One + One

geq

Datatype-Generic Approach



(==) Bool One + One geq

Generic Building Blocks

```
data Zero
data One = One
data x + y = Inl x | Inr y
data x * y = x :*: y
```

Generic Building Blocks

```
data Zero
data One = One
data x + y = Inl x | Inr y
data x * y = x :*: y
```

Universe : all types representable with the building blocks

Generically Representable

```
class Representable a where

  type Rep a

  repIso :: a <~> Rep a
```

Generically Representable

```
class Representable a where

  type Rep a

  repIso :: a <~> Rep a
```

associated type (*)

(*) Type Checking with Open Type Functions.
T. Schrijvers et al. ICFP 2008

Representable Infrastructure

```
toRep :: Representable a
      => a -> Rep a
toRep = to repIso

fromRep :: Representable a
        => Rep a -> a
fromRep = from repIso
```

Example 1

```
data Bool = True | False

instance Representable Bool where
  type Rep Bool = One + One

  repIso = Iso t f
   where
    t True    =  Inl One
    t False   =  Inr One
    f (Inl One)  =  True
    f (Inr One)  =  False
```

Example 2

```
data Maybe a = Nothing | Just a

instance Representable (Maybe a)
where
  type Rep (Maybe a) = One + a

  repIso = Iso t f
   where
    t Nothing     =   Inl One
    t (Just x)    =   Inr x
    f (Inl One)    =   Nothing
    f (Inr x)      =   Just x
```

Example 2

```
data Maybe a = Nothing | Just a

instance Representable (Maybe a)
where
   type Rep (Maybe a) = One + a

   repIso = Iso t f
    where
      t Nothing    =   Inl One
      t (Just x)   =   Inr x
      f (Inl One)   =   Nothing
      f (Inr x)     =   Just x
```

Example 3

```
instance Representable [a] where
  type Rep [a] = One + (a * [a])

  repIso = Iso t f
   where
     t []         =  Inl One
     t (x:xs)     =  Inr (x :*: xs)
     f (Inl One)          =  []
     f (Inr (x:*:xs))     =  (x:xs)
```

Example 3

```
instance Representable [a] where
  type Rep [a] = One + (a * [a])

  repIso = Iso t f
   where
    t []         =   Inl One
    t (x:xs)     =   Inr (x :*: xs)
    f (Inl One)           =   []
    f (Inr (x:*:xs))      =   (x:xs)
```

!

Generic Equality

```
class GEq a where
  geq :: a -> a -> Bool
```

type class for equality

Derived Equality

```
instance GEq Bool where
  geq b1 b2 =
    geq (toRep b1) (toRep b2)
```

delegate to generic definition

Derived Equality

```
deriveGEq :: (Representable a, GEq (Rep a))
          => a -> a -> Bool
deriveGEq x y  =  geq (toRep x) (toRep y)

instance GEq Bool where
  geq = deriveGEq

instance GEq a => GEq (Maybe a) where
  geq = deriveGEq

instance GEq a => GEq [a] where
  geq = deriveGEq
```

This happens when you derive a type class.

Generic Definition

once for the whole universe

```
instance GEq One where
   geq One One   =  True

instance (GEq x, GEq y) => GEq (x + y)
 where
   geq (Inl x1) (Inl x2)  =  geq x1 x2
   geq (Inr y1) (Inr y2)  =  geq y1 y2
   geq _        _         =  False

instance (GEq x, GEq y) => GEq (x * y)
 where
   geq (x1:*:y1) (x2:*:y2)  =     geq x1 x2
                               && geq y1 y2
```

Primitive Types

```
instance GEq Int where
  geq  =  (==)     -- built-in definition

instance GEq Char where
  geq  =  (==)     -- built-in definition
```

Summary

Datatype Generic Programming

- ★ Structural view of datatypes
- ★ Structure-based functions
  - ✦ equality, comparison
  - ✦ (de)serialisation
  - ✦ enumeration of values

Datatype Generic Programming

- ★ Structural view of datatypes
- ★ Structure-based functions
  - ✦ equality, comparison
  - ✦ (de)serialisation
  - ✦ enumeration of values

Typed Reflection

More to Learn

- Many different universes
  - fixpoints of functors
  - containers
  - …
- Many different libraries
  `regular, uniplate, syb, generic-deriving, multirec, …`

Next time: 17/3/2015

Data Genericity	Recursion Schemes	GADTs	DSLs
Expression Problem	Monads	Type Families	Type Classes
Rank-N Poly-morphism	Effect Handlers	Free Theorems	...

Data Genericity	Recursion Schemes	GADTs	DSLs
Expression Problem	Monads	Type Families	Type Classes
Rank-N Poly-morphism	Effect Handlers	Free Theorems	...

Join the Google Group:

Leuven Haskell User Group