# Datatype-Generic Programming in Haskell

Andres Löh

(thanks to José Pedro Magalhães, Simon Peyton Jones and many others)

Skills Matter "In the Brain" – 9 April 2013



Well-Typed
The Haskell Consultants

# Datatypes are great

- Easy to introduce.
- Distinguished from existing types by the compiler.
- Added safety.
- Can use domain-specific names for types and constructors.
- Quite readable.

# Datatypes are not so great

- New datatypes have no associated library.
- Cannot be compared for equality, cannot be (de)serialized, cannot be traversed, . . .

Fortunately, there is **deriving**.

# Derivable classes

In Haskell 2010:

`Eq`, `Ord`, `Enum`, `Bounded`, `Read`, `Show`

# Derivable classes

In Haskell 2010:

Eq, Ord, Enum, Bounded, Read, Show

In GHC (in addition to the ones above):

Functor, Traversable, Typeable, Data, Generic

# What about other classes?

For many additional classes, we can intuitively derive instances.

But can we also do it in practice?

# What about other classes?

For many additional classes, we can intuitively derive instances.

But can we also do it in practice?

Options:

- use an external preprocessor,
- use Template Haskell,
- use `data-derive`,
- or use the GHC Generic support.

# GHC Generic support

From the user perspective:

**Step 1**

Define a new datatype and derive Generic for it.

```
data MyType a b =
  Flag Bool | Combo (a, a) | Other b Int (MyType a a)
  deriving Generic
```

# GHC Generic support

From the user perspective:

**Step 2**

Use a library that makes use of GHC Generic and give an empty instance declaration for a suitable type class:

```
import Data.Binary
...
instance (Binary a, Binary b) ⇒ Binary (MyType a b)
```

Analyzing **deriving**

# Equality as an example

```
class Eq′ a where
  eq :: a → a → Bool
```

Let's define some instances by hand.

# Equality on binary trees

```
data T = L | N T T
```

```
instance Eq' T where
  eq  L          L          = True
  eq (N x1 y1) (N x2 y2) = eq x1 x2 && eq y1 y2
  eq  _          _          = False
```

# Equality on another type

```
data Choice = I Int | C Char | B Choice Bool | S Choice
```

# Equality on another type

```
data Choice = I Int | C Char | B Choice Bool | S Choice
```

Assuming instances for `Int`, `Char`, `Bool`:

```
instance Eq′ Choice where
  eq (I n₁    ) (I n₂    ) = eq n₁ n₂
  eq (C c₁    ) (C c₂    ) = eq c₁ c₂
  eq (B x₁ b₁) (B x₂ b₂) = eq x₁ x₂ &&
                             eq b₁ b₂
  eq (S x₁    ) (S x₂    ) = eq x₁ x₂
  eq _          _          = False
```

# What is the pattern?

- How many cases does the function definition have?
- What is on the right hand sides?

Well-Typed

# What is the pattern?

- How many cases does the function definition have?
- What is on the right hand sides?
- How many clauses are there in the conjunctions on each right hand side?

# What is the pattern?

- How many cases does the function definition have?
- What is on the right hand sides?
- How many clauses are there in the conjunctions on each right hand side?

Relevant concepts:

- number of constructors in datatype,
- number of fields per constructor,
- recursion leads to recursion,
- other types lead to invocation of equality on those types.

# More datatypes

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

Like before, but with labels in the leaves.

# More datatypes

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

Like before, but with labels in the leaves.

```
instance Eq′ a ⇒ Eq′ (Tree a) where
  eq (Leaf n1     ) (Leaf n2     ) = eq n1 n2
  eq (Node x1 y1) (Node x2 y2) = eq x1 x2 && eq y1 y2
  eq _            _              = False
```

# Yet another equality function

This is often called a rose tree:

```
data Rose a = Fork a [Rose a]
```

# Yet another equality function

This is often called a rose tree:

```
data Rose a = Fork a [Rose a]
```

Assuming an instance for lists:

```
instance Eq′ a ⇒ Eq′ (Rose a) where
  eq (Fork x₁ xs₁) (Fork x₂ xs₂) = eq x₁ x₂ && eq xs₁ xs₂
```

# More concepts

- Parameterization of types is reflected by parameterization of the functions (via constraints on the instances).
- Using parameterized types in other types then works as expected.

# The equality pattern

## An informal description

In order to define equality for a datatype:

- introduce a parameter for each parameter of the datatype,
- introduce a case for each constructor of the datatype,
- introduce a final catch-all case returning `False`,
- for each of the other cases, compare the constructor fields pair-wise and combine them using `(&&)`,
- for each field, use the appropriate equality instance.

# The equality pattern

## An informal description

In order to define equality for a datatype:

- introduce a parameter for each parameter of the datatype,
- introduce a case for each constructor of the datatype,
- introduce a final catch-all case returning `False`,
- for each of the other cases, compare the constructor fields pair-wise and combine them using `(&&)`,
- for each field, use the appropriate equality instance.

If we can describe it, can we write a program to do it?

# Interlude:
# type isomorphisms

# Isomorphism between types

Two types $A$ and $B$ are called isomorphic if we have functions

$$f :: A \rightarrow B$$
$$g :: B \rightarrow A$$

that are mutual inverses, i.e., if

$$f \circ g \equiv id$$
$$g \circ f \equiv id$$

# Example

## Lists and Snoc-lists are isomorphic

```
data SnocList a = Lin | SnocList a :> a
```

# Example

## Lists and Snoc-lists are isomorphic

```
data SnocList a = Lin | SnocList a :> a
```

```
listToSnocList :: [a] → SnocList a
listToSnocList []          = Lin
listToSnocList (x   :  xs) = listToSnocList xs :> x

snocListToList :: SnocList a → [a]
snocListToList Lin         = []
snocListToList (xs :> x  ) = x : snocListToList xs
```

We can (but won't) prove that these are inverses.

# The idea of datatype-generic programming

- Represent a type A as an isomorphic type Rep A.

# The idea of datatype-generic programming

- Represent a type A as an isomorphic type Rep A.
- If a limited number of type constructors is used to build Rep A,

# The idea of datatype-generic programming

- Represent a type `A` as an isomorphic type `Rep A`.
- If a limited number of type constructors is used to build `Rep A`,
- then functions defined on each of these type constructors

# The idea of datatype-generic programming

- Represent a type A as an isomorphic type Rep A.
- If a limited number of type constructors is used to build Rep A,
- then functions defined on each of these type constructors
- can be lifted to work on the original type A

# The idea of datatype-generic programming

- Represent a type `A` as an isomorphic type `Rep A`.
- If a limited number of type constructors is used to build `Rep A`,
- then functions defined on each of these type constructors
- can be lifted to work on the original type `A`
- and thus on any representable type.

# Choice between constructors

Which type best encodes choice between constructors?

# Choice between constructors

Which type best encodes choice between constructors?

Well, let's restrict to two constructors first.

# Choice between constructors

Which type best encodes choice between constructors?

Well, let's restrict to two constructors first.

Booleans encode choice, but do not provide information what the choice is about.

# Choice between constructors

Which type best encodes choice between constructors?

Well, let's restrict to two constructors first.

Booleans encode choice, but do not provide information what the choice is about.

```
data Either a b = Left a | Right b
```

# Choice between constructors

Which type best encodes choice between constructors?

Well, let's restrict to two constructors first.

Booleans encode choice, but do not provide information what the choice is about.

```
data Either a b = Left a | Right b
```

Choice between three things:

```
type Either₃ a b c = Either a (Either b c)
```

Well-Typed

# Combining constructor fields

Which type best encodes combining fields?

# Combining constructor fields

Which type best encodes combining fields?

Again, let’s just consider two of them.

# Combining constructor fields

Which type best encodes combining fields?

Again, let's just consider two of them.

```
data (a, b) = (a, b)
```

# Combining constructor fields

Which type best encodes combining fields?

Again, let's just consider two of them.

```
data (a, b) = (a, b)
```

Combining three fields:

```
type Triple a b c = (a, (b, c))
```

# What about constructors without arguments?

We need another type.

# What about constructors without arguments?

We need another type.

Well, how many values does a constructor without argument encode?

# What about constructors without arguments?

We need another type.

Well, how many values does a constructor without argument encode?

```
data () = ()
```

# Representing types

# Representing types

To keep representation and original types apart, let's define isomorphic copies of the types we need:

```
data U       = U
data a :+: b = L a | R b
data a :*: b = a :*: b
```

# Representing types

To keep representation and original types apart, let's define isomorphic copies of the types we need:

```
data U       = U
data a :+: b = L a | R b
data a :*: b = a :*: b
```

We can now get started:

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

How do we represent `Bool`?

# Representing types

To keep representation and original types apart, let's define isomorphic copies of the types we need:

```
data U         = U
data a :+: b = L a | R b
data a :*: b = a :*: b
```

We can now get started:

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

How do we represent `Bool`?

```
type RepBool = U :+: U
```

# A class for representable types

```
class Generic a where
  type Rep a
  from :: a → Rep a
  to   :: Rep a → a
```

The type `Rep` is an associated type.

# A class for representable types

```
class Generic a where
  type Rep a
  from :: a → Rep a
  to   :: Rep a → a
```

The type Rep is an associated type.

Equivalent to defining Rep separately as a type family:

```
type family Rep a
```

# Representable Booleans

```
instance Generic Bool where
  type Rep Bool = U :+: U
  from False = L U
  from True  = R U
  to   (L U) = False
  to   (R U) = True
```

Well-Typed

# Representable lists

```
instance Generic [a] where
  type Rep [a] = U :+: (a :*: [a])
  from []              = L U
  from (x : xs)        = R (x :*: xs)
  to   (L U          ) = []
  to   (R (x :*: xs)) = x : xs
```

# Representable lists

```
instance Generic [a] where
  type Rep [a] = U :+: (a :*: [a])
  from []              = L U
  from (x : xs)        = R (x :*: xs)
  to   (L U          ) = []
  to   (R (x :*: xs)) = x : xs
```

Note:

- shallow transformation,
- no constraint on Generic a required.

# Representable trees

```
instance Generic (Tree a) where
  type Rep (Tree a) = a :+: (Tree a :*: Tree a)
  from (Leaf n    ) = L n
  from (Node x y  ) = R (x :*: y)
  to   (L n       ) = Leaf n
  to   (R (x :*: y)) = Node x y
```

# Representable rose trees

```
instance Generic (Rose a) where
  type Rep (Rose a) = a :*: [Rose a]
  from (Fork x xs) = x :*: xs
  to   (x :*: xs   ) = Fork x xs
```

# Representing primitive types

We don't . . .

# Back to equality

# Intermediate summary

- We have defined class Generic that maps datatypes to representations built up from U, (:+:), (:∗:) and other datatypes.
- If we can define equality on the representation types, then we should be able to obtain a generic equality function.
- Let us apply the informal recipe from earlier.

# A class for generic equality

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

# Instance for sums

```
instance (GEq a, GEq b) ⇒ GEq (a :+: b) where
  geq (L a₁) (L a₂) = geq a₁ a₂
  geq (R b₁) (R b₂) = geq b₁ b₂
  geq _      _      = False
```

# Instance for products and unit

```
instance (GEq a, GEq b) ⇒ GEq (a :*: b) where
   geq (a₁ :*: b₁) (a₂ :*: b₂) = geq a₁ a₂ && geq b₁ b₂
instance GEq U where
   geq U U = True
```

# Instances for primitive types

```
instance GEq Int where
  geq = ((==) :: Int → Int → Bool)
```

What now?

# Dispatching to the representation type

$$\mathsf{defaultEq} :: (\mathsf{Generic}\ a, \mathsf{GEq}\ (\mathsf{Rep}\ a)) \Rightarrow a \rightarrow a \rightarrow \mathsf{Bool}$$
$$\mathsf{defaultEq}\ x\ y = \mathsf{geq}\ (\mathsf{from}\ x)\ (\mathsf{from}\ y)$$

# Dispatching to the representation type

```
defaultEq :: (Generic a, GEq (Rep a)) ⇒ a → a → Bool
defaultEq x y = geq (from x) (from y)
```

Defining generic instances is now trivial:

```
instance          GEq Bool       where
  geq = defaultEq
instance GEq a ⇒ GEq [a]        where
  geq = defaultEq
instance GEq a ⇒ GEq (Tree a)   where
  geq = defaultEq
instance GEq a ⇒ GEq (Rose a)   where
  geq = defaultEq
```

# Dispatching to the representation type

```
defaultEq :: (Generic a, GEq (Rep a)) ⇒ a → a → Bool
defaultEq x y = geq (from x) (from y)
```

Or with the `DefaultSignatures` language extension:

```
class GEq a where
  geq :: a → a → Bool
  default geq :: (Generic a, GEq (Rep a)) ⇒ a → a → Bool
  geq = defaultEq

instance GEq Bool
instance GEq a ⇒ GEq [a]
instance GEq a ⇒ GEq (Tree a)
instance GEq a ⇒ GEq (Rose a)
```

Isn’t this as bad as before?

# Amount of work

## Question

Haven't we just replaced some tedious work (defining equality for a type) by some other tedious work (defining a representation for a type)?

# Amount of work

## Question

Haven't we just replaced some tedious work (defining equality for a type) by some other tedious work (defining a representation for a type)?

Yes, but:

- The representation has to be given only once, and works for potentially many generic functions.
- Since there is a single representation per type, it could be generated automatically by some other means (compiler support, TH).
- In other words, it's sufficient if we can use **deriving** on class Generic.

# So can we derive Generic?

Well-Typed

# So can we derive Generic?

Yes (with `DeriveGeneric`) ...

# So can we derive Generic?

Yes (with `DeriveGeneric`) ...

...but the representations are not quite as simple as we've pretended before:

```
class Generic a where
  type Rep a
  from :: a → Rep a
  to   :: Rep a → a
```

# So can we derive Generic?

Yes (with `DeriveGeneric`) . . .

. . . but the representations are not quite as simple as we've pretended before:

```
class Generic a where
  type Rep a :: * → *
  from :: a → Rep a x
  to   :: Rep a x → a
```

Representation types are actually of kind $* \to *$.

Well-Typed

# An extra argument?

- It's a pragmatic choice.
- Facilitates some things, because we also want to derive classes parameterized by type constructors (such as `Functor`).
- For now, let's just try to "ignore" the extra argument.

# Simple vs. GHC representation

Old:

```
type instance Rep (Tree a) = a :+: (Tree a :*: Tree a)
```

New:

```
type instance Rep (Tree a) =
  M1 D D1Tree
    (M1 C C1_0Tree
        (M1 S NoSelector (K1 P a))
      :+:
     M1 C C1_1Tree
        (M1 S NoSelector (K1 R (Tree a))
          :*:
         M1 S NoSelector (K1 R (Tree a))
        )
    )
```

Well-Typed

# Simple vs. GHC representation

Old:

```
type instance Rep (Tree a) = a :+: (Tree a :*: Tree a)
```

New:

```
type instance Rep (Tree a) =
                                          a
      :+:
        (                                  Tree a
          :*:
                                           Tree a
        )
```

Well-Typed

# Familiar components

Everything is now lifted to kind $* \to *$:

```
data U1        a = U1
data (f :+: g) a = L1 (f a) | R1 (g a)
data (f :*: g) a = f a :*: g a
```

Well-Typed

# Wrapping constant types

This is an extra type constructor wrapping every constant type:

```
newtype K1 t c a = K1 {unK1 :: c}
data P    -- marks parameters
data R    -- marks other occurrences
```

The first argument `t` is not used on the right hand side. It is supposed to be instantiated with either `P` or `R`.

# Meta information

```
newtype M1 t i f a = M1 {unM1 :: f a}
data D    -- marks datatypes
data C    -- marks constructors
data S    -- marks (record) selectors
```

Depending on the tag t, the position i is to be filled with a datatype belonging to class Datatype, Constructor, or Selector.

Well-Typed

# Meta information – contd.

```
class Datatype d where
  datatypeName :: w d f a → String
  moduleName   :: w d f a → String
```

# Meta information – contd.

```
class Datatype d where
  datatypeName :: w d f a → String
  moduleName   :: w d f a → String
```

```
instance Datatype D1Tree where
  datatypeName _ = "Tree"
  moduleName   _ =    ...
```

Similarly for constructors.

# Adapting the equality class(es)

Works on representation types:

```
class GEq′ f where
  geq′ :: f a → f a → Bool
```

Works on “normal” types:

```
class GEq a where
  geq :: a → a → Bool
  default geq :: (Generic a, GEq′ (Rep a)) ⇒ a → a → Bool
  geq x y = geq′ (from x) (from y)
```

Instance for `GEq Int` and other primitive types as before.

# Adapting the equality class(es) – contd.

```
instance (GEq' f, GEq' g) => GEq' (f :+: g) where
  geq' (L1 x) (L1 y) = geq' x y
  geq' (R1 x) (R1 y) = geq' x y
  geq' _      _      = False
```

Similarly for `:*:` and `U1`.

# Adapting the equality class(es) – contd.

```
instance (GEq′ f, GEq′ g) ⇒ GEq′ (f :+: g) where
  geq′ (L1 x) (L1 y) = geq′ x y
  geq′ (R1 x) (R1 y) = geq′ x y
  geq′ _      _      = False
```

Similarly for `:*:` and `U1`.

An instance for constant types:

```
instance GEq a ⇒ GEq′ (K1 t a) where
  geq′ (K1 x) (K1 y) = geq x y
```

# Adapting the equality classes – contd.

For equality, we ignore all meta information:

```
instance GEq′ f ⇒ GEq′ (M1 t i f) where
  geq′ (M1 x) (M1 y) = geq′ x y
```

All meta information is grouped under a single datatype, so that we can easily ignore it all if we want to.

# Adapting the equality classes – contd.

For equality, we ignore all meta information:

```
instance GEq′ f ⇒ GEq′ (M1 t i f) where
  geq′ (M1 x) (M1 y) = geq′ x y
```

All meta information is grouped under a single datatype, so that we can easily ignore it all if we want to.

Functions such as `show` and `read` can be implemented generically by accessing meta information.

# Constructor classes

To cover classes such as Functor, Traversable, Foldable generically, we need a way to map between a type constructor and its representation:

```
class Generic1 f where
  type Rep1 f :: * → *
  from1 :: f a → Rep1 f a
  to1   :: Rep1 f a → f a
```

Use the same representation type constructors, plus

```
data Par1 p   = Par1 {unPar1 :: p  }
data Rec1 f p = Rec1 {unRec1 :: f p}
```

GHC from version 7.6 is able to derive Generic1, too.

# Conclusions

- For more examples, look at `generic-deriving`.
- As a user of libraries, less boilerplate, easy to use.
- Safer (but less powerful) than Template Haskell.
- As a library author: consider using this!

Thank you – Questions?

# Extra slides

# Template Haskell

- Has the full syntax tree. Can do much more.
- You have to do more work to derive using TH.
- It's trickier to get it right. Corner cases. Name manipulation.
- Datatype-generic functions are type-checked.
- Uniform interface to the user.
- Admittedly, allowing **deriving** would be even easier.

# SYB, uniplate, multiplate, regular, multirec

- Similar ideas.
- Need other representations.
- Except for SYB, no direct GHC support.
- But we can convert! (ICFP 2013 submission)