Datatype-generic Programming with GHC

Andres Löh

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

13 July 2012

Well-Typed

The Haskell Consultants

Preparations

We use `ghc-7.4.1`.

If you happen to have a really recent development snapshot of `ghc-7.5`, that's even better.

We don't strictly need any Cabal packages, but

```
generic-deriving-1.2.1
```

is helpful.

Haven't you ever wondered
how **deriving** works?

Equality on binary trees

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

Let's try ourselves:

Equality on binary trees

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

Let's try ourselves:

```
eqT :: T → T → Bool
eqT  L          L          = True
eqT (N x₁ y₁) (N x₂ y₂) = eqT x₁ x₂ && eqT y₁ y₂
eqT  _          _          = False
```

Equality on binary trees

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

Let's try ourselves:

```
eqT :: T → T → Bool
eqT  L          L          = True
eqT (N x1 y1) (N x2 y2) = eqT x1 x2 && eqT y1 y2
eqT  _          _          = False
```

Easy enough, let's try another ...

Equality on another type

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

Well-Typed

Equality on another type

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

```
eqChoice :: Choice → Choice → Bool
eqChoice (I n₁   ) (I n₂   ) = eqInt n₁ n₂
eqChoice (C c₁   ) (C c₂   ) = eqChar c₁ c₂
eqChoice (B x₁ b₁) (B x₂ b₂) = eqChoice x₁ x₂ &&
                               eqBool b₁ b₂
eqChoice (S x₁   ) (S x₂   ) = eqChoice x₁ x₂
eqChoice _          _          = False
```

Equality on another type

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

```
eqChoice :: Choice → Choice → Bool
eqChoice (I n₁   ) (I n₂   ) = eqInt n₁ n₂
eqChoice (C c₁   ) (C c₂   ) = eqChar c₁ c₂
eqChoice (B x₁ b₁) (B x₂ b₂) = eqChoice x₁ x₂ &&
                               eqBool b₁ b₂
eqChoice (S x₁   ) (S x₂   ) = eqChoice x₁ x₂
eqChoice _          _          = False
```

Do you see a pattern?

A pattern for defining equality

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

A pattern for defining equality

- 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?

A pattern for defining equality

- 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.

How to define equality now?

More datatypes

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

Like before, but with labels in the leaves.

How to define equality now?

We need equality on `a`!

More datatypes

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

Like before, but with labels in the leaves.

How to define equality now?

We need equality on `a`!

```
eqTree :: (a → a → Bool) → Tree a → Tree a → Bool
eqTree eqa (Leaf n₁    ) (Leaf n₂    ) = eqa n₁ n₂
eqTree eqa (Node x₁ y₁) (Node x₂ y₂) = eqTree eqa x₁ x₂ &&
                                        eqTree eqa y₁ y₂
eqTree eqa _             _             = False
```

Type classes

Note how the definition of `eqTree` is perfectly suited for a type class instance:

```
instance Eq a ⇒ Eq (Tree a) where
  (==) = eqTree (==)
```

Type classes

Note how the definition of eqTree is perfectly suited for a type class instance:

```
instance Eq a ⇒ Eq (Tree a) where
  (==) = eqTree (==)
```

In fact, type classes are usually implemented as dictionaries, and an instance declaration is translated into a dictionary transformer.

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]
```

Let's assume we already have:

```
eqList :: (a → a → Bool) → [a] → [a] → Bool
```

How to define `eqRose`?

Yet another equality function

This is often called a rose tree:

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

Let's assume we already have:

```
eqList :: (a → a → Bool) → [a] → [a] → Bool
```

How to define `eqRose`?

```
eqRose :: (a → a → Bool) → Rose a → Rose a → Bool
eqRose eqa (Fork x₁ xs₁) (Fork x₂ xs₂) =
                eqa x₁ x₂ && eqList (eqRose eqa) xs₁ xs₂
```

No fallback case needed because there is only one constructor.

More concepts

- Parameterization of types is reflected by parameterization of the functions.
- Application of parameterized types is reflected by application of the functions.

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 function; combine equality functions and use the parameter functions as needed.

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 function; combine equality functions and use the parameter functions as needed.

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

$$f :: A \to B$$
$$g :: B \to A$$

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 prove that these are inverses.

The idea of datatype-generic programming

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

Well-Typed

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,

Well-Typed

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.

The idea of datatype-generic programming – contd.

In fact, we do not even quite need an isomorphic type.

For a type A, we need a type $\mathsf{Rep}\ A$ and $\mathsf{from} :: A \to \mathsf{Rep}\ A$ and $\mathsf{to} :: \mathsf{Rep}\ A \to A$ such that

$$\mathsf{to} \circ \mathsf{from} \equiv \mathsf{id}$$

We call such a combination an embedding-projection pair.

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 a
```

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 a
```

Choice between three things:

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

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)
```

Well-Typed

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 () = ()
```

Well-Typed

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
```

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. GHC allows us to define datatypes and type synonyms within classes, depending on the class parameter(s).

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. GHC allows us to define datatypes and type synonyms within classes, depending on the class parameter(s).

This is 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
```

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
```

Question

Are Bool and Rep Bool isomorphic?

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
```

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
```

Note that the representation of recursive types mentions the original types – if needed, we can apply the transformation multiple times.

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 that the representation of recursive types mentions the original types – if needed, we can apply the transformation multiple times.

Note further that we do not require `Generic a`.

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
```

Well-Typed

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

For some types, it does not make sense to define a structural representation – for such types, we will have to define generic functions directly.

```
instance Generic Int where
  type Rep Int = Int
  from = id
  to   = id
```

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.

Equality on sums

```
eqSum :: (  a          → a          → Bool) →
         (         b →          b → Bool) →
            a :+: b → a :+: b → Bool
eqSum eqa eqb (L a₁) (L a₂) = eqa a₁ a₂
eqSum eqa eqb (R b₁) (R b₂) = eqb b₁ b₂
eqSum eqa eqb  _       _     = False
```

Equality on products

```
eqProd :: (  a       → a       → Bool) →
          (      b →       b → Bool) →
             a :*: b → a :*: b → Bool
eqProd eqa eqb (a₁ :*: b₁) (a₂ :*: b₂) =
  eqa a₁ a₂ && eqb b₁ b₂
```

Well-Typed

Equality on units

```
eqUnit :: U → U → Bool
eqUnit U U = True
```

What now?

A class for generic equality

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

A class for generic equality

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

```
instance (GEq a, GEq b) ⇒ GEq (a :+: b) where
  geq = eqSum geq geq
instance (GEq a, GEq b) ⇒ GEq (a :*: b) where
  geq = eqProd geq geq
instance                  GEq U         where
  geq = eqUnit
```

A class for generic equality

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

```
instance (GEq a, GEq b) ⇒ GEq (a :+: b) where
  geq = eqSum geq geq
instance (GEq a, GEq b) ⇒ GEq (a :*: b) where
  geq = eqProd geq geq
instance                  GEq U         where
  geq = eqUnit
```

Instances for primitive types:

```
instance                  GEq Int       where
  geq = eqInt
```

Dispatching to the representation type

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

Dispatching to the representation type

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

Defining generic instances is now trivial:

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

Dispatching to the representation type

```
eq :: (Generic a, GEq (Rep a)) ⇒ a → a → Bool
eq 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 = eq
instance GEq Bool
instance GEq a ⇒ GEq [a]
instance GEq a ⇒ GEq (Tree a)
instance GEq a ⇒ GEq (Rose a)
```

Have we won

or

have we lost?

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?

So can we derive `Generic`?

Yes! (With `DeriveGeneric`.)

Well-Typed

So can we derive Generic?

Yes! (With `DeriveGeneric`.)

But it's not quite as simple as we've seen 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 it's not quite as simple as we've seen before:

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

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

An extra argument?

Having an extra argument is an admittedly somewhat questionable, but pragmatic choice:

- We are not only interested in deriving classes parameterized by types, but also classes by type constructors.
- Haskell's kind system doesn't (well, didn't) support kind polymorphism.
- The current choice allows two representations, one for fully applied types of kind $*$, one for type constructors of kind $* \to *$, to share a single set of representation type constructors.

Well-Typed

An extra argument?

Having an extra argument is an admittedly somewhat questionable, but pragmatic choice:

- We are not only interested in deriving classes parameterized by types, but also classes by type constructors.
- Haskell's kind system doesn't (well, didn't) support kind polymorphism.
- The current choice allows two representations, one for fully applied types of kind $*$, one for type constructors of kind $* \to *$, to share a single set of representation type constructors.

For the beginning, we can just try to "ignore" the additional argument.

Well-Typed

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

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
```

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.

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.

Example functions

Many example functions are defined in the package `generic-deriving`.

Example functions

Many example functions are defined in the package `generic-deriving`.

Many related representations are used in other packages on Hackage, such as `instant-generics` or `regular` or `multirec`.

For these, the representations cannot be generated automatically by GHC (but usually by Template Haskell).

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 will be able to derive Generic1, too.