Generic programming

Haskell and Cryptocurrencies

Dr. Lars Brünjes, IOG
Robertino Martinez, IOG
Karina Lopez, IOG

September, 2023

Goals

- Introduce datatype-generic programming.
- GHC generics.

Motivation

User-defined datatypes are fantastic

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

User-defined datatypes are problematic

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

User-defined datatypes are problematic

- 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 , Foldable , Traversable , Typeable ,
Data , Generic
```

What about other classes?

For many additional classes, we can come up with an informal algorithm that explains how to derive them. we can intuitively derive instances.

But can we also do it in practice?

Options for deriving other classes

- Template Haskell.
- External preprocessor (such as `data-derive`).
- GHC `Generic` support.
- One of many other libraries for datatype-generic programming in Haskell.

Options for deriving other classes

- Template Haskell.
- External preprocessor (such as `data-derive`).
- GHC `Generic` support.
- One of many other libraries for datatype-generic programming in Haskell.

In this lecture, we'll look at the ideas of datatype-generic programming that are the foundation for the latter two approaches.

GHC generics – the user perspective

Using a generic function

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
```

This requires the `DeriveGeneric` language extension and importing the `Generic` type class from `GHC.Generics`.

Using a generic function

Step 2a

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)
```

That is all. The generic implementation is derived by GHC and supplied automatically.

Using a generic function

Step 2b

With recent GHC versions, we can also use `deriving` instead of an empty instance declaration:

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

This requires the `DeriveAnyClass` extension. It only works in a truly robust way from GHC 8.2 onwards (although in some simple cases it may work with 8.0 or even 7.10).

Deriving `Generic` is still necessary!

Explaining how to derive a class

Equality as an example

Let's work with our "copy" of the `Eq` class:

```
class Eq' a where
  eq' :: a -> a -> Bool
```

Equality as an example

Let's work with our "copy" of the `Eq` class:

```
class Eq' a where
  eq' :: a -> a -> Bool
```

Let's define some instances by hand.

Equality on binary trees

```haskell
data T = A | N T T
```

```haskell
instance Eq' T where
  eq'  A          A          = 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

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

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

```haskell
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?

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

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

More datatypes

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

```haskell
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

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

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?*

Datatype-generic programming

The key idea of datatype-generic programming

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

The key 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 key 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 key 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 key 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.

Many options

What to choose as representation `Rep` of a type?

Many options

What to choose as representation `Rep` of a type?

- There are many different options.
- The choice influences which datatypes can be represented, how easy it is to define certain generic functions, and how the resulting generic programming library looks and feels.
- This is one of the reasons why there are so many different approaches to datatype-generic programming in Haskell.

Many options

What to choose as representation `Rep` of a type?

- There are many different options.
- The choice influences which datatypes can be represented, how easy it is to define certain generic functions, and how the resulting generic programming library looks and feels.
- This is one of the reasons why there are so many different approaches to datatype-generic programming in Haskell.

In the following, we will primarily look at one particular representation, called the binary sums of products representation.

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)
```

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.

```haskell
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?

```haskell
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 example of 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 example of an associated type.

Equivalent to defining `Rep` separately as a type family:

```
type family Rep a
```

Representable Booleans

```haskell
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 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

```haskell
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

```haskell
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 do not define instances of `Generic` for primitive types such as `Int`, `Char` or `Float`.

Representing primitive types

We do not define instances of `Generic` for primitive types such as `Int`, `Char` or `Float`.

This will imply that we cannot derive a generic function for such primitive types.

If we want a function to work on these types, we have to provide an appropriate class instance manually.

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

```haskell
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

```haskell
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

```haskell
instance GEq Int where
  geq = ((==) :: Int -> Int -> Bool)
```

What now?

Dispatching to the representation type

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

Dispatching to the representation type

```haskell
defaultEq ::
  (Generic a, GEq (Rep a)) => a -> a -> Bool
defaultEq x y = geq (from x) (from y)
```

Defining generic instances is now trivial:

```haskell
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

```haskell
defaultEq ::
  (Generic a, GEq (Rep a)) => a -> a -> Bool
defaultEq x y = geq (from x) (from y)
```

Or with the DefaultSignatures language extension:

```haskell
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)
```

Dispatching to the representation type

```haskell
defaultEq ::
  (Generic a, GEq (Rep a)) => a -> a -> Bool
defaultEq x y = geq (from x) (from y)
```

Or with `DeriveAnyClass` / `StandaloneDeriving`:

```haskell
class GEq a where
  geq :: a -> a -> Bool
  default geq ::
    (Generic a, GEq (Rep a)) => a -> a -> Bool
  geq = defaultEq

deriving instance GEq Bool
deriving instance GEq a => GEq [a]
deriving instance GEq a => GEq (Tree a)
deriving 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`?

So can we derive `Generic`?

Yes (with `DeriveGeneric`) ...

So can we derive `Generic`?

Yes (with `DeriveGeneric`) ...

...but the representations in GHC 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 in GHC 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 `* -> *`.

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:

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

New:

```haskell
type instance Rep (Tree a) =
  D1 (MetaData "Tree" "Main" "main" False)
    (C1 (MetaCons "Leaf" PrefixI False)
       (S1 (MetaSel Nothing NoSourceUnpackedness NoSourceStrictness DecidedLazy)
          (Rec0 a)
       )
      :+:
     C1 (MetaCons "Node" PrefixI False)
       (S1 (MetaSel Nothing NoSourceUnpackedness NoSourceStrictness DecidedLazy)
          (Rec0 (Tree a))
         :*:
        S1 (MetaSel Nothing NoSourceUnpackedness NoSourceStrictness DecidedLazy)
          (Rec0 (Tree a))
       )
    )
```

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
       )
```

Familiar components

Everything is now lifted to kind `* -> *`:

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

All the extra noise

- Every ocurrence of a type as an argument is wrapped in an extra application of `Rec0`.
- This is to detect where the “structure” combinators end and the new type starts.

All the extra noise

- Every ocurrence of a type as an argument is wrapped in an extra application of `Rec0`.
- This is to detect where the “structure” combinators end and the new type starts.
- All the metadata is included a type-level information, too.
- This includes names of the datatypes, constructors and record selectors, but also fixity info for infix constructors, and strictness information.
- This is for defining generic functions that make use of this info (consider e.g. `Show` or `Read`).

Extra constructors

```haskell
type Rec0 = K1 R

newtype K1 i c p = K1 {unK1 :: c}
```

There is no other instantiation of `K1` than `R` actually being used anymore.

Extra constructors

```haskell
type Rec0 = K1 R

newtype K1 i c p = K1 {unK1 :: c}
```

There is no other instantiation of `K1` than `R` actually being used anymore.

```haskell
type D1 = M1 D  -- for datatypes
type C1 = M1 C  -- for constructors
type S1 = M1 S  -- for selectors

newtype M1 i (c :: Meta) f p = M1 {unM1 :: f p}
```

Used to capture various kinds of metadata.

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.

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

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

Adapting the equality class(es) – contd.

```haskell
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:

```haskell
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, such as here.

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, such as here.

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    }
data (:.:) f g p = Comp1 {unComp1 :: f (g p)}
```

GHC is able to derive `Generic1`, too.

Conclusions

- Generic programming is a powerful programming technique.
- There are also many other interesting packages in the area (most notably `generics-sop`, `uniplate` and `syb`).

Outlook: Template Haskell (TH)

- A full meta-programming extension to Haskell.
- Has the full syntax tree. Can do much more.
- You have to do more work to derive classes using TH.
- It's trickier to get it right. Corner cases. Name manipulation.
- Datatype-generic functions are type-checked, TH only checks the generated code.