generics-eot Documentation

Release 0.1

Sönke Hahn

April 03, 2018

Contents

`generics-eot` is a library for datatype generic programming that tries to be very simple to understand and use. It's heavily inspired by the awesome `generics-sop` package (http://hackage.haskell.org/package/generics-sop).

CHAPTER 1

Links:

github:	https://github.com/soenkehahn/generics-eot
hackage:	http://hackage.haskell.org/package/generics-eot
stackage:	https://www.stackage.org/package/generics-eot
readthedocs:	http://generics-eot.readthedocs.org/en/latest/

CHAPTER 2

Tutorial:

generics-eot tutorial

This tutorial is meant to be read alongside with the haddock comments in Generics.Eot. Its source is a compiled haskell file, so we have to get some language pragmas and imports out of the way first:

```
{-# LANGUAGE DefaultSignatures #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE UndecidableInstances #-}

module Generics.Eot.Tutorial where

import           Data.Char
import           Data.List
import           Data.Typeable

import           Generics.Eot
```

`generics-eot` allows roughly three different kinds of operations:

1. Accessing meta information about ADTs (`datatype` for names, `Proxy` and `Eot` for field types). Example: Generation of database schemas for ADTs.
2. Deconstructing values generically (`toEot`). Example: Serialization to a binary format.
3. Constructing values of an ADT generically (`fromEot`). Example: Deserialization from a binary format.

Sometimes only one of the three forms is used but often multiple have to be combined. For example serialization to JSON usually requires both `datatype` and `toEot`.

1st Example: Meta Information Without Types: Field Names

This simple function extracts the names of all field selectors and returns them as a list:

```
namesOfFields :: HasEot a => Proxy a -> [String]
namesOfFields proxy =
  nub $
  concatMap (fieldNames . fields) $
```

```
  constructors $ datatype proxy
  where
    fieldNames :: Fields -> [String]
    fieldNames fields = case fields of
      Selectors names -> names
      _ -> []
```

And here's proof that it works (using doctest):

```
data A = A1 {
    foo :: String,
    bar :: Int
  }
  | A2 {
    bar :: Int,
    baz :: Bool
  }
  deriving (Generic, Show)

-- $ >>> namesOfFields (Proxy :: Proxy A)
-- ["foo","bar","baz"]
```

The `Generic` instance: Don't forget!!!

To be able to use generic functions that are written with `generics-eot` you need to derive an instance for `GHC.Generics.Generic` (using `DeriveGeneric`) for your ADTs. This will automatically give you an instance for `HasEot`.

When the instance for `GHC.Generics.Generic` is missing the type error messages are unfortunately very confusing and unhelpful. They go something like this:

```
Couldn't match type 'GHC.Generics.Rep WithoutGeneric'
                with 'GHC.Generics.D1 c f'
The type variables 'c', 'f' are ambiguous
In the expression: namesOfFields (Proxy :: Proxy WithoutGeneric)
```

So don't forget: you need a `Generic` instance.

`Eot`: Isomorphic representations

Part of the type class `HasEot` is the type-level function `Eot` that maps ADTs to isomorphic types. These isomorphic types are always a combination of `Either`s, tuples and the uninhabited type `Void`. For example this type:

```
data B = B1 Int | B2 String Bool | B3
  deriving (Generic)
```

would be mapped to:

```
Either (Int, ()) (Either (String, (Bool, ())) (Either () Void))
```

Tip: Here's how you can execute the type-level function `Eot` in `ghci`:

```
-- $ >>> :kind! Eot B
-- Eot B :: *
-- = Either (Int, ()) (Either ([Char], (Bool, ())) (Either () Void))
```

For the exact rules of this mapping see here: Eot.

If we have an ADT `a` then we can convert values of type `a` to this isomorphic representation `Eot a` with `toEot` and we can convert in the other direction with `fromEot`. Generic functions always operate on these isomorphic representations and then convert from or to the real ADTs with `fromEot` and `toEot`.

These generic isomorphic types are referred to as "eot" – short for "`Either`s of tuples".

2nd Example: Deconstructing Values: Serialization

We start by writing a function that operates on the eot representations. The eot representations follow simple patterns and always look similar, but they don't look exactly the same for different ADTs. For this reason we have to use a type class:

```
class EotSerialize eot where
  eotSerialize :: Int -- ^ The number of the constructor being passed in
    -> eot -- ^ The eot representation
    -> [Int] -- ^ A simple serialization format
```

Now we need to write instances for the types that occur in eot types. Usually these are:

- `Either this next`:
  - If as eot value we get `Left this` it means that the original value was constructed with the constructor that corresponds to `this`. In this case we put the number of the constructor into the output and continue with serializing the fields of type `this`.
  - If we get `Right rest` it means that one of the following constructors was the one that the original value was built with. We continue by increasing the constructor counter and serializing the value of type `rest`.

  Note that this results in `EotSerialize` class constraints for both `this` and `rest`. If we write the correct instances for all eot types these constraints should always be fulfilled.

```
instance (EotSerialize this, EotSerialize next) =>
  EotSerialize (Either this next) where

  eotSerialize n (Left fields) = n : eotSerialize n fields
  eotSerialize n (Right next) = eotSerialize (succ n) next
```

- `Void`: We need this instance to make the compiler happy, but it'll never be used. If you look at the type you can also see that: an argument of type `Void` cannot be constructed. (Often the function `Data.Void.absurd` comes in handy for implementing these cases. `generics-eot` re-exports both the type `Data.Void.Void` and `Data.Void.absurd` for convenience.)

```
instance EotSerialize Void where
  eotSerialize _n void = absurd void
```

- `(x, xs)`: Right-nested 2-tuples are used to encode all the fields for one specific constructor. So `x` is the current field and `xs` are the remaining fields. To serialize this we serialize `x` (using `serialize`) and also write the length of the resulting list into the output. This will allow deserialization.

  Note: We could use `EotSerialize` to serialize the fields. But that would be a bit untrue to the spirit, since the fields are not eot types. Apart from that we might want to encode a field of e.g. type `Either a b` differently than the eot type `Either a b`. So we use a very similar but distinct type class called `Serialize`.

  The value of type `xs` contains the remaining fields and will be encoded recursively with `eotSerialize`:

```
instance (Serialize x, EotSerialize xs) => EotSerialize (x, xs) where
  eotSerialize n (x, xs) =
```

```
    let xInts = serialize x
    in length xInts : xInts ++ eotSerialize n xs
```

- `()`: Finally we need an instance for the unit type that marks the end of the fields encoded in 2-tuples. Since `()` doesn't carry any information, we can encode it as the empty list:

```
instance EotSerialize () where
  eotSerialize _ () = []
```

This is the class `Serialize`:

```
class Serialize a where
  serialize :: a -> [Int]
```

We give `serialize` a default implementation, but please ignore that for now. It'll be explained later in the section about *DefaultSignatures*:

```
  default serialize :: (HasEot a, EotSerialize (Eot a)) => a -> [Int]
  serialize = genericSerialize
```

`Serialize` is used to serialize every field of the used ADTs, so we need instances for all of them:

```
instance Serialize Int where
  serialize i = [i]

instance Serialize String where
  serialize = map ord

instance Serialize Bool where
  serialize True = [1]
  serialize False = [0]

instance Serialize () where
  serialize () = []
```

To tie everything together we provide a function `genericSerialize` that converts a value of some ADT into an eot value using `toEot` and then uses `eotSerialize` to convert that eot value into a list of `Int`s.

```
genericSerialize :: (HasEot a, EotSerialize (Eot a)) => a -> [Int]
genericSerialize = eotSerialize 0 . toEot
```

And it works too:

```
-- $ >>> genericSerialize (A1 "foo" 42)
-- [0,3,102,111,111,1,42]
-- >>> genericSerialize (A2 23 True)
-- [1,1,23,1,1]
```

3rd Example: Constructing Values: Deserialization

Deserialization works very similarly. It differs in that the functions turn lists of `Int`s into eot values.

Here's the `EotDeserialize` class with instances for:

- `Either this next`
- `Void`
- `(x, xs)`

- ()

```
class EotDeserialize eot where
  eotDeserialize :: [Int] -> eot

instance (EotDeserialize this, EotDeserialize next) =>
  EotDeserialize (Either this next) where

  eotDeserialize (0 : r) = Left $ eotDeserialize r
  eotDeserialize (n : r) = Right $ eotDeserialize (pred n : r)
  eotDeserialize [] = error "invalid input"

instance EotDeserialize Void where
  eotDeserialize _ = error "invalid input"

instance (Deserialize x, EotDeserialize xs) =>
  EotDeserialize (x, xs) where

  eotDeserialize (len : r) =
    let (this, rest) = splitAt len r
    in (deserialize this, eotDeserialize rest)
  eotDeserialize [] = error "invalid input"

instance EotDeserialize () where
  eotDeserialize [] = ()
  eotDeserialize (_ : _) = error "invalid input"
```

And here's the `Deserialize` class plus all instances to deserialize the fields:

```
class Deserialize a where
  deserialize :: [Int] -> a

instance Deserialize Int where
  deserialize [n] = n
  deserialize _ = error "invalid input"

instance Deserialize String where
  deserialize = map chr

instance Deserialize () where
  deserialize [] = ()
  deserialize (_ : _) = error "invalid input"

instance Deserialize Bool where
  deserialize [0] = False
  deserialize [1] = True
  deserialize _ = error "invalid input"
```

And here's `genericDeserialize` to tie it together. It uses `eotDeserialize` to convert a list of `Int`s into an eot value and then `fromEot` to construct a value of the wanted ADT.

```
genericDeserialize :: (HasEot a, EotDeserialize (Eot a)) => [Int] -> a
genericDeserialize = fromEot . eotDeserialize
```

Here you can see it in action:

```
-- $ >>> genericDeserialize [0,3,102,111,111,1,42] :: A
-- A1 {foo = "foo", bar = 42}
-- >>> genericDeserialize [1,1,23,1,1] :: A
-- A2 {bar = 23, baz = True}
```

And it is the inverse of `genericSerialize`:

```
-- $ >>> (genericDeserialize $ genericSerialize $ A1 "foo" 42) :: A
-- A1 {foo = "foo", bar = 42}
```

4th Example: Meta Information with types: generating SQL schemas

Accessing meta information **including** the types works very similarly to deconstructing or constructing values. It uses the same structure of type classes and instances for the eot-types. The difference is: since we don't want actual values of our ADT as input or output we operate on `Proxy`s of our eot-types.

As an example we're going to implement a function that generates SQL statements that create tables that our ADTs would fit into. To be able to use nice names for the table and columns we're going to traverse the type-less meta information (see *1st Example*) at the same time.

(Note that the generated SQL statements are targeted at a fictional database implementation that magically understands Haskell types like `Int` and `String`, or rather `[Char]`.)

Again we start off by writing a class that operates on the eot-types. Besides the eot-type the class has an additional parameter, `meta`, that will be instantiated by the corresponding types used for untyped meta information.

```
class EotCreateTableStatement meta eot where
  eotCreateTableStatement :: meta -> Proxy eot -> [String]
```

Our first instance is for the complete datatype. `eot` is instantiated to `Either fields Void`. Note that this instance only works for ADTs with exactly one constructor as we don't support types with multiple constructors. `meta` is instantiated to `Datatype` which is the type for meta information for ADTs.

```
instance EotCreateTableStatement [String] fields =>
  EotCreateTableStatement Datatype (Either fields Void) where

  eotCreateTableStatement datatype Proxy = case datatype of
    Datatype name [Constructor _ (Selectors fields)] ->
      "CREATE TABLE " :
      name :
      " COLUMNS " :
      "(" :
      intercalate ", " (eotCreateTableStatement fields (Proxy :: Proxy fields)) :
      ");" :
      []
    Datatype _ [Constructor name (NoSelectors _)] ->
      error ("constructor " ++ name ++ " has no selectors, this is not supported")
    Datatype name _ ->
      error ("type " ++ name ++ " must have exactly one constructor")
```

The second instance is responsible for creating the parts of the SQL statements that declare the columns. As such it has to traverse the fields of our ADT. `eot` is instantiated to the usual `(x, xs)`. `meta` is instantiated to `[String]`, representing the field names. The name of the field type is obtained using `typeRep`, therefore we need a `Typeable x` constraint.

```
instance (Typeable x, EotCreateTableStatement [String] xs) =>
  EotCreateTableStatement [String] (x, xs) where

  eotCreateTableStatement (field : fields) Proxy =
    (field ++ " " ++ show (typeRep (Proxy :: Proxy x))) :
    eotCreateTableStatement fields (Proxy :: Proxy xs)
  eotCreateTableStatement [] Proxy = error "impossible"
```

The last instances is for `()`. It's needed as the base case for traversing the fields and as such returns just an empty list.

```
instance EotCreateTableStatement [String] () where
  eotCreateTableStatement [] Proxy = []
  eotCreateTableStatement (_ : _) Proxy = error "impossible"
```

`createTableStatement` ties everything together. It obtaines the meta information through `datatype` passing a `Proxy` for `a`. And it creates a `Proxy` for the eot-type `Proxy :: Proxy (Eot a)`. Then it calls `eotCreateTableStatement` and just `concat`s the resulting snippets.

```
createTableStatement :: forall a . (HasEot a, EotCreateTableStatement Datatype (Eot a)) =>
  Proxy a -> String
createTableStatement proxy =
  concat $ eotCreateTableStatement (datatype proxy) (Proxy :: Proxy (Eot a))
```

As an example, we're going to use `Person`:

```
data Person
  = Person {
    name :: String,
    age :: Int
  }
  deriving (Generic)
```

And here's the created SQL statement:

```
-- $ >>> putStrLn $ createTableStatement (Proxy :: Proxy Person)
-- CREATE TABLE Person COLUMNS (name [Char], age Int);
```

If we try to use an ADT with multiple constructors, we get a type error due to a missing instance:

```
-- $ >>> putStrLn $ createTableStatement (Proxy :: Proxy A)
-- <BLANKLINE>
-- ...
--     • No instance for (EotCreateTableStatement
--                          Datatype
--                          (Either ([Char], (Int, ())) (Either (Int, (Bool, ())) Void)))
--         arising from a use of ‘createTableStatement’
-- ...
```

If we try to use it with an ADT with a single constructor but no selectors, we get a runtime error:

```
data NoSelectors
  = NotSupported Int Bool
  deriving (Generic)

-- $ >>> putStrLn $ createTableStatement (Proxy :: Proxy NoSelectors)
-- *** Exception: constructor NotSupported has no selectors, this is not supported
-- ...
```

DefaultSignatures

There is a GHC language extension called DefaultSignatures. In itself it has little to do with generic programming, but it makes a good companion.

How DefaultSignatures work

Imagine you have a type class called `ToString` which allows to convert values to `String`s:

```
class ToString a where
  toString :: a -> String
```

You can write instances manually, but you might be tempted to give the following default implementation for `toString`:

```
toString = show
```

The idea is that then you can just write down an empty `ToString` instance:

```
instance ToString A
```

and you get to use `toString` on values of type `A` for free, because `A` has a `Show` instance.

But that default implementation doesn't work, because in the class declaration we don't have an instance for `Show a`. `ghc` says:

```
Could not deduce (Show a) arising from a use of 'show'
 from the context (ToString a)
```

One solution would be to make `ToString` a subclass of `Show`, but then we cannot implement `ToString` instances manually anymore for types that don't have a `Show` instance. `DefaultSignatures` provide a better solution. The extension allows you to further narrow down the type for your default implementation for class methods:

```
class ToString2 a where
  toString2 :: a -> String
  default toString2 :: Show a => a -> String
  toString2 = show
```

Then writing down empty instances works for types that have a `Show` instance:

```
instance ToString2 Int

-- $ >>> toString2 (42 :: Int)
-- "42"
```

Note: if you write down an empty `ToString2` instances for a type that does not have a `Show` instance, the error message looks like this:

```
No instance for (Show NoShow)
```

This might be confusing especially since haddock docs don't list the default signatures or implementations and users of the class might be wondering why `Show` comes into play at all.

How to use `DefaultSignatures` for generic programming

`DefaultSignatures` are especially handy when doing generic programming. Remember the type class `Serialize` from the *second example*? In that example we used it to serialize the fields of our ADTs in the generic serialization through `genericSerialize` and `EotSerialize`. We just assumed that we would have a manual implementation for all field types. But we also gave it a default implementation for `serialize` in terms of `genericSerialize`:

```
default serialize :: (HasEot a, EotSerialize (Eot a)) => a -> [Int]
serialize = genericSerialize
```

Note that the default implementation has the same class constraints as `genericSerialize`.

Now we can write empty instances for custom ADTs:

```haskell
data C
  = C1 Int String
  deriving (Generic)

instance Serialize C
```

You could say that by giving this empty instance we give our blessing to use `genericSerialize` for this type, but we don't have to actually implement anything. And it works:

```haskell
-- $ >>> serialize (C1 42 "yay!")
-- [0,1,42,4,121,97,121,33]
```

Important is that we still have the option to implement instances manually by overwriting the default implementation. This is needed for basic types like `Int` and `Char` that don't have useful generic representations. But it also allows us to overwrite instances for ADTs manually. For example you may want a certain type to be serialized in a special way that deviates from the generic implementation or you may implement an instance manually for performance gain.