Applications of Datatype Generic Programming in Haskell

BOB Konferenz 2016

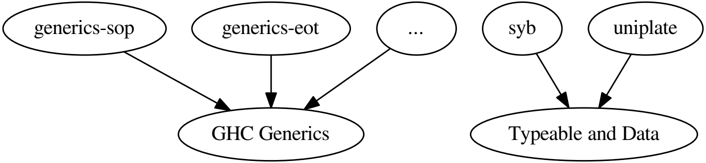
Sönke Hahn

Table of Contents

```haskell
{-# LANGUAGE DeriveAnyClass #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE InstanceSigs #-}
{-# LANGUAGE ScopedTypeVariables #-}

{-# OPTIONS_GHC -fno-warn-missing-methods #-}

module Slides where

import Data.Aeson
import Data.Aeson.Encode.Pretty
import Data.String.Conversions
import Data.Swagger
import Generics.Eot
import Text.Show.Pretty
import WithCli
import qualified Data.ByteString.Lazy.Char8 as LBS
```

Motivation

- The classical example for Datatype Generic Programming (DGP) is serialization / deserialization.
- Demonstration of `getopt-generics`
- DGP can be used in many more circumstances, similar to reflection.
- This should be explored more.



Disclaimer

The code in this presentation uses `generics-eot`. But I'm biased, because I wrote it. Everything is equally possible with either `generics-sop` or `GHC Generics` and probably other libraries.

How to use generic functions?

```haskell
data User
  = User {
    name :: String,
    age :: Int
  }
  | Anonymous
  deriving (Show, Generic, ToJSON, FromJSON, ToSchema)

-- $ >>> LBS.putStrLn $ encode $ User "paula" 3
-- {"tag":"User","age":3,"name":"paula"}

-- $ >>> let x = "{\"tag\":\"Anonymous\",\"contents\":[]}"
-- >>> eitherDecode (cs x) :: Either String User
-- Right Anonymous
```

```
-- $ >>> let proxy = Proxy :: Proxy User
-- >>> LBS.putStrLn $ encodePretty $ toSchema proxy
-- {
--     "minProperties": 1,
--     "maxProperties": 1,
--     "type": "object",
--     "properties": {
--         "User": {
--             "required": [
--                 "name",
--                 "age"
--             ],
--             "type": "object",
--             "properties": {
--                 "age": {
--                     "maximum": 9223372036854775807,
--                     "minimum": -9223372036854775808,
--                     "type": "integer"
--                 },
--                 "name": {
```

How to write generic functions?

Three Kinds of Generic Functions

- Consuming (e.g. serialization)
- Producing (e.g. deserialization)
- Accessing Meta Information (e.g. creating a JSON schema)

Very often, these three kinds are have to be combined.
Consuming and producing relies on an **isomorphic, generic representation**.

Isomorphic, Generic Representations

There's multiple possible **isomorphic** types for `User`:

```
data User
  = User {
    name :: String,
    age :: Int
  }
  | Anonymous
  deriving (Show, Generic, ToJSON, FromJSON)
```

Probably the shortest:

```
Maybe (Int, String)
```

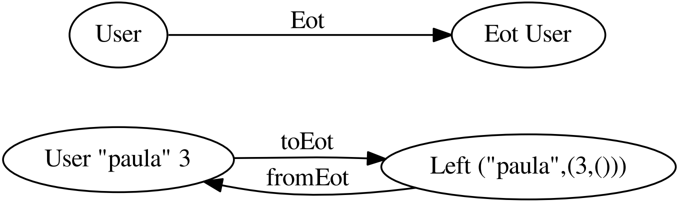
Or:

```
Either (Int, String) ()
```

The one `generics-eot` uses:

```
Either ([Char], (Int, ())) (Either () Void)
```

Mapping to the generic representation: typeclass `HasEot`



- `Eot`: type-level function to map custom ADTs to types of generic representations
- `toEot`: function to convert values in custom ADTs to their generic representation
- `fromEot`: function to convert values in generic representation back to values in the custom ADT

HasEot in action

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

-- >>> toEot $ User "paula" 3
-- Left ("paula",(3,()))

-- >>> fromEot $ Right ()
-- Anonymous
```

End-markers (`()` and `Void`) are needed to unambiguously identify fields. (todo?)

HasEot's method datatype

```
-- $ >>> datatype (Proxy :: Proxy User)
-- Datatype {datatypeName = "User", constructors = [Constr

Datatype {
  datatypeName = "User",
  constructors = [
    Constructor {
      constructorName = "User",
      fields = Selectors ["name", "age"]
    },
    Constructor {
      constructorName = "Anonymous",
      fields = NoFields
    }
  ]
}
```

Example: `getConstructorName`

What we want to implement:

```
-- | returns the name of the used constructor
getConstructorName :: a -> String
```

We start by writing the generic function `eotConstructorName`:

```
class EotConstructorName eot where
  eotConstructorName :: [String] -> eot -> String
```

Example: getConstructorName

Then we need instances for the different possible generic representations. One for `Either x xs`:

```
instance EotConstructorName xs =>
  EotConstructorName (Either x xs) where

  eotConstructorName (name : _) (Left _) = name
  eotConstructorName (_ : names) (Right xs) =
    eotConstructorName names xs
  eotConstructorName _ _ = error "shouldn't happen"
```

Example: getConstructorName

And one for Void to make the compiler happy:

```
instance EotConstructorName Void where
  eotConstructorName :: [String] -> Void -> String
  eotConstructorName _ void =
    seq void $ error "shouldn't happen"
```

Example: getConstructorName

```
getConstructorName :: forall a .
  (HasEot a, EotConstructorName (Eot a)) =>
  a -> String
getConstructorName a =
  eotConstructorName
    (map constructorName $ constructors $
       datatype (Proxy :: Proxy a))
    (toEot a)

-- $ >>> getConstructorName $ User "Paula" 3
-- "User"
-- >>> getConstructorName Anonymous
-- "Anonymous"
```

DGP is a lot like reflection in object-oriented languages. There are some key differences:

- nullable types – libraries using reflection usually need to know, which fields are nullable
- sumtypes/subtypes – both pose problems for lots of use-cases, but also sometimes offer interesting possibilities
- dynamic typing – makes schema generation difficult
- ducktyping – makes it ambiguous which fields are relevant
- type-level computations – types of generic functions can depend on the structure of the datatype, e.g. setting default levels for a database table.

Demonstration

Thank you!

- wiki.haskell.org/Generics
- hackage.haskell.org/package/generic-deriving
- hackage.haskell.org/package/generics-sop
- generics-eot.readthedocs.org/en/latest/