Applications of Datatype Generic Programming in Haskell

BOB Konferenz 2016

Sönke Hahn

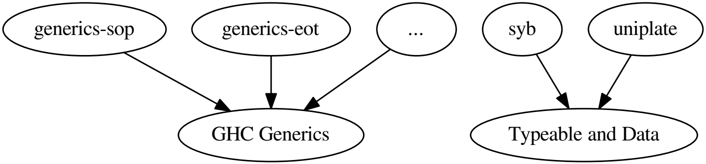
Table of Contents

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

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

module Slides where

import Data.Aeson as Aeson
import Data.Aeson.Encode.Pretty as Aeson.Pretty
import Data.Swagger as Swagger
import Generics.Eot
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.

Generic Libraries



How to use generic functions

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

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

userJson :: LBS.ByteString
userJson = "{\"tag\":\"Anonymous\",\"contents\":[]}"
-- > Aeson.eitherDecode userJson :: Either String User
-- Right Anonymous
```

```haskell
userSwaggerSchema = LBS.putStrLn $
  Aeson.Pretty.encodePretty $
  Swagger.toSchema (Proxy :: Proxy User)
-- > userSwaggerSchema
-- {
--     "minProperties": 1,
--     "maxProperties": 1,
--     "type": "object",
--     "properties": {
--         "User": {
--             "required": [
--                 "name",
--                 "age"
--             ],
--             "type": "object",
--             "properties": {
--                 "age": {
--                     "maximum": 9223372036854775807,
-- ...
```

How to write generic functions

What we want to implement as an example:

```
-- | returns the name of the used constructor
getConstructorName :: (...) => a -> String
```

Three Kinds of Generic Functions

- Accessing Meta Information
- Consuming
- Producing

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

Accessing meta information

(haddocks for 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
    }
  ]
}
```

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 (String, Int)
```

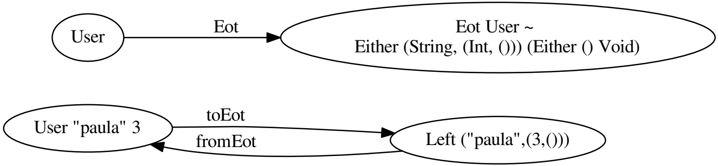
Or:

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

The one `generics-eot` uses:

```
Either (String, (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-marker for fields: ()

If we omit the end-marker:

```
Eot User ~ Either (String, Int) (Either () Void)
```

Consider:

```
data Foo = Foo (String, Int) | Bar
```

We need:

```
Eot User ~ Either (String, (Int, ())) (Either () Void)
```

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

(haddocks for Datatype)

```haskell
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"
```

Comparison to reflection

> [...] reflection is the ability of a computer program to examine [...] and modify its own structure and behavior (specifically the values, meta-data, properties and functions) at runtime.
>
> (From Wikipedia)

Comparison to reflection

- DGP solves very similar problems as reflection in object-oriented languages.
- Unlike reflection, DGP happens (at least in part) at compile time and can statically ensure certain properties of used ADTs, e.g.:
  - Every field is mappable to a database type
  - The ADT has exactly one constructor

Comparison to reflection

- 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 e.g. schema generation difficult
- type-level computations – types of generic functions can depend on the structure of the datatype, e.g. setting default levels for a database table.

Possible applications of DGP

- binary serialization (consuming)
- serialization to JSON (consuming & meta information)
- generating default values (producing)
- generating arbitrary test data (producing)
- database schema generation (meta information)
- database inserts (consuming & meta information)
- command line interfaces (producing & meta information)
- traversals (consuming & producing)
- html forms (producing & meta information)
- parsing configuration files (producing & meta information)
- routing HTTP requests (consuming & meta information)
- etc. . .

Conclusion

- I think of DGP as reflection for Haskell
- DGP supports serialization and deserialization very maturely
- DGP has many other possible applications, lots of them unexplored
- DGP is not **that** complicated and fun!
- Conclusion: You should all go and write generic libraries!

Thank you!

- wiki.haskell.org/Generics
- hackage.haskell.org/package/generic-deriving
- hackage.haskell.org/package/generics-sop
- generics-eot.readthedocs.org/en/latest/
- These slides: github.com/soenkehahn/bobkonf-generics