Autodocodec

A self-documenting encoder and decoder

MHUG Talk 28th July 2022

David Overton & Daniel Chambers

The Problem

- You have some Haskell types that represent Data Transfer Objects for some API
- You want to write round-trippable JSON serialization and deserialization (codec) for each type
- You also want to generate an OpenAPI 3 schema for the types that matches the JSON codecs
- You'd like to add useful documentation

Example

```haskell
data Person = Person
  { personName :: Text,
    personAge :: Int
  }
  deriving (Generic)

instance ToJSON Person
instance FromJSON Person
instance ToSchema Person

aPerson :: Person
aPerson = Person "John Smith" 21
```

`toJSON`

```json
{
    "personAge": 21,
    "personName": "John Smith"
}
```

`toSchema`

```json
{
    "properties": {
        "personAge": { "type": "integer" },
        "personName": { "type": "string" }
    },
    "required": [ "personName", "personAge" ],
    "type": "object"
}
```

Not bad. **But...**

But ...

- What if we want to customize it?
- How can we document our types and fields nicely?
- How can we get more control over the schema generated, including named schema types, references, etc?
- What if we also want JSON Schema / YAML Schema / Swagger 2?
- Can we just write one definition that will automatically do all of this for us?

Introducing Autodocodec

```haskell
data Person = Person
  { personName :: Text,
    personAge :: Int
  }
  deriving (FromJSON, ToJSON, ToSchema) via Autodocodec Person

instance HasCodec Person where
  codec =
    object "Person" $
      Person
        <$> requiredField "name" "The person's name" .= personName
        <*> requiredField "age" "The person's age" .= personAge
```

```
Person {personName = "John Smith", personAge = 21}
```

```
{
    "age": 21,
    "name": "John Smith"
}
```

```
{
    "properties": {
        "age": { "description": "The person's age", "type": "number" },
        "name": { "description": "The person's name", "type": "string" }
    },
    "required": [ "name", "age" ],
    "type": "object"
}
```

Cool! So what else can it do?

Optional fields, more documentation

```
data Person = Person
  { personName :: Text,
    personAge :: Maybe Int
  }
  deriving (FromJSON, ToJSON, ToSchema) via Autodocodec Person

instance HasCodec Person where
  codec =
    object "Person" personCodec
      <?> "An object representing a person"
    where
      personCodec =
        Person
          <$> requiredField "name" "The person's name" .= personName
          <*> optionalField "age" "The person's age" .= personAge
```

```json
{
    "description": "An object representing a person",
    "properties": {
        "age": {
            "description": "The person's age",
            "type": "number"
        },
        "name": {
            "description": "The person's name",
            "type": "string"
        }
    },
    "required": [
        "name"
    ],
    "type": "object"
}
```

Bounded enum - using `Show` and `Bounded` instances

```
data Colour = Red | Green | Blue
  deriving stock (Enum, Bounded)
  deriving (FromJSON, ToJSON, ToSchema) via Autodocodec Colour

instance HasCodec Colour where
  codec = shownBoundedEnumCodec
```

```
{
    "enum": [ "Red", "Green", "Blue" ],
    "type": "string"
}
```

Enum with explicit values

```haskell
data Colour = Red | Green | Blue
  deriving (FromJSON, ToJSON, ToSchema) via Autodocodec Colour

instance HasCodec Colour where
  codec = stringConstCodec [(Red, "red"), (Green, "green"),  (Blue, "blue")]
```

```json
{
    "enum": [ "red", "green", "blue" ],
    "type": "string"
}
```

Newtype

```haskell
newtype Name = Name { unName :: Text }
  deriving (FromJSON, ToJSON, ToSchema) via Autodocodec Name

instance HasCodec Name where
  codec = dimapCodec Name unName textCodec <?> "A name"
```

```json
{
    "description": "A name",
    "type": "string"
}
```

Recursive Sum type

```haskell
data Expression
  = LiteralExpression Int
  | SumExpression Expression Expression
  | ProductExpression Expression Expression
  deriving (FromJSON, ToJSON, ToSchema) via (Autodocodec Expression)
```

```haskell
instance HasCodec Expression where
  codec =
    named "Expression" $ object "Expression" $ discriminatedUnionCodec "type" enc dec
    where
      valueFieldCodec = requiredField' "value"
      lrFieldsCodec = (,) <$> requiredField' "left" .= fst <*> requiredField' "right" .= snd
      enc = \case
        LiteralExpression n -> ("literal", mapToEncoder n valueFieldCodec)
        SumExpression l r -> ("sum", mapToEncoder (l, r) lrFieldsCodec)
        ProductExpression l r -> ("product", mapToEncoder (l, r) lrFieldsCodec)
      dec =
        HashMap.fromList
          [ ( "literal",
              ("LiteralExpression", mapToDecoder LiteralExpression valueFieldCodec)
            ),
            ( "sum",
              ("SumExpression", mapToDecoder (uncurry SumExpression) lrFieldsCodec)
            ),
            ( "product",
              ("ProductExpression", mapToDecoder (uncurry ProductExpression) lrFieldsCodec)
            )
          ]
```

```json
"Expression": {
    "discriminator": {
        "mapping": {
            "literal": "LiteralExpression",
            "product": "ProductExpression",
            "sum": "SumExpression"
        },
        "propertyName": "type"
    },
    "oneOf": [
        {
            "$ref": "#/components/schemas/ProductExpression"
        },
        {
            "$ref": "#/components/schemas/LiteralExpression"
        },
        {
            "$ref": "#/components/schemas/SumExpression"
        }
    ]
}
```

```json
"LiteralExpression": {
    "properties": {
        "type": {
            "enum": [ "literal" ],
            "type": "string"
        },
        "value": { "type": "number" }
    },
    "required": [ "value", "type" ],
    "type": "object"
}
```

```json
"ProductExpression": {
    "properties": {
        "left": {
            "$ref": "#/components/schemas/Expression"
        },
        "right": {
            "$ref": "#/components/schemas/Expression"
        },
        "type": {
            "enum": [ "product" ],
            "type": "string"
        }
    },
    "required": [ "left", "right", "type" ],
    "type": "object"
}
```

```
"SumExpression": {
    "properties": {
        "left": {
            "$ref": "#/components/schemas/Expression"
        },
        "right": {
            "$ref": "#/components/schemas/Expression"
        },
        "type": {
            "enum": [ "sum" ],
            "type": "string"
        }
    },
    "required": [ "left", "right", "type" ],
    "type": "object"
}
```

How does it work?

The `Codec` GADT

```
data Codec context input output where
  ...
```

A `Codec` is a recursive data structure that captures the structure of a data type, along with information about how to construct/destructure it.

- `context`: Used to split the GADT into two parts:
    1. codecs for JSON Values `type ValueCodec = Codec JSON.Value`
    2. codecs for JSON Objects `type ObjectCodec = Codec JSON.Object`
- `input`: The type that this codec can encode
- `output`: The type that this codec can decode

The `HasCodec` typeclass

```haskell
class HasCodec value where
  codec :: ValueCodec value value
```

Types can have an instance of this typeclass when they have a codec that describes how to encode and decode them as a JSON value.

The `Codec` type parameters are therefore set as such:

- `context`: `JSON.Value`
- `input`: `value`
- `output`: `value`

Basic Codecs

These capture the basic JSON data types.

```
NullCodec   :: ValueCodec () ()
BoolCodec   :: {- Name of bool -}   Maybe Text -> ValueCodec Bool Bool
StringCodec :: {- Name of string -} Maybe Text -> ValueCodec Text Text
NumberCodec :: {- Name of number -} Maybe Text -> Maybe NumberBounds -> ValueCodec Scientific Scientific
```

The basic data types have matching `HasClass` instances already. For example:

```
instance HasCodec Text where
  codec = StringCodec Nothing
```

Autodocodec knows how to encode/decode these basic types, so these codecs effectively act as placeholders and contain no encoding/decoding logic.

The ArrayOf Codec

```
ArrayOfCodec ::
  Maybe Text ->                  -- Name of the array, for error messages and doco
  ValueCodec input output -> -- Codec to use with the array elements
  ValueCodec (Vector input) (Vector output)
```

How to encode/decode an array is also built in to Autodocodec. All you need to do is tell it how do encode and decode each of the values (`ValueCodec input output`).

Naturally, a typeclass instance exists to help encode/decode Haskell lists:

```
instance HasCodec a => HasCodec [a] where
  codec = dimapCodec Vector.toList Vector.fromList . ArrayOfCodec Nothing
```

What's that `dimapCodec` nonsense?

```
dimapCodec ::
  (oldOutput -> newOutput) ->
  (newInput -> oldInput) ->
  Codec context oldInput oldOutput ->
  Codec context newInput newOutput
```

The `ArrayOfCodec` is a

`ValueCodec (Vector input) (Vector output)`

but we need a

`ValueCodec [input] [output]`

`dimapCodec` lets us provide mapping functions to convert the Vector to and from a list.

```
codec = dimapCodec Vector.toList Vector.fromList . ArrayOfCodec Nothing
```

The Bimap Codec captures encoding/decoding logic

```
BimapCodec ::
  (oldOutput -> Either String newOutput) -> -- Decoding function
  (newInput -> oldInput) ->                 -- Encoding function
  Codec context oldInput oldOutput ->       -- The old codec
  Codec context newInput newOutput
```

The decoding function is allowed to fail. The encoding function must always succeed.

`dimapCodec` records our mapping functions using a `BimapCodec`:

```
dimapCodec decode encode codec = BimapCodec (Right . decode) encode codec
```

Okay, what about JSON objects?

Let's start talking about `ObjectCodec`s.

Capturing an object property

```
RequiredKeyCodec ::
  Text ->                        -- Property name
  ValueCodec input output ->     -- Codec for the property value
  Maybe Text ->                  -- Doco about the property
  ObjectCodec input output

OptionalKeyCodec ::
  Text ->                        -- Property name
  ValueCodec input output ->     -- Codec for the property value
  Maybe Text ->                  -- Doco about the property
  ObjectCodec (Maybe input) (Maybe output)
```

These two codecs allow us to capture the existence of a property (a "key") on an object, along with how to encode/decode the property value.

`RequiredKeyCodec` is for when the property must exist on the object, `OptionalKeyCodec` is for when it does not need to exist (hence the `Maybe` in the codec's `input` and `output` types).

The ObjectOf Codec

```
ObjectOfCodec ::
  Maybe Text ->                    -- Name of the object
  ObjectCodec input output ->     -- Codec of the object
  ValueCodec input output
```

Once we know how to encode/decode an object, we can capture that as a `ValueCodec` that can en/decode a Value that is an Object by using `ObjectOfCodec`. (A JSON Object is a JSON Value)

Example: A simple JSON object

```json
{ "newZealandText": "simple as, bro" }
```

The matching Haskell type:

```haskell
data NewZealandObject = NewZealandObject { _nzoText :: Text }
```

The codec:

```haskell
instance HasCodec NewZealandObject where
  codec :: ValueCodec NewZealandObject NewZealandObject
  codec =
    RequiredKeyCodec propName propValueCodec doco  -- Make the property codec
      & dimapCodec NewZealandObject _nzoText       -- Map Text type in/outof the record type
      & ObjectOfCodec objectName                   -- Wrap in ObjectOf
    where
      objectName = Just "NewZealandObject"
      propName = "newZealandText"
      propValueCodec = StringCodec Nothing
      doco = Just "Must be said in an NZ accent"
```

Multiple Properties via Applicative

We can see how to capture one property, but in order to capture multiple properties and combine them into a Haskell record, we need Applicatives!

```
ApCodec ::
  ObjectCodec input (output -> newOutput) ->
  ObjectCodec input output ->
  ObjectCodec input newOutput

PureCodec ::
  output ->
  ObjectCodec void output
```

```
instance Applicative (ObjectCodec input) where
  pure = PureCodec
  (<*>) = ApCodec
```

A key observation is that the applicative instance is only over the `output` type parameter and not `input`!

Example: JSON object with multiple properties (Step 1)

```
{ "firstName": "Daniel", "lastName": "Chambers" }
```

```
data FullName = FullName { _fnFirstName :: Text, _fnLastName :: Text }

instance HasCodec FullName where
  codec :: ValueCodec FullName FullName
  codec =
    ObjectOfCodec (Just "FullName") $
      FullName
        <$> firstNameCodec -- ERROR! Applicative changes the output type param,
        <*> lastNameCodec  --        but not the input type param!
    where
      firstNameCodec :: ObjectCodec Text Text
      firstNameCodec = RequiredKeyCodec "firstName" textCodec noDoco

      lastNameCodec :: ObjectCodec Text Text
      lastNameCodec = RequiredKeyCodec "lastName" textCodec noDoco

      textCodec = StringCodec Nothing
      noDoco = Nothing
```

Example: JSON object with multiple properties (Step 2)

```
{ "firstName": "Daniel", "lastName": "Chambers" }
```

```
data FullName = FullName { _fnFirstName :: Text, _fnLastName :: Text }

instance HasCodec FullName where
  codec :: ValueCodec FullName FullName
  codec =
    ObjectOfCodec (Just "FullName") $
      FullName
        <$> firstNameCodec
        <*> lastNameCodec
    where
      firstNameCodec :: ObjectCodec FullName Text
      firstNameCodec =
        dimapCodec id _fnFirstName $ RequiredKeyCodec "firstName" textCodec noDoco

      lastNameCodec :: ObjectCodec FullName Text
      lastNameCodec =
        dimapCodec id _fnLastName $ RequiredKeyCodec "lastName" textCodec noDoco

      textCodec = StringCodec Nothing
      noDoco = Nothing
```

Example: JSON object with multiple properties (Step 3)

```json
{ "firstName": "Daniel", "lastName": "Chambers" }
```

```haskell
data FullName = FullName { _fnFirstName :: Text, _fnLastName :: Text }

instance HasCodec FullName where
  codec :: ValueCodec FullName FullName
  codec =
    object "FullName" $
      FullName
        <$> requiredField' "firstName" .= _fnFirstName
        <*> requiredField' "lastName" .= _fnLastName
```

- `object` makes our `ObjectOfCodec` with a name
- `requiredField'` makes our `RequiredKeyCodec` (with no documentation)
- `.=` maps our input type parameter

Much cleaner!

What about sum types?

The Either Codec

```
EitherCodec ::
  Union ->                        -- Do the types overlap or not?
  Codec context input1 output1 -> -- Codec for the first alternative
  Codec context input2 output2 -> -- Codec for the second alternative
  Codec context (Either input1 input2) (Either output1 output2)
```

The Either Codec allows us to capture the alternative between two codecs. When decoding, the first codec is tried first, and then the second is tried.

`Union` controls whether we allow the encoded representations to overlap. If we declare it to be a `DisjointUnion`, then decoding fails if both decoders succeed. `PossiblyJointUnion` allows us to simply accept the first that is successfully decoded.

Example: Accept either Text or Number

```haskell
data TextOrNumber
  = Text Text
  | Number Scientific

instance HasCodec TextOrNumber where
  codec :: ValueCodec TextOrNumber TextOrNumber
  codec =
    dimapCodec decode encode stringOrNumber
    where
      stringOrNumber :: ValueCodec (Either Text Scientific) (Either Text Scientific)
      stringOrNumber =
        EitherCodec DisjointUnion (StringCodec Nothing) (NumberCodec Nothing Nothing)

      decode :: Either Text Scientific -> TextOrNumber
      decode = \case
        Left txt -> Text txt
        Right sci -> Number sci

      encode :: TextOrNumber -> Either Text Scientific
      encode = \case
        Text txt -> Left txt
        Number sci -> Right sci
```

Hardcoding Values - The Eq Codec

Sometimes we have a known discrete value we want to use in the encoding/decoding. For example, enums are a set of known discrete values.

```
EqCodec ::
  (Show value, Eq value) =>
  value ->                  -- Value to match
  ValueCodec value value -> -- Codec for the value
  ValueCodec value value
```

The `EqCodec` allows us to capture a discrete value in our encoding/decoding structure and only succeed at decoding if that particular value is matched.

Example: Yes/No enum (Step 1)

```haskell
data YesNo = Yes | No

instance HasCodec YesNo where
  codec =
    dimapCodec decode encode $ EitherCodec DisjointUnion yes no
    where
      yes :: ValueCodec Text YesNo
      yes = dimapCodec (const Yes) id $ EqCodec "Yes" (StringCodec Nothing)

      no :: ValueCodec Text YesNo
      no = dimapCodec (const No) id $ EqCodec "No" (StringCodec Nothing)

      decode :: Either YesNo YesNo -> YesNo
      decode = either id id

      encode :: YesNo -> Either Text Text
      encode = \case
        Yes -> Left "Yes"
        No -> Right "No"
```

Example: Yes/No enum (Step 2)

```
data YesNo = Yes | No
  deriving stock (Eq, Show, Bounded, Enum)

instance HasCodec YesNo where
  codec = shownBoundedEnumCodec
```

Much simpler 😉

So how is all this used to produce JSON Serialization and OpenAPI Schema?

Basically: walk the assembled data structure of codecs and based on what you encounter, perform JSON encoding/decoding or create an OpenAPI schema!

Let's look at a snippet to get a sense of it.

EqCodec

JSON Decoding

```
EqCodec expectedValue valueCodec -> do
  actualValue <- go inputValue valueCodec
  if expectedValue == actualValue
    then pure actualValue
    else fail $ unwords ["Expected", show expectedValue, "but got", show actualValue]
--------------------------------------------------------------------------------------------------------------
```

OpenAPI Schema

```
EqCodec expectedValue valueCodec ->
  pure $
    NamedSchema Nothing $
      let jsonVal = toJSONVia valueCodec expectedValue
        in mempty
            { _schemaEnum = Just [jsonVal],
              _schemaType = Just $ case jsonVal of
                Aeson.Object {} -> OpenApiObject
                Aeson.Array {} -> OpenApiArray
                Aeson.String {} -> OpenApiString
                Aeson.Number {} -> OpenApiNumber
                Aeson.Bool {} -> OpenApiBoolean
                Aeson.Null -> OpenApiNull
            }
--------------------------------------------------------------------------------------------------------------
```