aeson-schemas: Safely extract JSON data when data types are too cumbersome[1]

Brandon Chinn

1 August 2020

[1] http://hackage.haskell.org/package/aeson-schemas

Agenda

- Motivation (5 min)
- Using `aeson-schemas` (10 min)
- Implementing `aeson-schemas` (10 min)
  - Type-level programming 101
- Final Thoughts (5 min)

Motivation

Motivation

Parsing data with aeson

```
class ToJSON a where
  toJSON :: a -> Value

class FromJSON a where
  parseJSON :: Value -> Parser a
```

Motivation

Parsing data with aeson

```json
{
    "users": [
        {
            "id": 1,
            "name": "Alice"
        },
        ...
    ]
}
```

```haskell
data User = User
  { id   :: Int
  , name :: String
  }
  deriving
    ( Show
    , Generic
    , FromJSON
    )
```

Motivation

Parsing data with aeson

```json
{
  "name": "Policy1",
  "permissions": [
    {
      "resource": {
        "name": "secretdata.txt",
        "owner": "john@example.com"
      },
      "access": "READ"
    }
  ]
}
```

```haskell
data Result = Result
  { name        :: String
  , permissions :: [Permission]
  }

data Permission = Permission
  { resource :: Maybe Resource
  , access   :: String
  }

data Resource = Resource
  { name  :: String
  , owner :: Maybe String
  }
```

Motivation

Querying a GraphQL API

```
type Query {
    users: [User!]!
}
type User {
    id: ID!
    name: String!
    posts: [Post!]!
}
type Post {
    id: ID!
    name: String!
    createdAt: String!
}
```

```
query {
    users {
        id
        name
        posts {
            id
            name
        }
    }
}
```

Motivation

Querying a GraphQL API

```haskell
data Query = Query
  { users :: Maybe [User]
  }

data User = User
  { id    :: Maybe String
  , name  :: Maybe String
  , posts :: Maybe [Post]
  }

data Post = Post
  { id        :: Maybe ID
  , name      :: Maybe String
  , createdAt :: Maybe String
  }
```

- Pros
  - Direct translation of GraphQL schema
- Cons
  - Handle `Nothing` / use `fromJust`
  - `id` field name shadows `Prelude.id`
  - Duplicate name field

Motivation

Querying a GraphQL API

```haskell
data Query1 = Query1
  { users :: [User1]
  }

data User1 = User1
  { id    :: String
  , name  :: String
  , posts :: [Post1]
  }

data Post1 = Post1
  { id   :: String
  , name :: String
  }
```

- Pros
  - No more `Maybe`
- Cons
  - Redefine type per use
  - Record names still duplicated

Problem Requirements

1. Type safe
2. Avoid polluting namespace
3. Nice query language

Using `aeson-schemas`

Using aeson-schemas

```haskell
import Data.Aeson.Schema (schema)

type MySchema = [schema|
  {
    users: List {
      id: Int,
      name: Text,
    },
  }
|]
```

```haskell
import Data.Aeson (decodeFileStrict)
import Data.Aeson.Schema (Object, get)

obj <- fromJust <$>
  decodeFileStrict "example.json"
    :: IO (Object MySchema)

-- outputs:
-- ["Alice", "Bob", "Claire"]
print [get| obj.users[].name |]
```

Using aeson-schemas

schema quasiquoter

```
type BasicSchema = [schema|
  {
    a: Bool,
    b: Int,
    c: Double,
    d: Text,
    e: UTCTime,
  }
|]
```

```
type ComplexSchema = [schema|
  {
    foo: List {
      a: Int,
      b: Maybe Text,
    },
    bar: List Maybe Bool,
  }
|]
```

Using aeson-schemas

get quasiquoter

```
let users = [get| obj.a.b.users |]

map [get| .name |] users

-- compare:
--    map (fmap c . b) (a obj)
[get| obj.a[].b?.c |]
```

Using aeson-schemas

GraphQL query

```
query {
    users {
        id
        name
        posts {
            id
            name
        }
    }
}
```

aeson-schemas schema

```
type Query1 = [schema|
  {
    users: List {
      id: Text,
      name: Text,
      posts: List {
        id: Text,
        name: Text,
      },
    },
  }
|]
```

Implementing `aeson-schemas`

Type-level programming

Value	**Type**	**Kind**[2]
`True`, `False`	`Bool`	`*`
`Just 1`, `Nothing`	`Maybe Int`	`*`
N/A	`Maybe`	`* -> *`

[2] `*` is actually deprecated in favor of `Type` from `Data.Kind`, but I like how `*` looks better, so that's why I'm using it.

Type-level programming

With -XDataKinds

Value	Type	Kind
`True`, `False`	`Bool`	`*`
N/A	`'True`, `'False`	`Bool`

Type-level programming

Demo: `Restaurant.hs`

Type-level programming

Type families

```
type family Foo a where
  Foo Int = [Int]
  Foo Bool = Maybe Bool

x :: Foo Int
x = [1, 2, 3]

y :: Foo Bool
y = Just True
```

Implementing `aeson-schemas`

1. Define the schema
2. Parse JSON data into `Object`
3. Extract data from `Object`

Implementing `aeson-schemas`

Defining the schema

```haskell
import GHC.TypeLits (Symbol)

data SchemaType
  = SchemaInt
  | SchemaText
  | SchemaList SchemaType
  | SchemaObject [(Symbol, SchemaType)]
```

Implementing aeson-schemas

Defining the schema

```haskell
{-# LANGUAGE DataKinds #-}

type MySchema = 'SchemaObject
  '[ '( "users"
      , 'SchemaList (
          'SchemaObject
            '[ '("id", 'SchemaInt)
             , '("name", 'SchemaText)
             ]
        )
      )
   ]
```

Implementing aeson-schemas

Parsing data into Object

```haskell
data Object (schema :: SchemaType) = UnsafeObject (HashMap Text Dynamic)

instance (IsSchemaType schema, SchemaResult schema ~ Object schema)
    => FromJSON (Object schema) where
  parseJSON = parseValue @schema

type family SchemaResult (schema :: SchemaType) where
  SchemaResult 'SchemaInt = Int
  SchemaResult 'SchemaText = Text
  SchemaResult ('SchemaList inner) = [SchemaResult inner]
  SchemaResult ('SchemaObject schema) = Object ('SchemaObject schema)

class IsSchemaType (schema :: SchemaType) where
  parseValue :: Value -> Parser (SchemaResult schema)
```

Implementing `aeson-schemas`

Parsing data into `Object`

```haskell
instance IsSchemaType 'SchemaInt where
  parseValue = Aeson.parseJSON -- :: Value -> Parser Int

instance IsSchemaType 'SchemaText where
  parseValue = Aeson.parseJSON -- :: Value -> Parser Text

instance IsSchemaType inner => IsSchemaType ('SchemaList inner) where
  parseValue (Aeson.Array a) = traverse (parseValue @inner) (Vector.toList a)
  parseValue _ = fail "..."
```

Implementing aeson-schemas

Parsing data into Object

```
-- ref: SchemaObject [(Symbol, SchemaType)]
instance (...) => IsSchemaType ('SchemaObject ('(key, inner) ': rest)) where
  parseValue value@(Aeson.Object o) = do
    let key = Text.pack $ symbolVal (Proxy @key)

    inner <- parseValue @inner (HashMap.lookupDefault Aeson.Null key o)

    UnsafeObject rest <- parseValue @rest value

    return $ UnsafeObject $ HashMap.insert key (toDyn inner) rest

  parseValue _ = fail "..."

instance IsSchemaType ('SchemaObject '[]) where
  parseValue (Aeson.Object _) = return $ UnsafeObject HashMap.empty
  parseValue _ = fail "..."
```

Implementing aeson-schemas

Extracting data from Object

```
let o :: Object ('SchemaObject '[ '("foo", 'SchemaInt) ])
    o = ...

getKey @"foo" o :: Int
```

Implementing aeson-schemas

Extracting data from Object

```haskell
-- Fcf.Lookup    :: a -> [(a, b)] -> Fcf.Exp (Maybe b)
-- Fcf.FromMaybe :: a -> Maybe a -> Fcf.Exp a
-- Fcf.=<<       :: (a -> Fcf.Exp b) -> Fcf.Exp a -> Fcf.Exp b
-- Fcf.Eval      :: Fcf.Exp a -> a

type family LookupSchema (key :: Symbol) (schema :: SchemaType) where
  LookupSchema key ('SchemaObject schemaTypeMap) = Fcf.Eval (
    Fcf.FromMaybe (
        TypeError (
              'Text "Key '" ':<>: 'Text key
        ':<>: 'Text "' does not exist in the following schema:"
        ':$$: 'ShowType schemaTypeMap )
    ) =<< Fcf.Lookup key schemaTypeMap )
```

Implementing aeson-schemas

Extracting data from Object

```
getKey
  :: forall key initialSchema. (...)
  => Object initialSchema -> SchemaResult (LookupSchema key initialSchema)
getKey (UnsafeObject o) =
  fromMaybe (error "This should not happen") $
    fromDynamic (o ! Text.pack key)
  where
    key = symbolVal (Proxy @key)
```

Implementing aeson-schemas

```haskell
type MySchema = 'SchemaObject
  '[ '( "users",
        'SchemaList (
          'SchemaObject '[ '("id", 'SchemaInt), '("name", 'SchemaText) ]
         )
      )
   ]

o <- fromJust <$> decodeFileStrict "example.json" :: IO (Object MySchema)

let names :: [Text]
    names = map (getKey @"name") $ getKey @"users" o
```

Final Thoughts

Thank You

LEAPYEAR

https://leapyear.io

Q & A