PureScript-Simple-JSON Documentation

Justin Woo

Aug 13, 2019

Contents

1 Pages **3**

1.1 Introduction . . . 3
1.2 Quickstart . . . 4
1.3 Working with Inferred Record Types . . . 6
1.4 Usage with Affjax . . . 9
1.5 Usage with Generics-Rep . . . 10
1.6 FAQ . . . 13

This is a guide for the PureScript library Simple-JSON, which provides an easy way to decode either `Foreign` (JS) values or JSON `String` values with the most "obvious" representation. This guide will also try to guide you through some of the details of how PureScript the language works, as you may be new to PureScript or not know its characteristics.

Overall, this library provides you largely automatic ways to get decoding, but does not try to decode any types that do not actually have a JS representation. This means that this library does not provide you with automatic solutions for decoding and encoding Sum or Product types, but there is a section below on how to use Generics in PureScript to achieve the encoding of Sum and Product types that you want.

Tip: If you are coming from Elm, you can think of this library as providing the automatic encoding/decoding of ports, but actually giving you explicit control of the results and allowing you to define encodings as you need.

Note: If there is a topic you would like more help with that is not in this guide, open a issue in the Github repo for it to request it.

CHAPTER 1

Pages

1.1 Introduction

1.1.1 What is `Foreign`?

In PureScript, untyped JS values are typed as `Foreign` and are defined in the Foreign library. Usually when you define FFI functions, you should define the results of the functions as `Foreign` and then decode them to a type if you want to ensure safety in your program.

For example, this library exposes the method parseJSON with the type

```
parseJSON :: String -> F Foreign
```

We'll visit what this `F` failure type is later, since you won't need to use it most of the time when you use this library.

1.1.2 How you should use this library

Generally, you should try to separate your transport types from your domain types such that you never try to tie down the model used in your program to whatever can be represented in JS. For example, a sum type

```
data IsRegistered
  = Registered DateString
  | NotRegistered
```

is the correct model to use in your program, while the transport may be defined

```
type RegistrationStatus =
  { registrationDate :: Maybe DateString
  }
```

While you `could` use `Maybe DateString` all over your application, this type suffers in that there is just not much information for your users to take from this type. If you used a newtype of this, the actual matching usages would still suffer the same problem.

1.1.3 On Sum Types

Many users complain that Simple-JSON should provide automatic serialization of sum types, but you'll find that preferred encodings for sum types are like opinions – everyone has one. Instead of giving you a default that wouldn't make sense in the scope of Simple-JSON as providing decoding for JS-representable types, we'll go over how PureScript's Generics-Rep work and how easy it is for you to work with sum types with your preferred methods.

1.2 Quickstart

1.2.1 Decoding / Reading JSON

Simple-JSON can be used to easily decode from types that have JSON representations, such as numbers, booleans, strings, arrays, and records.

Let's look at an example using a record alias:

```
type MyRecordAlias =
  { apple :: String
  , banana :: Array Int
  }
```

Now we can try decoding some JSON:

```
import Simple.JSON as JSON

testJSON1 :: String
testJSON1 = """
{ "apple": "Hello"
, "banana": [ 1, 2, 3 ]
}
"""

main = do
  case JSON.readJSON testJSON1 of
    Right (r :: MyRecordAlias) -> do
      assertEqual { expected: r.apple, actual: "Hello"}
      assertEqual { expected: r.banana, actual: [ 1, 2, 3 ] }
    Left e -> do
      assertEqual { expected: "failed", actual: show e }
```

Since `JSON.readJSON` returns `Either MultipleErrors a`, we need to provide the compiler information on what type the `a` should be. We accomplish this by establishing a concrete type for `a` with the type annotation `r :: MyRecordAlias`, so the return type is now `Either MultipleErrors MyRecordAlias`, which is the same as `Either MultipleErrors { apple :: String, banana :: Array Int }`.

And that's it!

1.2.2 Encoding / Writing JSON

Encoding JSON is a failure-proof operation, since we know what we want to encode at compile time.

```
main = do
  let
    myValue =
```

(continues on next page)

(continued from previous page)

```
      { apple: "Hi"
      , banana: [ 1, 2, 3 ]
      } :: MyRecordAlias

  log (JSON.writeJSON myValue) -- {"banana":[1,2,3],"apple":"Hi"}
```

And that's all we need to do to encode JSON!

1.2.3 Working with Optional values

For most cases, the instance for `Maybe` will do what you want by decoding `undefined` and `null` to `Nothing` and writing `undefined` from `Nothing` (meaning that the JSON output will not contain the field).

```
type WithMaybe =
  { cherry :: Maybe Boolean
  }

testJSON3 :: String
testJSON3 = """
{ "cherry": true
}
"""

testJSON4 :: String
testJSON4 = """
{}
"""
```

```
main = do
  case JSON.readJSON testJSON3 of
    Right (r :: WithMaybe) -> do
      assertEqual { expected: Just true, actual: r.cherry }
    Left e -> do
      assertEqual { expected: "failed", actual: show e }

  case JSON.readJSON testJSON4 of
    Right (r :: WithMaybe) -> do
      assertEqual { expected: Nothing, actual: r.cherry }
    Left e -> do
      assertEqual { expected: "failed", actual: show e }

  let
    withJust =
      { cherry: Just true
      } :: WithMaybe
    withNothing =
      { cherry: Nothing
      } :: WithMaybe

  log (JSON.writeJSON withJust) -- {"cherry":true}
  log (JSON.writeJSON withNothing) -- {}
```

If you explicitly need `null` and not `undefined`, use the `Nullable` type.

```
main =
  case JSON.readJSON testJSON3 of
    Right (r :: WithNullable) -> do
      assertEqual { expected: toNullable (Just true), actual: r.cherry }
    Left e -> do
      assertEqual { expected: "failed", actual: show e }

  case JSON.readJSON testJSON4 of
    Right (r :: WithNullable) -> do
      assertEqual { expected: "failed", actual: show r }
    Left e -> do
      let errors = Array.fromFoldable e
      assertEqual { expected: [ErrorAtProperty "cherry" (TypeMismatch "Nullable␣
↪Boolean" "Undefined")], actual: errors }

  let
    withNullable =
      { cherry: toNullable Nothing
      } :: WithNullable
  log (JSON.writeJSON withNullable) -- {"cherry":null}
```

1.3 Working with Inferred Record Types

1.3.1 How records work in PureScript

In PureScript, a `Record` type is parameterized by `# Type`

```
data Record :: # Type -> Type
```

As seen on Pursuit, this means that records are an application of row types of `Type`, such that the two definitions are equivalent:

```
type Person = { name :: String, age :: Number }

type Person = Record ( name :: String, age :: Number )
```

With this knowledge, we can work with records in a generic way where any operation with the correct row type constraints is valid.

This is unlike other languages where records are often simply product types with selector information. Let's look at some examples of this at work.

1.3.2 Modifying a field's type

Say that we wanted to read in JSON into this type:

```
type RecordWithEither =
  { apple :: Int
  , banana :: Either Int String
  }
```

We know that there's no representation of this `Either Int String` in JavaScript, but it would be convenient to read some value into it. First, let's define a function to read in any `Either`:

```
readEitherImpl
  :: forall a b
   . JSON.ReadForeign a
  => JSON.ReadForeign b
  => Foreign
  -> Foreign.F (Either a b)
readEitherImpl f
    = Left <$> JSON.readImpl f
  <|> Right <$> JSON.readImpl f
```

Now we can read in to an `Either` any `a` and `b` that have instances for `ReadForeign`. We can then use this to modify a field in an inferred context:

```
readRecordWithEitherJSON :: String -> Either Foreign.MultipleErrors RecordWithEither
readRecordWithEitherJSON s = runExcept do
  inter <- JSON.readJSON' s
  banana <- readEitherImpl inter.banana
  pure $ inter { banana = banana }
```

So what goes on here is that since the result of the function is our `RecordWithEither` with a field of `banana :: Either Int String`, the type is inferred "going backwards", so with the application of our function that is now concretely typed in this context as `readEitherImpl :: Foreign -> Foreign.F (Either Int String)`, the `inter` is read in as `{ apple :: Int, banana :: Foreign }`.

In this case, we used record update syntax to modify our inferred record, but we also could have done this generically using `Record.modify` from the Record library.

1.3.3 PureScript-Record in a nutshell

Most of PureScript-Record revolves around usages of two row type classes from Prim.Row:

```
class Cons
  (label :: Symbol) (a :: Type) (tail :: # Type) (row :: # Type)
  | label a tail -> row, label row -> a tail

class Lacks
  (label :: Symbol) (row :: # Type)
```

`class Cons` is a relation of a field of a given `Symbol` label (think type-level `String`), its value `Type`, a row type `tail`, and a row type `row` which is made of the `tail` and the field put together. This is very much like your normal `List` of `Cons a` and `Nil`, but with the unordered row type structure at the type level (that `(a :: String, b :: Int)` is equivalent to `(b :: Int, a :: String)`).

`class Lacks` is a relation of a given `Symbol` label not existing in any of the fields of `row`.

With this bit of knowledge, we can go ahead and look at the docs of the Record library.

Let's go through a few of these. First, `get`:

```
get
  :: forall r r' l a
   . IsSymbol l
  => Cons l a r' r
  => SProxy l
  -> { | r }
  -> a
```

So here right away we can see that the `Cons` constraint is used to declare that the label `l` provided by the `SProxy` argument must exist in the row type `r`, and that there exists a `r'`, a complementary row type, which is `r` but without the field `l, a`. With this, this function is able to get out the value of type `a` at label `l`. This function doesn't know what concrete label is going to be used, but it uses this constraint to ensure that the field exists in the record.

```
insert
  :: forall r1 r2 l a
   . IsSymbol l
  => Lacks l r1
  => Cons l a r1 r2
  => SProxy l
  -> a
  -> { | r1 }
  -> { | r2 }
```

With `insert`, we work with the input row type `r1` and the output row type `r2`. The constraints here work that the `r1` row should not contain a field with label `l`, and that the result of adding a field of `l, a` to `r1` yields `r2`.

Now, the most involved example:

```
rename
  :: forall prev next ty input inter output
   . IsSymbol prev
  => IsSymbol next
  => Cons prev ty inter input
  => Lacks prev inter
  => Cons next ty inter output
  => Lacks next inter
  => SProxy prev
  -> SProxy next
  -> { | input }
  -> { | output }
```

Because PureScript does not solve multiple constraints simultaneously, we work with three row types here: `input`, `inter` (intermediate), and `output`. This function takes two `Symbol` types: one for the current label of the field and one for the next label. Then the constraints work such that `inter` is `input` without the field `prev, ty` and lacks any additional fields of `prev`, as row types can have duplicate labels as they are not only for records. Then `output` is constructured by adding the field `next, ty` to `inter` and checking that the `inter` does not already contain a field with the label `next`. While this seems complicated at first, slowly reading through the constraints will show that this is a series of piecewise operations instead of being a multiple-constraint system.

1.3.4 Application of generic Record functions

Say we have a type where we know the JSON will have the wrong name:

```
type RecordMisnamedField =
  { cherry :: Int
  }
```

If the JSON we receive has this field but with the name "grape", what should we do?

We can apply the same inferred record type method as above but with `Record.rename`:

```
readRecordMisnamedField :: String -> Either Foreign.MultipleErrors RecordMisnamedField
readRecordMisnamedField s = do
  inter <- JSON.readJSON s
```

(continues on next page)

(continued from previous page)

```
  pure $ Record.rename grapeP cherryP inter
  where
    grapeP = SProxy :: SProxy "grape"
    cherryP = SProxy :: SProxy "cherry"
```

So again, by applying a function that renames `grape, Int` to `cherry, Int`, the inferred record type of the `inter` is `{ grape :: Int }` and that is the type used to decode the JSON.

Hopefully this page has shown you how powerful row type based Records are in PureScript and the generic operations they allow.

You might be interested in reading through slides for further illustrations of how generic record operations work and how they can be used with Simple-JSON.

1.4 Usage with Affjax

There is an issue that discusses how usage with Affjax goes here: https://github.com/justinwoo/purescript-simple-json/issues/51

1.4.1 Manually

In short, you can use the `string` response format for the request:

```
import Prelude

import Affjax (get)
import Affjax.ResponseFormat (ResponseFormatError(..), string)
import Data.Bifunctor (lmap)
import Data.Either (Either(..))
import Data.List.NonEmpty (singleton)
import Effect.Aff (launchAff_)
import Effect.Class.Console (log)
import Simple.JSON (readJSON)

type MyRecordAlias = { userId :: Int }

main = void $ launchAff_ $ do
  res <- get string "https://jsonplaceholder.typicode.com/todos/1"
  case lmap transformError res.body >>= readJSON of
    Right (r :: MyRecordAlias) -> do
      log "all good"
    Left e -> do
      log "all bad"

transformError (ResponseFormatError e _) = singleton e
```

1.4.2 With Simple-Ajax

You can use Dario's library for making requests with Affjax and handling errors with Variant here: https://github.com/darioooddenino/purescript-simple-ajax

1.5 Usage with Generics-Rep

1.5.1 Motivation

If you really want to work with sum types using Simple-JSON, you will have to define instances for your types accordingly. Normally, this would mean that you would have to define a bunch of instances manually. For example,

```
data IntOrBoolean
  = Int Int
  | Boolean Boolean

instance readForeign :: JSON.ReadForeign IntOrBoolean where
  readImpl f
      = Int <$> Foreign.readInt f
    <|> Boolean <$> Foreign.readBoolean f
```

But this ends up with us needing to maintain a mountain of error-prone boilerplate, where we might forget to include a constructor or accidentally have duplicate cases. We should be able to work more generically to write how instances should be created once, and then have all of these instances created for us for free.

This is the idea of using datatype generics, which are provided by the Generics-Rep library in PureScript.

1.5.2 Generics-Rep in short

Since what makes Generics-Rep work is in the PureScript compiler as a built-in derivation, you can read through its source to get the gist of it: Link

So once you've skimmed through that, let's first look at `class Generic`:

```
class Generic a rep | a -> rep where
  to :: rep -> a
  from :: a -> rep
```

The functional dependencies here declare that instances of `Generic` are determined by the type given, so only `a` needs to be known to get `rep`. Then we have a method for turning the representation into our type with `to` and our type into a representation with `from`. This means that if we define a function that can produce a `F rep` from decoding `Foreign` in our `JSON.ReadForeign` instances, we can map the `to` function to it to get `F a`. We'll see how that works later.

If some of this isn't familiar to you, you should read about type classes from some source like PureScript By Example

Then, let's look at some of the most relevant representation types:

```
-- | A representation for types with multiple constructors.
data Sum a b = Inl a | Inr b

-- | A representation for constructors which includes the data constructor name
-- | as a type-level string.
newtype Constructor (name :: Symbol) a = Constructor a

-- | A representation for an argument in a data constructor.
newtype Argument a = Argument a
```

These will be the main types that will need to write instances for when we define a type class to do some generic decoding. These correspond to the following parts of a definition:

```
data Things = Apple Int |    Banana String
--            a          Sum b
-- e.g. Sum (Inl a) (Inr b)

data Things = Apple              Int | Banana String
--            Constructor(name) a
-- e.g. Constructor "Apple" a

data Things = Apple Int | Banana String
--                  Argument(a)
-- e.g. Argument Int
```

This diagram probably won't be that useful the first time you read it, but you may find it to be nice to return to.

You can read more coherent explanations like in the documentation for GHC Generics in generics-deriving

1.5.3 Applying Generics-Rep to decoding untagged JSON values

Let's revisit the `IntOrBoolean` example, but this time by using Generics-Rep.

```
import Data.Generic.Rep as GR
import Data.Generic.Rep.Show (genericShow)

data IntOrBoolean2
  = Int2 Int
  | Boolean2 Boolean

-- note the underscore at the end for the `rep` parameter of class Generic
derive instance genericIntOrBoolean2 :: GR.Generic IntOrBoolean2 _

instance showIntOrBoolean2 :: Show IntOrBoolean2 where
  show = genericShow
  -- now we get a Show based on Generic

instance readForeignIntOrBoolean2 :: JSON.ReadForeign IntOrBoolean2 where
  readImpl f = GR.to <$> untaggedSumRep f
  -- as noted above, mapping to so that we go from F rep to F IntOrBoolean

class UntaggedSumRep rep where
  untaggedSumRep :: Foreign -> Foreign.F rep
```

So with our class `UntaggedSumRep`, we have our method `untaggedSumRep` for decoding `Foreign` to `rep`.

Once we have this code, we'll get some errors about missing instances for `Sum`, `Constructor`, and `Argument` as expected.

First, we define our `Sum` instance so we take the alternative of a `Inl` construction and `Inr` construction:

```
instance untaggedSumRepSum ::
  ( UntaggedSumRep a
  , UntaggedSumRep b
  ) => UntaggedSumRep (GR.Sum a b) where
  untaggedSumRep f
      = GR.Inl <$> untaggedSumRep f
    <|> GR.Inr <$> untaggedSumRep f
```

And in our instance we have clearly constrained `a` and `b` for having instances of `UntaggedSumRep`, so that we can use `untaggedSumRep` on the members.

Then we define our `Constructor` instance:

```
instance untaggedSumRepConstructor ::
  ( UntaggedSumRep a
  ) => UntaggedSumRep (GR.Constructor name a) where
  untaggedSumRep f = GR.Constructor <$> untaggedSumRep f
```

This definition similar to above, but just with our single constructor case.

This is where you would try reading `f` into a record by doing something like `record :: { tag :: String, value :: Foreign } <- f` in a do block, if you wanted to represent sum types in that way. Sky's the limit!

Then let's define the argument instance that will call `readImpl` on the `Foreign` value.

```
instance untaggedSumRepArgument ::
  ( JSON.ReadForeign a
  ) => UntaggedSumRep (GR.Argument a) where
  untaggedSumRep f = GR.Argument <$> JSON.readImpl f
```

And so at this level, we try to decode the `Foreign` value directly to the type of the argument.

With just these few lines of code, we now have generic decoding for our untagged sum type encoding that we can apply to any sum type where `Generic` is derived and the generic representation contains `Sum`, `Constructor`, and `Argument`. To get started with your own instances, check out the example in test/Generic.purs in the Simple-JSON repo.

1.5.4 Working with "Enum" sum types

If you have sum types where all of the constructors are nullary, you may want to work with them as string literals. For example:

```
data Fruit
  = Abogado
  | Boat
  | Candy
derive instance genericFruit :: Generic Fruit _
```

Like the above, we should write a function that can work with the generic representation of sum types, so that we can apply this to all enum-like sum types that derive `Generic` and use it like so:

```
instance fruitReadForeign :: JSON.ReadForeign Fruit where
  readImpl = enumReadForeign

enumReadForeign :: forall a rep
   . Generic a rep
  => EnumReadForeign rep
  => Foreign
  -> Foreign.F a
enumReadForeign f =
  to <$> enumReadForeignImpl f
```

First, we define our class which is take the rep and return a `Foreign.F rep`:

```
class EnumReadForeign rep where
  enumReadForeignImpl :: Foreign -> Foreign.F rep
```

Then we only need two instance for this class. First, the instance for the `Sum` type to split cases:

```
instance sumEnumReadForeign ::
  ( EnumReadForeign a
  , EnumReadForeign b
  ) => EnumReadForeign (Sum a b) where
  enumReadForeignImpl f
      = Inl <$> enumReadForeignImpl f
    <|> Inr <$> enumReadForeignImpl f
```

Then we need to match on `Constructor`, but only when its second argument is `NoArguments`, as we want only to work with enum sum types.

```
instance constructorEnumReadForeign ::
  ( IsSymbol name
  ) => EnumReadForeign (Constructor name NoArguments) where
  enumReadForeignImpl f = do
    s <- JSON.readImpl f
    if s == name
       then pure $ Constructor NoArguments
       else throwError <<< pure <<< Foreign.ForeignError $
            "Enum string " <> s <> " did not match expected string " <> name
    where
      name = reflectSymbol (SProxy :: SProxy name)
```

We put a `IsSymbol` constraint on `name` so that can reflect it to a string and check if it is equal to the string that is taken from the foreign value. In the success branch, we construct the `Constructor` value with the `NoArguments` value.

With just this, we can now decode all enum-like sums:

```
readFruit :: String -> Either Foreign.MultipleErrors Fruit
readFruit = JSON.readJSON

main = do
  logShow $ readFruit "\"Abogado\""
  logShow $ readFruit "\"Boat\""
  logShow $ readFruit "\"Candy\""
```

1.6 FAQ

1.6.1 How do I get instances of ReadForeign/WriteForeign for my newtypes?

See the post about PureScript newtype deriving here: https://github.com/paf31/24-days-of-purescript-2016/blob/master/4.markdown

So you can do everything given some definition of a newtype and its instances:

```
-- from test/Quickstart.purs

newtype FancyInt = FancyInt Int

derive newtype instance eqFancyInt :: Eq FancyInt
derive newtype instance showFancyInt :: Show FancyInt
derive newtype instance readForeignFancyInt :: JSON.ReadForeign FancyInt
derive newtype instance writeForeignFancyInt :: JSON.WriteForeign FancyInt
```

1.6.2 Why isn't this library Aeson-compatible?

There are a few factors involved here:

1. I (Justin) don't use Aeson instances.
2. Many Aeson instances revolve around using Sum and Product types (or Haskell Records, which are not structurally similar to PureScript Records).
3. I would rather give you the tools to write your own so that you have instances that match what you are using by having docs/guides like in this page: https://purescript-simple-json.readthedocs.io/en/latest/generics-rep.html
4. There doesn't seem to be anyone else making a general solution library and publishing it.

1.6.3 I just want some random encoding for my Sum types!

If you really are sure you don't want to use the existing instances for Variant (from purescript-variant), you can use the code from here: https://github.com/justinwoo/purescript-simple-json-generics

You might also choose to use this library: https://github.com/justinwoo/purescript-kishimen

1.6.4 How do I handle keys that aren't lower case?

PureScript record labels can be quoted.

```
type MyRecord =
  { "Apple" :: String }

fn :: MyRecord -> String
fn myRecordValue =
  myRecordValue."Apple"
```