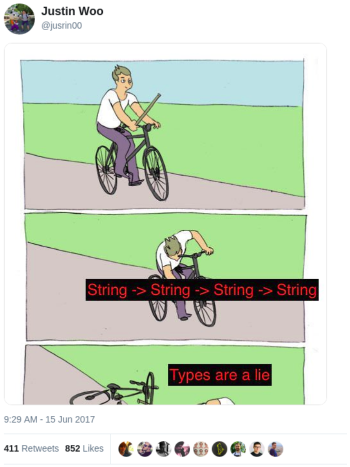
Type-safe routing with PureScript row types

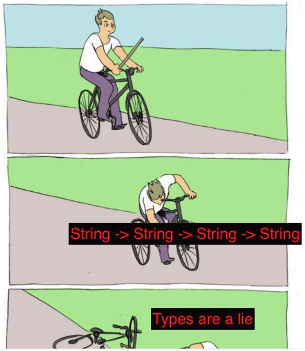
Justin Woo

Monadic Warsaw

2018 Nov 15

@jusrin00 on Twitter


Justin Woo
@jusrin00



9:29 AM - 15 Jun 2017

411 Retweets 852 Likes

後輩ちゃんと文法のコダワリ
※if文・for文 プログラムの基本的な文法
i want to set up my haskell project but there seems to be two options
① cabal new-build
② stack
oh no, we've gotten into religious territory
obviously you use stack
obviously you use cabal
STACK
CABAL
CABALSTACK
絵:結うき。(@yuki_limit) 話:ビタワン(@vitaone_)
3
27
65

Today’s topic:

Type-safe routing with PureScript row types

What is PureScript?

- PureScript is a language like Haskell that can compile to JavaScript
- Some amount of things different as a result of JavaScript being the primary compile target: strict evaluation, naming of instances, etc.
- Most importantly, we have row types which are used for many things, e.g. Records are not product types:

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

type MyRecord =        { a :: Int, b :: String }
type MyRecord = Record ( a :: Int, b :: String )
```

What are “row types”?

- A “row” an unordered collection of fields, consisting of a Symbol label and an associated type.
- While they can be non-Type kinded (e.g. `# CustomKind`), they are generally most useful as `# Type` because PureScript is not polykinded, e.g.:

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

- When used with records, you can define extensible record functions:

```
myFn :: forall r. { name :: String | r } -> String
```

- They also enable interesting libraries like PureScript-Variant, which implements polymorphic variant as a library

```
data Variant :: # Type -> Type
myFn :: forall r. Variant ( name :: String | r ) -> Maybe String
```

What is “Type-safe routing”?

- Vaguely: the ability to define routes of an application that can be accessed with the correct properties
- In my Vidtracker application, this meant being able to define routes that my backend would implement with expected request/responses types, that my front end could also use this information to make valid requests:

```
data Route (method :: RequestMethod) req res (url :: Symbol) = Route

type GetRoute = Route GetRequest Void
type PostRoute = Route PostRequest
```

- Of course, Vidtracker wasn’t always this sophisticated. . .

History of Vidtracker

- Started life as an Elixir app using a untyped Cycle.js frontend
- Ecto didn't (doesn't?) support SQLite, so I used a file with `:erlang.term_to_binary` and `:erlang.binary_to_term`
- Path.join didn't work on Windows correctly
- Std lib documentation was full of "this module is deprecated" messages
- Normal dynamic language pains around being able to change code without things breaking
- Gave up and used PureScript instead

First implementation: record type

- First implementation used this model:

```
type Route =
  { url :: String
  , method :: String
  }
```

- Only works by variable reference
- Request/Response types missing
- Very obvious bugs in the lines of:

```
getResult :: _ (VE (Tuple (Array Path) (Array WatchedData)))
getResult = do
  files <- getJSON "/api/files" -- good
  watched  <- getJSON "/api/files" -- lol wrong route
  pure $ Tuple <$> files <*> watched
```

- Or this:

```
getResult :: _ (VE (Tuple
                    (Array Path) -- good
                    (Array Path) -- lol borked code
                    ))
getResult = do
  files <- getJSON files.url
  watched  <- getJSON watched.url -- got the url right!
  pure $ Tuple <$> files <*> watched
```

Second attempt: simple phantom parameters

- This time, we have the request and response in the parameters:

```
newtype Route request response = Route
  { url :: String
  , method :: String
  }
```

- Pretty much almost there, where then we can write functions like so:

```
postRequest :: forall req res m
  . MonadAff m       -- can lift Aff to m
  => WriteForeign req -- request body can be serialized to JSON
  => ReadForeign res  -- response body can be deserialized from JSON

  => Route req res    -- parameterized Route type
  -> req              -- the request type
  -> m (JSON.E res)   -- Aff with the result of response deserialization
```

- Still, this route model doesn't account for that we have statically known URLs and methods

```
foreign import kind RequestMethod
foreign import data GetRequest :: RequestMethod
foreign import data PostRequest :: RequestMethod

data Route (method :: RequestMethod) req res (url :: Symbol) = Route

type GetRoute = Route GetRequest Void
type PostRoute = Route PostRequest
```

- This still doesn't ensure that all of our routes have actually been implemented, since they stand as individual quantities.

URLs with parameters: see examples of Symbol.Cons in action, e.g. https://github.com/justinwoo/purescript-kushiyaki, https://github.com/justinwoo/purescript-jajanmen

Putting our routes in a record

```
-- (a subset of my routes for slides purposes)
apiRoutes ::
  { files :: GetRoute (Array Path) "/api/files"
  , watched :: GetRoute (Array WatchedData) "/api/watched"
  , update :: PostRoute FileData (Array WatchedData) "/api/update"
  }
apiRoutes = N.reflectRecordProxy
```

- Basically same usage as before in frontend, except that now we reflect the URL
- Bonus: we can now go about checking that we have implemented all of the routes in the backend

```
main = do
  --- ...
      registerRoutes
        apiRoutes
        { files: getFiles config
        , watched: getWatchedData config
        , update: updateWatched config
        }
        app

registerRoutes :: forall routes handlers routesL
   . RowToList routes routesL
  => RoutesHandlers routesL routes handlers
  => Record routes
  -> Record handlers
  -> M.App
  -> Effect Unit
registerRoutes = registerRoutesImpl (RLProxy :: RLProxy routesL)
```

Unordered row types to an iterable type level data type

- Remember that record types are parameterized by `# Type`, which is an unordered collection of fields

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

- What if there were a class to turn this into some iterable structure?

```
class RowToList (row :: # Type) (list :: RowList)
  | row -> list
```

- The functional dependency here says that the row type determines the list

i.e. given a concrete row, you can use this type class to get an instance of RowToList to get out a concrete list to work with

What is `RowList`?

- Built-in type-level data type in the Prim modules of the compiler

```
kind RowList
data Cons :: Symbol -> Type -> RowList -> RowList
data Nil :: RowList

( a :: String, b :: Int )
  becomes
Cons "a" String (Cons "b" Int Nil)
```

- Then we can pattern match on RowList by using type class instances

```
class Keys (xs :: RowList)
instance nilKeys :: Keys Nil
instance consKeys :: Keys (Cons name ty tail)
```

Related: https://speakerdeck.com/justinwoo/type-classes-pattern-matching-for-types

Implementing routes/handlers pairing

```
class RoutesHandlers
  (routesL :: RowList)
  (routes :: # Type)
  (handlers :: # Type)
  | routesL -> routes handlers -- matches by routes rowlist
  where
    registerRoutesImpl :: forall proxy
          . proxy routesL -- any proxy of kind RowList -> Type
      -> Record routes
      -> Record handlers
      -> M.App
      -> Effect Unit
```

Nil case

- The base case, where we have reached the end of the list

```
instance routesHandlersNil :: RoutesHandlers Nil routes handlers where
  registerRoutesImpl _ _ _ _ = pure unit
```

- Nothing else to be done here

Cons case

```
instance routesHandlersCons ::
  ( RoutesHandlers tail routes handlers            -- handle the rest of the list
  , IsSymbol name                                  -- needed for Record.get
  , Row.Cons name handler handlers' handlers       -- get handler from handlers
  , Row.Cons name route routes' routes             -- get route from routes
  , RegisterHandler route handler                  -- type class for registering route
  ) => RoutesHandlers (Cons name route tail) routes handlers where
  registerRoutesImpl _ routes handlers app = do
      registerHandlerImpl route handler app        -- use of RegisterHandler
      registerRoutesImpl tailP routes handlers app -- register the remaining routes
    where
      nameP = SProxy :: SProxy name
      route = Record.get nameP routes
      handler = Record.get nameP handlers
      tailP = RLProxy :: RLProxy tail
```

Only the constraints matter:

```
instance routesHandlersCons ::
  ( RoutesHandlers tail routes handlers      -- handle the rest of the list
  , IsSymbol name                            -- needed for Record.get
  , Row.Cons name handler handlers' handlers -- get handler from handlers
  , Row.Cons name route routes' routes       -- get route from routes
  , RegisterHandler route handler            -- type class for registering route
  ) => RoutesHandlers (Cons name route tail) routes handlers where

-- Prim.Row.Cons works like a box of crayons:
class Cons (label :: Symbol) (a :: Type) (tail :: # Type) (row :: # Type)
  | label a tail -> row
  , label row -> a tail
```

- This is it! (the rest are your typical values that cause bugs)

Conclusion

- Implementing type-level routing with row types is fairly straightforward
- Techniques covered here are not limited to routing, and you might warp this to other purposes
- Main downside is that with this approach we need to define a `Type`-kinded proxy type (is this actually a downside?)
- Each route is individual, so there's no nesting like with some other libraries

Maybe for someone to explore sometime: maybe route information should be just anything that we can use `Row.Cons` to build up?

```
class DeriveHandlers (routes :: YourKind) (handlers :: # Type)
  | routes -> handlers
```

Maybe there's some existing work in https://github.com/owickstrom/purescript-hypertrout to take advantage of?

Thanks!

- Vidtracker codebase:

https://github.com/justinwoo/vidtracker

- Blog post about refining route modeling:

https://github.com/justinwoo/my-blog-posts#opting-in-to-better-types-and-guarantees-in-purescript

- Post about simple phantom types:

https://github.com/justinwoo/my-blog-posts#writing-a-full-stack-app-with-purescript-with-phantom-types

- Post about record of proxies with RowToList

https://github.com/justinwoo/my-blog-posts#record-based-api-route-handler-pairing-with-row-types