t1

August 23, 2021

[1]:
```
-- setup Jupyter notebook
:opt no-lint

-- necessary extensions & imports
{-# LANGUAGE OverloadedLabels #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE DataKinds #-}
-- :set -F -pgmF=record-dot-preprocessor
import Data.Row.WebRecords
import Control.Lens
```

1 Part 1. Declaring types of open records

Record types are parametrized by `Rows`

[2]:
```
:kind Rec
```

```
Rec :: Row * -> *
```

Basic operators for building a row:

[3]:
```
:kind (.==)
:kind (.+)
```

```
(.==) :: forall k. Symbol -> k -> Row k
```

```
(.+) :: forall k. Row k -> Row k -> Row k
```

Example of a row:

[4]:
```
type UserRow = "id" .== Int .+ "name" .== String .+ "friendIDs" .== [Int]
```

`(.+)` is commutative and aassociative: (plus demonstration of constraint chrecking)

[5]:
```
{-# LANGUAGE TypeFamilies #-}
```

```
ok = () :: UserRow ~ ("name" .== String .+ ("friendIDs" .== [Int] .+ "id" .==␣
 ↪Int)) => ()
```

Let's create a User :: Type

[6]:
```
type User = Rec UserRow

:kind! User
```

```
User :: *
= Rec ('R '[ "friendIDs" ':-> [Int], "id" ':-> Int, "name" ':-> [Char]])
```

Internal structure, we should not create such types manually

2 Part 2. Creating and accessing open records

We are using overloaded labels and old operators for creating records. Constraint `Forall l Unconstrained1` can be ignored here

[7]:
```
:t (.+)
:t (.==)
:t (#x .==)
```

```
(.+) :: forall (l :: Row *) (r :: Row *). Forall l Unconstrained1 => Rec l -> Rec r -> Rec (l
```

```
(.==) :: forall (l :: Symbol) a. KnownSymbol l => Label l -> a -> Rec ('R '[ l ':-> a])
```

```
(#x .==) :: forall a. a -> Rec ('R '[ "x" ':-> a])
```

All field labels and types are checked at compile time. Good enough error messages:

Not all fields are initialized

[8]:
```
bob :: User
bob = #id .== 12
```

```
<interactive>:2:7: error:
• Couldn't match type ''[]' with ''[ "id" 'Data.Row.Internal.:-> Int,␣
↪"name" 'Data.Row.Internal.:-> String]'
    Expected type: User
      Actual type: Rec ("friendIDs" .== [Int])
• In the expression: #id .== 12
  In an equation for 'bob': bob = #id .== 12
```

Typo in a field:

[9]:
```
bob :: User
bob = #id .== 12
   .+ #friends .== []
   .+ #name .== "Bob"
```

```
        <interactive>:2:7: error:
        • Couldn't match type ‘"friends"’ with ‘"friendIDs"’
          Expected type: User
            Actual type: Rec ('Data.Row.Internal.R '[ "friends" 'Data.Row.
↪Internal.:-> [Int], "id" 'Data.Row.Internal.:-> Int] .+ 'Data.Row.Internal.R␣
↪'[ "name" 'Data.Row.Internal.:-> String])
        • In the expression: #id .== 12 .+ #friends .== [] .+ #name .== "Bob"
          In an equation for ‘bob’: bob = #id .== 12 .+ #friends .== [] .+ #name␣
↪.== "Bob"
```

Wrong field type:

[10]:
```
bob :: User
bob = #id .== 12
   .+ #name .== "Bob"
   .+ #friendIDs .== Nothing
```

```
        <interactive>:2:7: error:
        • Couldn't match type ‘Maybe a0’ with ‘[Int]’
          Expected type: User
            Actual type: Rec ('Data.Row.Internal.R '[ "id" 'Data.Row.Internal.:
↪-> Int, "name" 'Data.Row.Internal.:-> String] .+ 'Data.Row.Internal.R '[␣
↪"friendIDs" 'Data.Row.Internal.:-> Maybe a0])
        • In the expression: #id .== 12 .+ #name .== "Bob" .+ #friendIDs .==␣
↪Nothing
          In an equation for ‘bob’: bob = #id .== 12 .+ #name .== "Bob" .+␣
↪#friendIDs .== Nothing
```

So let's create a user:

[11]:
```
bob :: User
bob = #name .== "Bob"
   .+ #id .== 12
   .+ #friendIDs .== [13, 14]
```

Autogenerated show and ToJSON/FromJSON instances don't care about order of fields:

[12]:
```
import Data.Aeson
import Data.ByteString.Lazy as LBS
LBS.putStr . encode $ toJSON bob
```

```
bob
```

```
{"friendIDs":[13,14],"name":"Bob","id":12}
```

```
#friendIDs .== [13,14] .+ #id .== 12 .+ #name .== "Bob"
```

Field accessing via Lens:

[13]:
```
bob ^. #id
view #name bob
```

```
12
```

```
"Bob"
```

[14]:
```
f :: User -> (String, Int)
f u = (u ^. #name <> " #" <> show (u ^. #id), u ^. #id)

f bob
```

```
("Bob #12",12)
```

With record dot preprocessor, we can also write

```
f :: User -> (String, Int)
f u = (u.name <> " #" <> show (u.id), u.id)
```

but preprocessor is not stable, space-sensetive, and some advanced updates can be nicely expressed only with lens

3 Part 3. Advanced updating

`Overloaded labels` allows us to use records as lenses for nested, polymorphic and monadic updates:

[71]:
```
{-# LANGUAGE TypeApplications #-}
import Data.Row.Records
z :: User
z = default' @Read (read "")

z
```

```
Prelude.read: no parse
```

4 Part 4. Changing a structure

Let's create a function that adds a field `name` to record:

[29]:
```
giveName s obj = (#name .== s) .+ obj

:t giveName
```

```
giveName :: forall a (r :: Row *). a -> Rec r -> Rec ('R '[ "name" ':-> a] .+ r)
```

`.+` in inferred result type is a type family that can raise a type error if such field already exists:

[30]:
```
thing = #id .== 124
     .+ #struct .== (#aaa .== "aaa" .+ #bbb .== "bbb")
     .+ #name .== "Ken"

thing

:t giveName "The thing" thing
giveName "The thing" thing
```

```
#id .== 124 .+ #name .== "Ken" .+ #struct .== (#aaa .== "aaa" .+ #bbb .== "bbb")

giveName "The thing" thing :: forall a. Num a => Rec ('R (("id" ':-> a) : (TypeError ...)))



        <interactive>:1:1: error:
        • Cannot inject a label into a row type that already has that label
          The label "name" was already assigned the type String and is now␣
↪trying to be assigned the type String.
        • When checking the inferred type
            it :: forall a. Num a => Rec ('Data.Row.Internal.R (("id" 'Data.Row.
↪Internal.:-> a) : (TypeError …)))
```

We can rename a field in struct to fix this

[17]:
```
giveName "The thing" $ rename #name #oldName thing
```

```
#id .== 124 .+ #name .== "The thing" .+ #oldName .== "Ken" .+ #struct .== (#aaa .== "aaa" .+ #
```

Also we can simply drop name field from old structure:

[22]:
```
giveName "The thing" $ thing .- #name
```

```
#id .== 124 .+ #name .== "The thing" .+ #struct .== (#aaa .== "aaa" .+ #bbb .== "bbb")
```

Suppose we have structure like this:

[18]:
```
struct = #user .== bob .+ #thing .==  thing

LBS.putStr . encode $ toJSON struct
```

```
{"thing":{"struct":{"bbb":"bbb","aaa":"aaa"},"name":"Ken","id":124},"user":{"friendIDs":[13,14
```

We can split a record to two parts, or restrict to subset. Via lens we can focus on subrecords to operate on them

[59]:
```
import Data.Row.Records (split)

type UserInfo = Rec ("name" .== String .+ "id" .== Int)
makeUser :: UserInfo -> [Int] -> User
makeUser u friends = u .+ #friendIDs .== friends

userInfo :: User -> UserInfo
userInfo = restrict

splitUser :: User -> (UserInfo, [Int])
splitUser u = let (a, b) = Data.Row.Records.split u in (a, b ^. #friendIDs)

splitUser bob
```

```
(#id .== 12 .+ #name .== "Bob",[13,14])
```

[36]:
```
import Data.Row.Records (restrict)
struct2 :: Rec ("thing" .== Rec ("name" .== String) .+ "user" .== Rec ("name" .
 ↪== String))
struct2 = struct & #thing %~ restrict
                 & #user %~ restrict

struct2
```

```
#thing .== (#name .== "Ken") .+ #user .== (#name .== "Bob")
```

5 Part 5. Functions to and from record structures

`r .! "a"` is either value of #a in r or TypeError

[2]:
```
f x = x ^. #aaa == 1

f (#aa .== 12)
```

```
       <interactive>:1:1: error:
       • Non type-variable argument in the constraint: Data.Generics.Product.
↪Fields.HasField' "aaa" s a
         (Use FlexibleContexts to permit this)
       • When checking the inferred type
           f :: forall a s. (Eq a, Data.Generics.Product.Fields.HasField' "aaa"␣
↪s a, Num a) => s -> Bool
```