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:

[11]:
```
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:

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

Autogenerated show and ToJSON/FromJSON instances:

[ ]:
```
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:

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

```
12
```

```
"Bob"
```

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

f bob
```

```
    <interactive>:1:3: error: Variable not in scope: bob :: Rec ('Data.Row.
↪Internal.R '[ "friendIDs" 'Data.Row.Internal.:-> [Int], "id" 'Data.Row.
↪Internal.:-> Int, "name" 'Data.Row.Internal.:-> String])
```

3 Part 3. Advanced updating

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

[ ]:
```
a = #id .== 124 .+ #name .== "Ken" .+ #struct .== (#aaa .== "aaa" .+ #bbb .==␣
↪"bbb")
a
```

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

[ ]:
```

```