Typeclasses

UI Architecture

Feb 2021

Takashi Idobe

MVC

Model

```
class CreateCounter < ActiveRecord::Migration[6.0]
  def change
    create_table :counter do |t|
      t.integer :count
    end
  end
end

class Counter < ApplicationRecord
end
```

Controller

```
Counter::Application.routes.draw do
  match '/incr' => 'counter#incr', via: :post
  match '/decr' => 'counter#decr', via: :post
  match '/count' => 'counter#count', via :get
end

class CounterController < ApplicationController
  def new
    @Counter = Counter.new
  end
  def get
```

```
    { count: @Counter.count }.to_json
  end
  def incr
    @Counter.count += 1
    { count: @Counter.count }.to_json
  end
  def decr
    @Counter.count -= 1
    { count: @Counter.count }.to_json
  end
end
```

View

```
# POST /incr
{count: 1}
# POST /decr
{count: 0}
# GET /count
{count: 0}
```

React

```
import React, { useState } from "react";

function App() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <button onClick={() => setCount(count - 1)}>-</button>
      {count}
      <button onClick={() => setCount(count + 1)}>+</button>
    </div>
  );
}
```

- What happens if we want to update this from a child component?
- What happens if we don't want to prop drill up and down?

```
function App() {
  return (
    <div>
```

```
      <Counter />
      <CounterCopy count={count}>
    </div>
  )
}
```

```
const initialState = { count: 0 };
function reducer(state = initialState, action) {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + 1 };
    case "DECREMENT":
      return { count: state.count - 1 };
    default:
      return state;
  }
}
```

```
function Counter({count, dispatch}) {
  increment = () => dispatch({ type: "INCREMENT" });
  decrement = () => dispatch({ type: "DECREMENT" });

  return (
    <div>
      <button onClick={decrement}>-</button>
      {count}
      <button onClick={increment}>+</button>
    </div>
  );
}

const mapStateToProps = (state) => ({
  count: state.count;
});

export default connect(mapStateToProps)(Counter);
```

Counter Copy

```
function CounterCopy({ count }) {
  return <div>Copy: {count}</div>;
}
```

The Elm Architecture

- **Model** the state of your application
- **View** The HTML on your page
- **Update** a way to update your state with messages

Let's build a counter in elm.

Main

```
import Browser
import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)

-- MAIN

main : Program Value Model Msg
main =
    Browser.application
        {
          init = init
        , view = view
        , update = update
        }
```

Model

```
-- MODEL

type alias Model = Int

init : Model
init = 0
```

Update

```
-- UPDATE

type Msg = Increment | Decrement

update : Msg -> Model -> Model
update msg model =
    case msg of
        Increment ->
            model + 1
        Decrement ->
            model - 1
```

View

```
-- VIEW

view : Model -> Html Msg
view model =
    div []
        [ button [ onClick Decrement ] [ text "-" ]
        , div [] [ text (String.fromInt model) ]
        , button [ onClick Increment ] [ text "+" ]
        ]
```

Example

Nice features of Elm

- Type Aliases
- Pattern Matching
- Maybe
- Result
- Error Handling

Structural Typing

```
type alias User =
    { name : String
    , age : Int
    }

isOldEnoughToVote : User -> Bool
isOldEnoughToVote user =
  user.age >= 18
```

```
isOldEnoughToVote : { name : String, age : Int } -> Bool
isOldEnoughToVote user =
  user.age >= 18
```

Enums

```
type UserStatus
  = Regular
  | Visitor

type alias User =
  { status : UserStatus
  , name : String
  }

thomas = { status = Regular, name = "Thomas" }
kate95 = { status = Visitor, name = "kate95" }
```

Pattern Matching

```
greetUser : User -> String
greetUser user =
    case user.status of
        Regular ->
            "Hi Regular " ++ user.name
        Visitor ->
            "Hi Visitor " ++ user.name
```

Enums

```
type User
  = Regular String String
  | Visitor String

thomas = Regular "Thomas" "Kahlua"
kate95 = Visitor "kate95"

greetUser : User -> String
greetUser user =
    case user of
        Regular name drink ->
            "Hi Regular " ++ name ++ "Who's favorite drink is: " ++ drink
        Visitor name ->
            "Hi Visitor " ++ name ++ "What would you like to order?"
```

Maybe

- When you're not sure an operation will succeed.

```
type Maybe a
  = Just a
  | Nothing
```

In use

```
> String.toInt "3"
Just 3
> String.toInt "3.14"
Nothing
> String.toInt "abc"
Nothing
> String.toInt Nothing
-- Compiler Error
```

In ruby?

```
> "3".to_i
=> 3
> "3.14".to_i
=> 3
> "abc".to_i
=> 0
> nil.to_i
=> 0
```

Results

```
isReasonableAge : String -> Result String Int
isReasonableAge input =
  case String.toInt input of
    Nothing ->
      Err "That is not a number!"

    Just age ->
```

```
      if age < 0 then
        Err "Please try again after you are born."

      else if age > 135 then
        Err "Are you some kind of turtle?"

      else
        Ok age
```

HTTP Requests

JSLand

```
fetch(url).then(res => res.json()).then(res => /* do something */)
// I forgot to add a .catch(), so this program could error out at any time.
```

---

in Elm

```
-- Main
main =
  Browser.element
    { init = init
    , update = update
    , view = view
    }
```

```
-- Model
type Model
  = Failure
  | Loading
  | Success String


init : () -> (Model, Cmd Msg)
init _ =
  ( Loading
  , Http.get
      { url = "https://elm-lang.org/assets/public-opinion.txt"
      , expect = Http.expectString GotText
      }
  )
```

```
-- UPDATE
type Msg
  = GotText (Result Http.Error String)


update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    GotText result ->
      case result of
        Ok fullText ->
          (Success fullText, Cmd.none)

        Err _ ->
          (Failure, Cmd.none)
```

```
-- VIEW
view : Model -> Html Msg
view model =
  case model of
    Failure ->
      text "I was unable to load your book."

    Loading ->
      text "Loading..."

    Success fullText ->
      pre [] [ text fullText ]
```

A breath of fresh air. All states are taken care of by the compiler.

The Most Famous Ruby Programmer

Why the Lucky Stiff on NULL

(To me, fighting NULL is the epitome of why I struggled as a programmer. I am not a natural at it, but I wanted very much to be–and I found no use for NULL. I never needed it, but it was always there. I kept pushing it down, painting over it, shutting it up, constantly checking for it–

“Are you NULL?

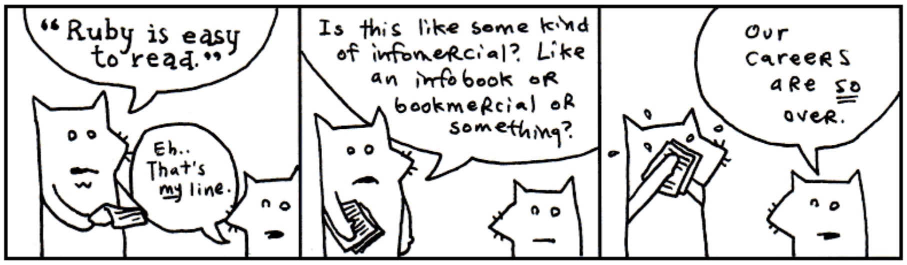
"Ruby is easy to read."
Eh.. That's my line.
Is this like some kind of infomercial? Like an infobook or bookmercial or something?
Our careers are so over.

Figure 1: Foxes

Are you NULL?

What about you?"

Cont.

–and sometimes I would deceive myself, that my problems were other things, but then NULL would pop up, I would find that it was the cause–however, NULL is never really the cause. It is someone you always run into in bad situations, someone you never want to see. NULL penetrates all the layers to find you, and can only say, helplessly, "Looks like you're having a problem." Endemic to the problem, not the problem, complicit, and might be the problem.)

Takeaways

- More compile time errors is good.
- Exceptions and nils and runtime errors are hard to track down.
- Functional Programming is intuitive.