Please

Ask questions through the app

Rate Session

Thank you!

@GOTOber

gotober.com

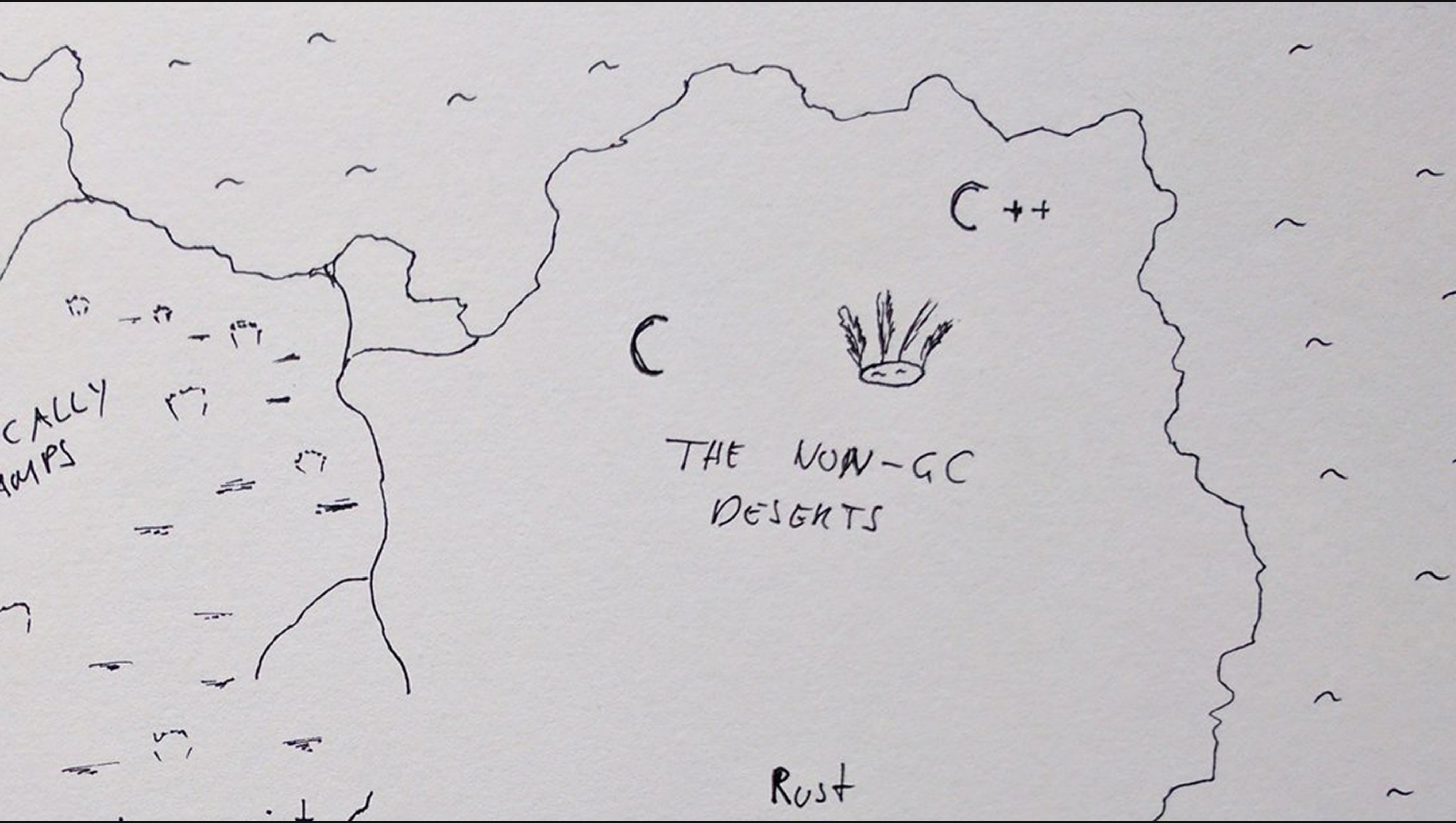
Make Web Apps Fun to Build & Easy to Refactor with Elm

danielbachler.de

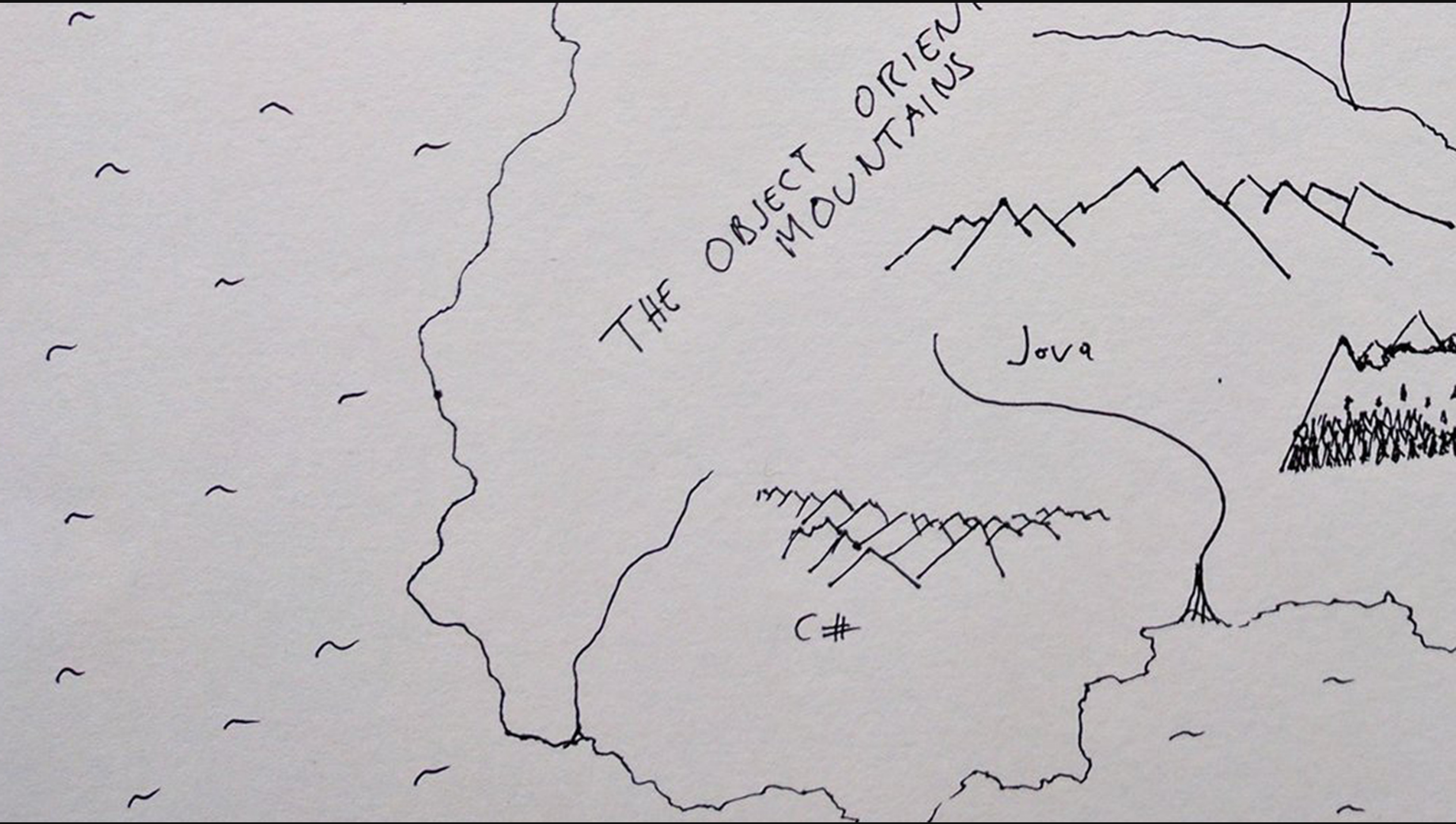
@danyx23 on Twitter

Work at Douglas Connect

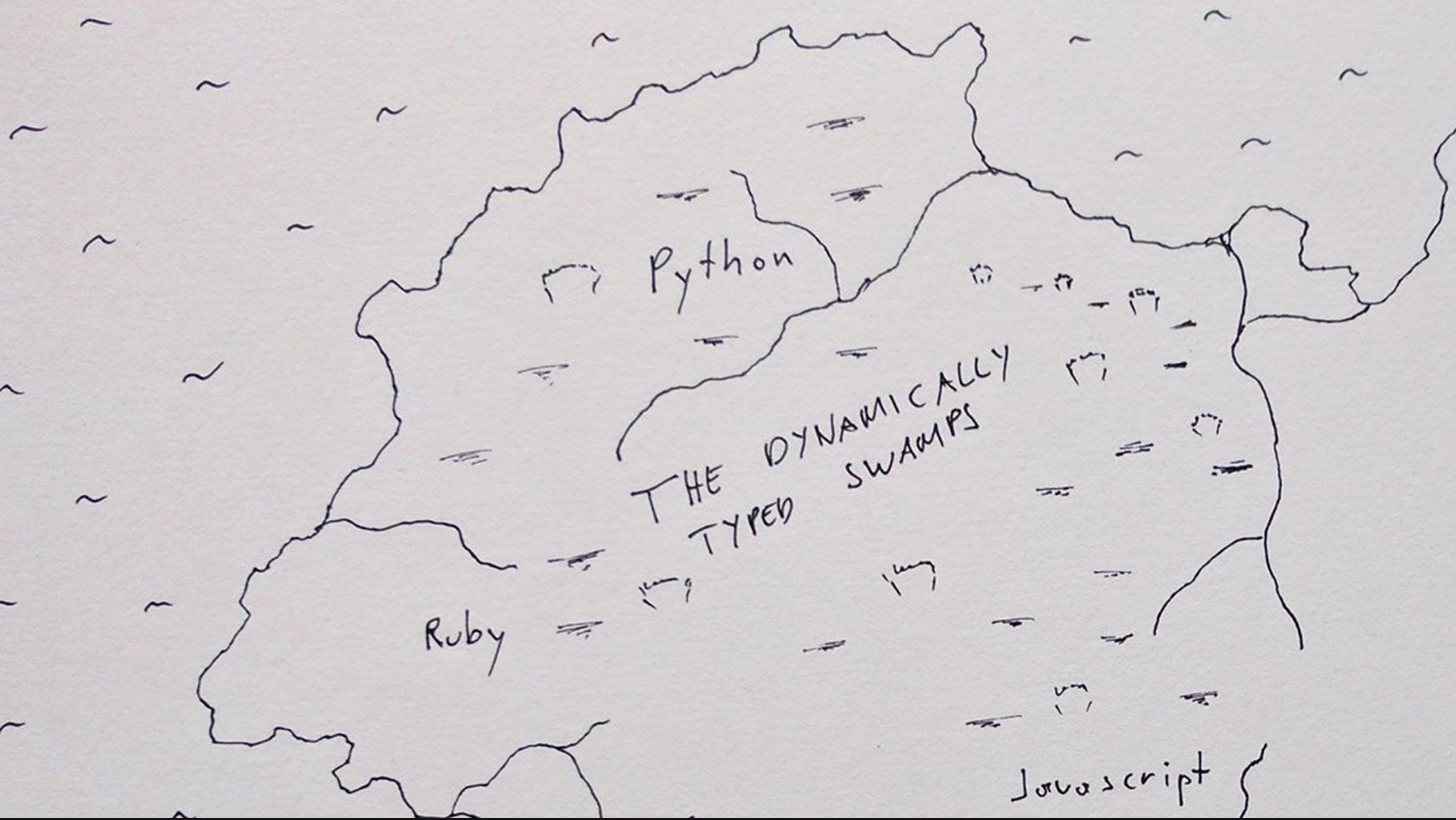
Python
THE DYNAMICALLY TYPED SWAMPS
Ruby
Javascript
Go
C++
C
THE NON-GC DESERTS
Rust
THE FUNTIONAL FORESTS
Elm
Purescript
Sea of Immutability
Haskell
THE OBJECT ORIENTED MOUNTAINS
Java
Scala
C#
Ocaml
F#
F*
Z3
COQ
AGDA

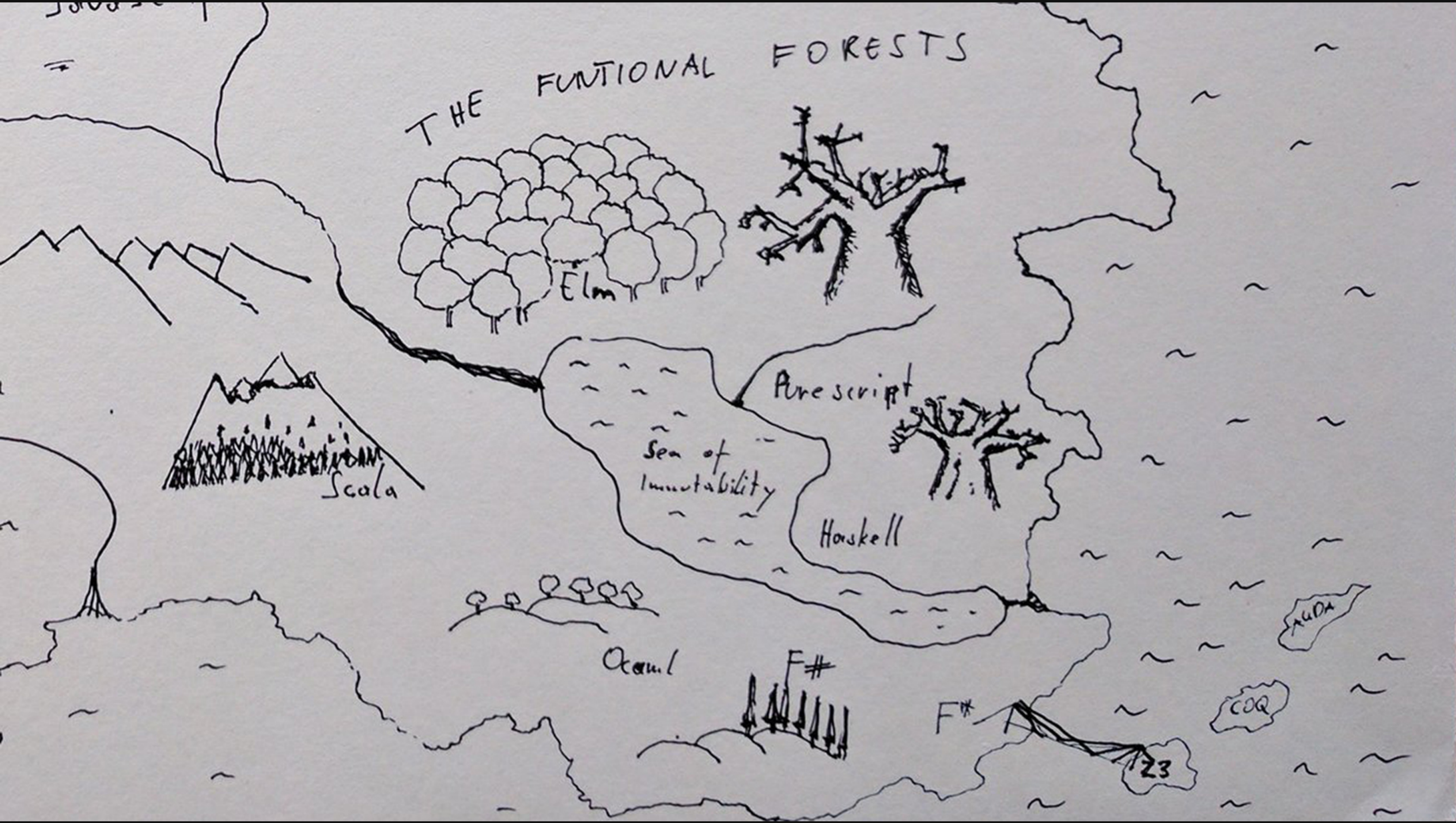


CALLY
AMPS
C++
C
THE NON-GC
DESERTS
Rust



THE OBJECT ORIEN
MOUNTAINS
Java
C#


Python
THE DYNAMICALLY
TYPED SWAMPS
Ruby
Javascript


THE FUNTIONAL FORESTS
Elm
Purescript
Scala
Sea of Immutability
Haskell
Ocaml
F#
F*
Z3
COQ
AGDA

Elm Elevator pitch

- Statically typed, purely functional programming language
- Compiles to Javascript
- **No runtime errors**
- Easy to learn, nice to use

Javascript syntax

```
function multiplyNumbers(a, b) {
    return a * b;
}

// Weird type coercion
var result = multiplyNumbers(4, "three");

```

Elm syntax

```
        multiplyNumbers a  b =
          a * b


-- Compile error!
    result = multiplyNumbers 4  "three"
```

Elm syntax

```

multiplyNumbers a b =
    a * b


result = multiplyNumbers 4 3
```

Type annotations

```
multiplyNumbers : Int -> Int -> Int
multiplyNumbers a b =
    a * b

result : Int
result = multiplyNumbers 4 3
```

Type annotations

```
type alias Person =
    { name : String
    , yearBorn : Int
    }

calculateAge : Int -> Person -> Int
calculateAge currentYear person =
    currentYear - person.yearBorn

```

Pain points Elm adresses

Code in dynamic languages is hard to refactor correctly

- So we do it less => lower code quality
- Often introduce bugs/crashes

In Elm, everything is fully typed

- Even when no type annotations are used, ever
- The compiler checks that all types match
- No "any" type

Records

(Product types)

```
type alias Programmer =
    { name : String
    , favouriteLanguage : String
    }

daniel : Programmer
daniel =
    { name = "Daniel"
    , favouriteLanguage = "Elm"
    }
```

Union types

(aka Sum types)

```
type Status
    = Pending
    | Completed

val1 = Pending

type alias Task =
    { name : String
    , status : Status
    }
```

Pattern matching

```
getUIString : Status -> String
getUIString status =
    case status of
        Pending -> "Not yet started"
        -- Compile error! Missing case!

```

Pattern matching

```
getUIString : Status -> String
getUIString status =
    case status of
        Pending -> "Not yet started"
        Completed -> "Completed"
```

What if only some states have data attached?

- Progress report when running
- How would you model this in another language?

```
type Status
    = Pending
    | Completed
    | Failed

type alias Task =
    { name : String
    , status : Status
    , currentItem : Int
    , numItems : Int
    , errors : List String
    }
```

Making invalid states unrepresentable

The real power of union types

```
type Status
    = Pending
    | Running Int Int -- Two ints as "payload" data
    | Completed
    | Failed (List String) -- a list of strings as "payload" data

val1 : Status
val1 = Running 0 10

type alias Task =
    { name : String
    , status : Status
    }
```

Pattern matching

```
getUIString : Status -> String
getUIString status =
    case status of
        Pending ->
            "Not yet started"
        Running current total ->
            (toString (current + 1)) ++ " of " ++ (toString total)
        Completed ->
            "Completed"
        Failed errors ->
            "Failed! Message : " ++ (String.join ", " errors)
```

Pattern matching

- Pattern matching is the only way to get payload "out" of a union type

Polymorphic types

(aka Generics)

```
type BinaryTree elementType
    = Leaf elementType
    | Node (BinaryTree elementType) (BinaryTree elementType)

leafOnly : BinaryTree Int
leafOnly =
    Leaf 23

smallTree : BinaryTree Int
smallTree =
    Node (Leaf 17) leafOnly
```

Undefined is not a function / NullReferenceException

- Elm does not have null/undefined
- This kills a whole family of bugs

How can it represent missing values?

Dealing with optional values

```
type Maybe a
    = Nothing
    | Just a

val1 : Maybe Int
val1 = Nothing

val2 : Maybe Int
val2 = Just 23
```

What if we need error information?

```
type Result err success
    = Ok success
    | Err err

val1 : Result String Int
val1 = Err "This is an error message"

val2 : Result String Int
val2 = Ok 23
```

All values are immutable

```
x = 1

x = 2 -- compile error

x = x + 1 -- compile error

y = x + 1 -- Ok
```

All (nested) fields are immutable

```
type alias Programmer =
    { name : String
    , favouriteLanguage : String
    }

programmerA : Programmer
programmerA =
    { name = "Daniel"
    , favouriteLanguage = "Elm"
    }

programmerA.name = "Eve"
-- Compile error!
```

Creating new record values based on old ones

```
type alias Programmer =
    { name : String
    , favouriteLanguage : String
    }

programmerA : Programmer
programmerA =
    { name = "Daniel"
    , favouriteLanguage = "Elm"
    }

programmerB : Programmer
programmerB =
    { programmerA
    | name = "Eve"
    }
```

This means we can't do loops in elm!

- Use map, fold (aka reduce), or recursion instead

Elm is entirely pure!

- No side effects possible in the language
- (Except Debug.log and Debug.crash)

Getting work done with Elm

- Elm comes with a small runtime
- No direkt Javascript FFI

```elm
-- Elm
addNumbers : Int -> Int -> Int
addNumbers a b =
    a + b

result1 = addNumbers 1 2
result2 = addNumbers 1 2
result 1 == result 2 -- True
```

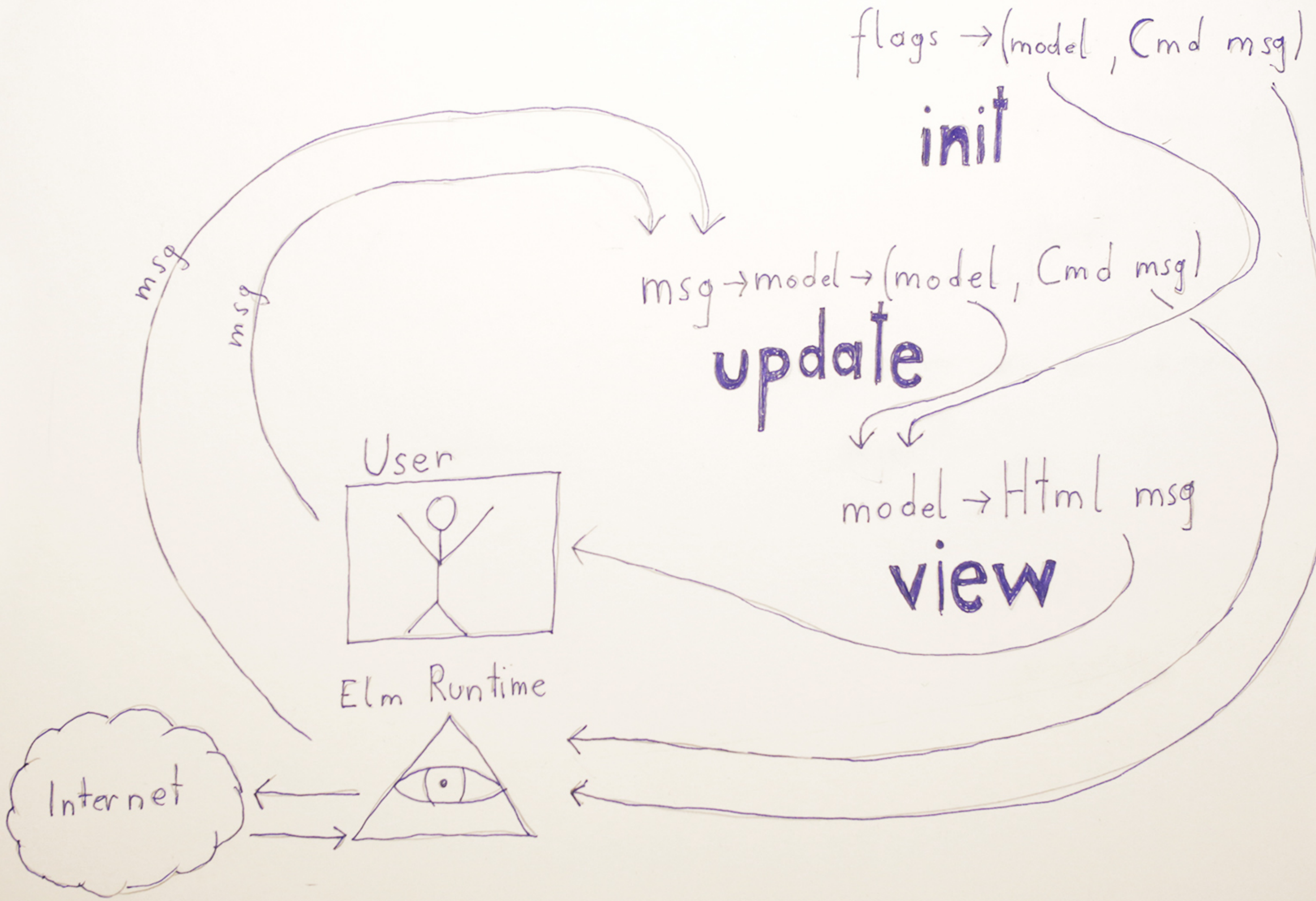
```javascript
/// Javascript
function addNumbersWeird(a, b) {
    window.myGlobalState = window.myGlobalState || 0;
    return a + b + (window.myGlobalState++);
}

var result1 = addNumbersWeird(1, 2);
var result2 = addNumbersWeird(1, 2);
result1 == result2; // False
```

This makes testing super nice

- Calling the same function again with the same arguments will always lead to the same result
- Thanks to static types, Unit testing can focus on actual logic
- Mocking is usually not necessary with pure functions

And refactorings are safe and fun!

The elm architecture


flags → (model , Cmd msg)
init
msg → model → (model , Cmd msg)
update
model → Html msg
view
msg
msg
User
Elm Runtime
Internet

Benefits

- Model is a single source of truth
- Visual elements are created from the current model
- Apps are well structured
- update function is the only place where your state is modified
- Possible to replay UI sessions easily, implement Undo/Redo, ...

How view functions work

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

```

```
<div class="counter">
    <button onClick="dispatch(Decrement)">-</button>
    <div>{model.counter}</div>
    <button onClick="dispatch(Increment)">+</button>
</div>

```

Demo time

Command values tell the Elm runtime to perform side effects

- Like HTTP requests
- Random number generation
- ...

Commands in action

```
import Http

type Msg
  = LoadClicked
  | Loaded (Result Http.Error String)

sendCommand : Cmd Msg
sendCommand =
  Http.send Loaded (Http.getString "https://example.com/books/war-and-pe

update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
    case msg of
        LoadClicked ->
            (model, sendCommand)
        Loaded (Ok text) ->
            ...
        Loaded (Err httpErr) ->
            ...
```

Ports are used to send messages to/from Javascript

- This lets you use any Javascript library / Browser API in native JS
- Send messages back and forth between Elm (Business Logic, Rendering) and your native JS code

Building production apps with Elm

- Overall: very nice experience
- No runtime exceptions, evar!
- Compiler helps you, especially when refactoring
- Wonderful confidence in our code

Obstacles with Elm

- Sometimes you need to use ports for trivial things (e.g. focus an input element)
- Can't publish modules with "native" Javascript as official elm package (e.g. library to use Web Audio API)
- Writing Json Decoders is a bit tedious

Elm is ready to be used in production

- Drastically reduced bug count
- Development speed does not slow down as project gets more complex
- Some JavaScript interop via ports probably necessary, but still much better than all JS!

Where to go to learn more?

- try.elm-lang.org
- http://elm-lang.org
- Try it for a side project or internal tool
- Go on the Elm slack and ask questions!

Thank you!

danielbachler.de

@danyx23 on Twitter

Please

Remember to rate this session

Thank you!

@GOTOber

gotober.com