Strongly-typed, Functional programming for the Frontend

You may not know it, but you may be suffering from JavaScript Fatigue

Frontend Málaga Meetup

Málaga, December 20th 2016

Carlos D.

What this talk is about

- The whats and the whys
- Practical usage

What this talk not is about

- *Functors, Applicatives, Semigroups, Monoids, Monads...*

Though we can talk about this briefly, of course!

Agenda

- What is Functional Programming? Why should we care?
- What is a strongly-typed language? Again, why should we care?
- Practical PureScript
  - Intro
  - Your basic toolset
  - Intro to the language
  - Looking at the produced output
  - Interop-*ing*
  - Demo of a working application

Code samples

```
$ git clone https://github.com/charlydagos/purescript-malaga-demo.git
```

My purpose

To get *at least some of you* interested in developing apps with PureScript.

Introduction

About me

I'm Carlos.

I've been paid to write Java, Scala and PHP.

I've been kindly asked to write Ruby, Python, Clojure, Objective-C, others...

I currently write Haskell.

Functional Programming

What is it?

It's a programming paradigm where we treat functions as *first class citizens*[1].

It means that...

- We can "talk" about a function without any special notation.
- We can "store" functions in variables.
- Functions can take other functions as arguments (*higher order functions*).

Is JavaScript a *functional programming language*?

[1] Evaluation order and avoidance of state and mutability is also part of the paradigm.

Functional Programming

Why is it important?

First and foremost: It's as important as learning any other paradigm.

Most importantly

It allows for types of abstractions that may otherwise not be found in languages that don't support functional programming.

```
var scream = function(name) {
    return name.toUpperCase() + "!!!";
}

var screamedNames = ["Alice", "Bob"].map(scream) // What is `scream`?
                                                 // What is `map`?

console.log(screamedNames) // What is the output?
```

What is the *imperative* (i.e. non-functional) code that would produce the same output?

Functional Programming

Why is it important?

Abstraction, Abstraction, Abstraction

- What makes functions so different from other *citizens*?
- What can we do with functions when they're first-class citizens?

Why is this important?

- What is our role as engineers and/or developers?
- How can we achieve this?
- What *mechanisms* do we have?

How does functional programming *and* strong typing help us achieve this?

Strongly-typed languages

What is it?

```
var scream = function(name) {
    return name.toUpperCase() + "!!!"
}

var screamNames = function(names) {
    return names.map(scream)
}

var names = ["Alice", "Bob"]

console.log(screamNames(names))
```

- Where are the bugs?
- How can we spot them?
- How many of them are there?
- How could we guarantee that our code is *safe*?

Strongly-typed languages

What is it?

Compare to

```
scream :: String -> String
scream name = name ++ "!!!"

screamNames :: [String] -> [String]
screamNames names = map scream names

names :: [String]
names = ["Alice", "Bob"]

main = log (show $ screamNames names)
```

- What are the differences?
- Where are the bugs now?
- What *really* changed?
- Where's the benefit?

Strongly-typed languages

Why is it important?

It's important to learn the difference between

- Languages
- Compilers
- Interpreters

A strongly-typed *language* allows a *compiler* implementation to make use of all the information supplied by the language in order to perform a *static analysis* of our program.

Some compilers may or may not include a *runtime*.

Strongly-typed languages

Why is it important?

What happens if we now say

```
scream :: String -> String
scream name = name <> "!!!"

screamNames :: [String] -> String
screamNames names = map scream names

names :: [String]
names = ["Alice", "Bob"]

main = log (show $ screamNames names)
```

- See the difference? (I made this bug as I wrote this presentation)
- What's the effect? (We'll see an example later!)

Practical PureScript

Intro

What is PureScript?

From http://www.purescript.org

*PureScript is a small **strongly typed** programming language that **compiles** to JavaScript.*

Looking at this sentence we can tell what we're getting.

Practical PureScript

Intro

Let's revisit our previous code

```
module Main where

import Prelude
import Control.Monad.Eff.Console (log)

scream :: String -> String
scream name = name <> "!!!"

screamNames :: Array String -> Array String
screamNames names = map scream names

names :: Array String
names = ["Alice", "Bob"]

main = log (show $ screamNames names)
```

Same output we were looking for before, but this time we can *make claims about our program*. Namely, that it *compiles*.

Practical PureScript

Intro

Let's install it!

- Mac

```
brew install purescript
```

- Unix-like systems

```
npm install -g purescript
```

- Windows

I recommend using a package manager like https://chocolatey.org/

And then

```
choco install purescript
```

Practical PureScript

Intro

We have now two tools

- `psc`: Our compiler
- `psci`: Our interactive Read-Evaluate-Print-Loop (REPL)

This is great! But not enough. We'll see more soon!

Where can we write PureScript?

I use nvim, another option is Atom editor with the plugins recommended here

https://github.com/purescript/purescript/wiki/Editor-and-tool-support

Practical PureScript

A basic toolset

Pulp! It's our most basic build tool.

```
$ mkdir example-app          # Makes a dir to work on
$ cd example-app             # Switch to that dir
$ npm init .                 # Starts a new project (creates package.json)
$ npm install --save pulp    # Install pulp as a dependency
$ pulp init                  # Creates a project skeleton
$ pulp psci                  # Launches us into a console
```

This gets us ready to start writing :)

Personal recommendation

`npm install -g pscid`

Run `pscid` in a shell next to your code to get an immediate update on errors and warnings.

Practical PureScript

Intro to the language

Basic syntax

The type system should tell us *everything* we need to know about a function, and for this we use the `::` symbol, which can be translated as `is of type`

```
f :: Int
f = 2
```

Modules

All the code in PureScript is separated into `modules`, in other words:

```
module Main where

f :: MyType
f = ...

g :: SomeType -> AnotherType
g = ...
```

Practical PureScript

Intro to the language

The type system

Strongly typed. What would be *weakly* typed?

Functions

```
-- What is f?
f :: Int -> Int -> Int
f x y = x + y

-- What is g?
g :: Int
g = 2

-- What is h?
h :: (Int -> Int) -> [Int] -> Int
h f ints = sum (map f ints)
```

Practical PureScript

Intro to the language

Data types

```
-- Type
data Hand = Rock | Paper | Scissors -- Values

-- Type
data Maybe a = Nothing | Just a     -- Values

-- Type Alias
type NameRecord t = { name :: String
                    , lastName :: String
                    | t
                    }

fullName :: forall t. NameRecord t -> String
fullName record = record.name <> " " <> record.lastName

fullName' :: forall t. { name :: String, lastName :: String | t } -> String
fullName' record = record.name <> " " <> record.lastName
```

Types vs *Values*? What's the relationship?

Practical PureScript

Intro to the language

Declarative programming and pattern-matching

```
data List a = End | Cons a (List a)

myList :: List Int
myList = Cons 1 (Cons 2 (Cons 3 End))

emptyList :: List Int
emptyList = End

find :: Int -> List Int -> Maybe Int
find x (Cons y rest) = if x == y
                       then Just x
                       else find x rest
find _ End = Nothing

main = do
    log (head myList)    -- Prints "Just 1"
    log (head emptyList) -- Prints "Nothing"

-- ^ Can that be abstracted?
```

Practical PureScript

Intro to the language

Effects

So far we've seen functions that don't do "anything"[2]

What about functions that do:

- Console IO
- Throw Exceptions
- Perform DOM manipulation
- Perform XMLHttpRequest / AJAX calls
- Interact with a websocket
- Read/Write to/from some storage

We use the `Eff` *monad*[3].

[2] Used very, very loosely
[3] We won't get into this today

Practical PureScript

Intro to the language

Effects

What makes these functions so different?

```
module Main where

import Control.Monad.Eff         (Eff)
import Control.Monad.Eff.Console (CONSOLE, log)

trivialEff :: forall e. Eff e String
trivialEff = pure "World"

main :: Eff (console :: CONSOLE) Unit
main = do
  string <- trivialEff
  log "Hey what's up..."
  log $ "..." <> string <> "!"
```

What is the type of `main`?

Practical PureScript

Produced output

PureScript's author's intention is to make the ouput *easy to read and to debug*.

Sidenote: CoffeeScript, anyone?

- Basic functions.
- Higher-order function.
- Simple data types.
- `Maybe a` example.

Practical PureScript

Interop

- Why would we need such a thing?

We don't want to rewrite the wheel!

- Foreign Function Interface (FFI)

It's as simple as placing a file of the same name next to the PureScript file with a `.js` extension, and using `exports` accordingly. Let's see an example.

Practical PureScript

A working app

- Pux
- Webpack
- Bower

Further Reading

On PureScript

- Learn PureScript: http://www.purescript.org/learn/
- PureScript by Example: https://leanpub.com/purescript

On Haskell

- Learn You a Haskell For Great Good: http://learnyouahaskell.com/
- Haskell Book: http://haskellbook.com/
- What I wish I knew when learning Haskell: http://dev.stephendiehl.com/hask/

On Category Theory

- Category Theory by Steve Awodey
- Category Theory for Developers: https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/
- Category Theory for Developers (Videos): https://www.youtube.com/playlist?list=PLbgaMIhjbmEnaH_LTkxLI7FMa2HsnawM_