The Erlang Ecosystem for Functional Programmers (and everyone else, too)

Robert Ellen

2023/05/16

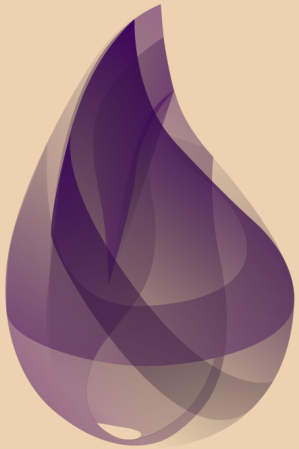
Summary

* it's been ten years since an Erlang talk at BFPG
* this is another whirlwind tour
* what makes the Erlang ecosystem awesome
* some things to help everyone write better Erlang / Elixir

What is the Erlang Ecosystem?

A group of languages, libraries, frameworks, and applications that are implemented on top of the Erlang virtual machine, the BEAM.

Languages include:



elixir

...plus dozens more

What is the Erlang Ecosystem?

Libraries and frameworks include:

OTP

What is the Erlang Ecosystem?

Built around a shared value in:

* massive concurrency
* fault-tolerance
* simplicity
* acknowledging the errors will occur so lets deal with them
* but at the same time, "Let it crash"

Brief history of Erlang

covered in my 2013 talk, but tonight...

Brief history of Erlang

Brief history of Erlang

* developed in the mid 1980s at Ericsson
* to run on next generation telephone switches
  - concurrent
  - fault-tolerant
  - distributed
  - soft real-time
* solved web-scale in the '80s



Brief history of Erlang

* fell out of favour at Ericsson in the late '90s
* open-sourced in 1998
* reports of market penetration of $> 50\%$ in mobile telephony switches
* reports of 1200k LOC and nine nines of uptime on the AXD301 switch

A taste of Erlang

```
-module(process_example).
-export([start/0, ping/2, pong/0]).

start() ->
    Pid = spawn(fun() -> pong() end),
    spawn(process_example, ping, [3, Pid]).

ping(0, Pid) ->
    Pid ! finished,
    io:fwrite("Ping finished~n");

ping(N, Pid) ->
    Pid ! {ping, self()},
    receive
        pong -> io:fwrite("Ping received pong~n")
    end,
    ping(N - 1, Pid).

pong() ->
    receive
        finished -> io:fwrite("Pong finished~n");
        {ping, Ping_PID} ->
            Ping_PID ! pong,
            io:fwrite("Pong received ping~n"),
            pong()
    end.
```

A sip of Elixir

```elixir
defmodule ProcessExample do
  def start() do
    pid = spawn(&pong/0)
    spawn(fn -> ping(3, pid) end)
  end

  def ping(n, pid) when n > 0 do
    send(pid, {:ping, self()})

    receive do
      :pong -> IO.puts("Ping received pong")
    end

    ping(n - 1, pid)
  end

  def ping(0, pid) do
    send(pid, :finished)
    IO.puts("Ping finished")
  end
```

A sip of Elixir

```
  def pong() do
    receive do
      :finished ->
        IO.puts("Pong finished")

      {:ping, pid} ->
        send(pid, :pong)
        IO.puts("Pong received ping")
        pong()
    end
  end
end
```

Crash

```
def ping(0, _pid) do
  apply(:foo, :bar, [])
end

Shell PID is: #PID<0.212.0>
...
[error] Process #PID<0.314.0> raised an exception
** (UndefinedFunctionError) function :foo.bar/0 is undefined
(module :foo is not available)
    :foo.bar()
```

Crash with a linked process

```elixir
def start() do
  IO.inspect(self(), label: "Shell PID is")
  pid = spawn(&pong/0)
  spawn_link(fn -> ping(3, pid) end)
end
```

```
Shell PID is: #PID<0.275.0>
...
** (EXIT from #PID<0.275.0>) shell process exited with reason:
an exception was raised:
    ** (UndefinedFunctionError) function :foo.bar/0 is undefined
    (module :foo is not available)
        :foo.bar()
```

Crash with traped exits

```elixir
def start() do
   IO.inspect(self(), label: "Shell PID is")
   Process.flag(:trap_exit, true)
   pid = spawn(&pong/0)
   spawn_link(fn -> ping(3, pid) end)

   receive do
     msg ->
       IO.inspect(msg, label: "Got message")
   end
 end
```

```
Shell PID is: #PID<0.282.0>
...
15:34:57.211 [error] Process #PID<0.313.0> raised an exception
** (UndefinedFunctionError) function :foo.bar/0 is undefined
(module :foo is not available)
    :foo.bar()
Got message: {:EXIT, #PID<0.313.0>, {:undef, [{:foo, :bar, [], []}]}
{:EXIT, #PID<0.313.0>, {:undef, [{:foo, :bar, [], []}]}}`
```

The BEAM

* Erlang's virtual machine / runtime system
* lightweight processes, SMP
* multi-generational, per-process garbage collector
* asynchronous, location-transparent, message-passing IPC
* hot-code-loading
* powerful REPL and introspection tools
* new JIT compiler (BEAM bytecode to machine code)

The OTP libraries

* Erlang's "standard library"
* dozens of modules providing various typical stdlib stuff
* core of which are for supervision trees
  - `gen_server`
  - supervisors

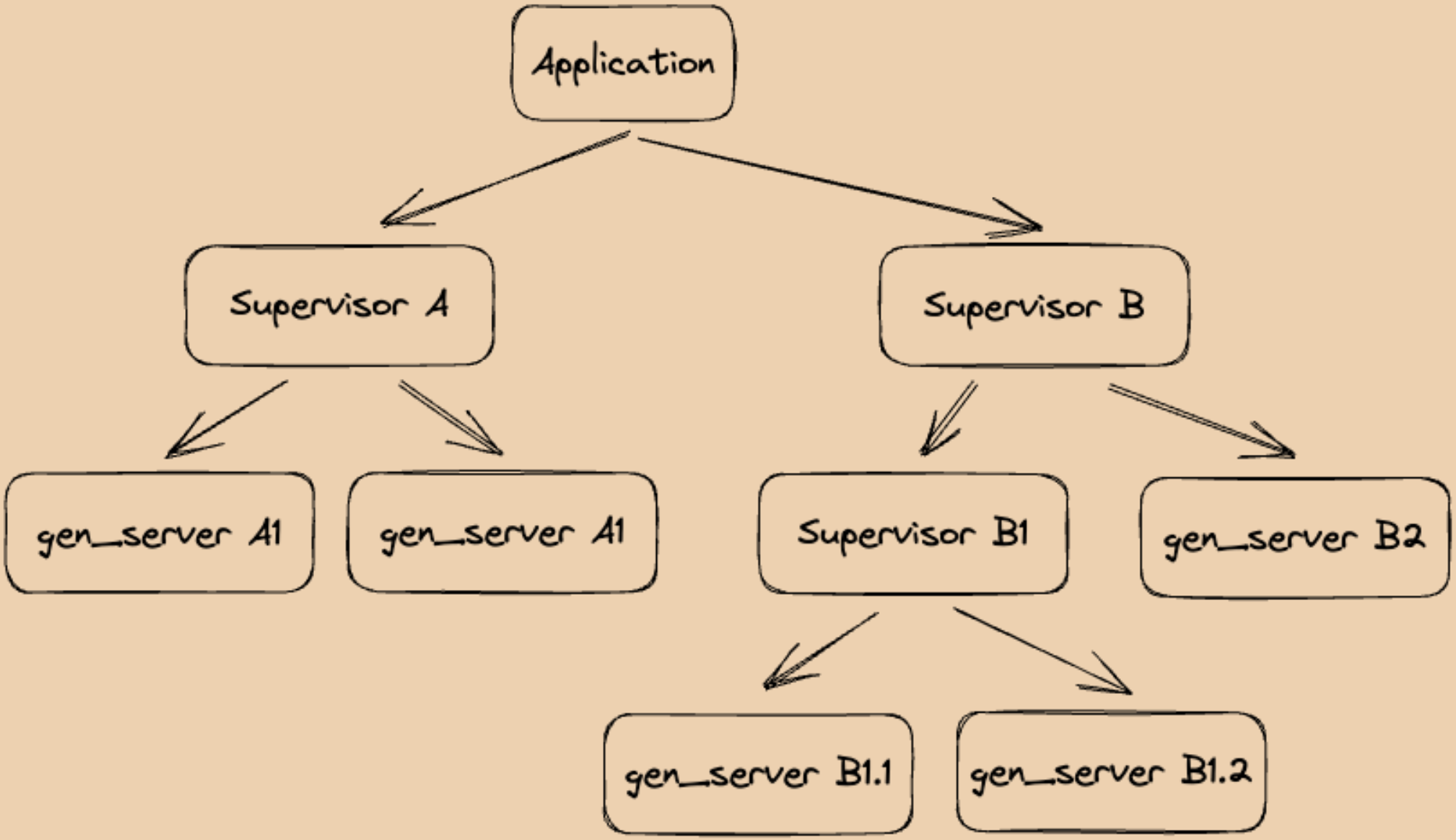
gen_server

* generic server process
* implements a behaviour with callbacks:
  - initialisation
  - handling calls (known synchronous messages)
  - handling casts (known asynchronous messages)
  - handling out-of-band messages (e.g. timers)
  - graceful termination
  - hot-code-loading

supervisor

* responsible for starting, stopping, and monitoring child processes
* *child specifications* dictate how child processes are restarted if they crash, whether other processes are affected by a crash, and if the supervisor itself should shut down

supervision trees



Elixir

Brief history of Elixir

* created by José Valim starting in 2012
* inspired by Erlang, Ruby, and, to a lesser extent, Lisp
* like Erlang, the language is quite stable

Features of Elixir

* Ruby-like syntax while retaining most of Erlang's semantics
  - ..., immutable data, HoF, side-effects anywhere, dynamically-typed, ...
* interoperate with Erlang
* hygienic macros
* highly ergonomic build tool: `mix`
* modern package manager: `hex`
* excellent unit test tool: exunit
* opinionated formatter
* drops SSA

Pipelines (|>)

```
[10..20]
|> Enum.random()
|> Faker.Lorem.paragraphs()
|> Enum.map(fn str -> String.replace(str, ~r/\p{P}/u, " ") end)
|> Enum.map(fn str -> String.downcase(str) end)
|> Enum.join()
|> String.split()
|> Enum.group_by(fn str -> str end)
|> Enum.map(fn {str, strs} -> {str, Enum.count(strs)} end)
|> Enum.sort_by(fn {_, count} -> count end, &>=/2)
```

with special form

```
# instead of this
case get_client(creds) do
  {:ok, client} -> case get_api_data(client) do
                     {:ok, data} -> write_to_db(data)
                     {:error, :api_error} -> ...
                   end
  {:error, :no_client} -> ...
end


# we can do this
with {:ok, client} <- get_client(creds),
     {:ok, data} <- get_api_data(client) do
  write_to_db(data)
else
    {:error, :no_client} -> ...
    {:error, :api_error} -> ...
end
```

Protocols

mechanism to achieve polymorphism over multiple data types.

```
# The Enum module can count maps and lists because they both
# implement the Enumerable protocol

%{:a => "a", :b => "b"} |> Enum.count
# 2
[1,2,3,4,5] |> Enum.count
# 5
```

Typespecs

(Erlang has these too)

```
@type foobar :: :foo | {:bar, term()}
@type my_map :: %{
        :a => number(),
        :b => binary(),
        :c => foobar(),
        optional(String.t()) => %SomeStruct{}
      }
```

Success Typing with Dialyzer

Dialyzer

* DIscrepancy anALYZer for ERlang
* static-analysis tool for Erlang, Elixir, and BEAM files
* Success Typing - optimise for avoiding false-positives

What is Success Typing?

* technique to check programs for type inconsistencies
* does not require the type annotations, but they help
* only considers a program in error if it is certain there is an inconsistency

Using Dialyzer in Elixir

```
# mix.exs
{:dialyxir, "~> 1.3", only: [:dev], runtime: false}

mix dialyzer # pass flags like --missing_return
```

Type example

```elixir
defmodule DialyzerExample.TypeExample do
  def add(a, b), do: a + b

  def concat(str1, str2), do: str1 <> str2

  @spec wat(number(), String.t()) :: number()
  def wat(a, b), do: a + b

  def run_add, do: add(1, :two)
  def run_concat, do: concat(:not_a_string, "suffix")
end
```

Type example - wat/2

```
lib/type_example.ex:6:invalid_contract
The @spec for the function does not match the success typing of the function.

Function:
DialyzerExample.TypeExample.wat/2

Success typing:
@spec wat(number(), number()) :: number()
```

Type example - add/2 (concat/2 similarly)

```
lib/type_example.ex:9:no_return
Function run_add/0 has no local return.
------------------------------------------------------------------------
lib/type_example.ex:9:call
The function call will not succeed.
DialyzerExample.TypeExample.add(1, :two)

will never return since the 2nd arguments differ
from the success typing arguments:

(number(), number())
```

Type example - extra warning flags

* mix dialyzer --extra_return --missing_return

```
defmodule DialyzerExample.ExtendedExample do
  @spec extra_return(integer()) :: :even | :odd | :zero
  def extra_return(a) do
    if rem(a, 2) == 0, do: :even, else: :odd
  end

  @spec missing_return(integer()) :: :even | :odd
  def missing_return(a) do
    cond do
      a == 0 -> :zero
      rem(a, 2) == 0 -> :even
      rem(a, 2) != 0 -> :odd
    end
  end
end
```

Type example - extra_return

```
lib/extended_example_a.ex:2:extra_range
The type specification has too many types for the function.

Function:
DialyzerExample.ExtendedExample.extra_return/1

Extra type:
:zero

Success typing:
:even | :odd
```

Type example - missing_return

```
lib/extended_example_a.ex:7:missing_range
The type specification is missing types returned by function.

Function:
DialyzerExample.ExtendedExample.missing_return/1

Type specification return types:
:even | :odd

Missing from spec:
:zero
```

Match example - inexhaustive function clauses

```elixir
defmodule DialyzerExample.MatchExampleA do
  @type tag :: :foo | :bar
  @type tagged_type :: {tag(), term()}

  @spec handle(tagged_type()) :: term()
  def handle(tagged_data)

  def handle({:foo, data}) do
    IO.inspect(data, label: "got foo")
  end

  # where do we handle :bar??
end
```

Match example - call unmatched function clause

```elixir
defmodule DialyzerExample.MatchExampleB do
  @type tag :: :foo | :bar
  @type tagged_type :: {tag(), term()}

  @spec handle(tagged_type()) :: term()
  def handle(tagged_data)

  def handle({:foo, data}),
    do: IO.inspect(data, label: "got foo")

  # Let's actually call handle/1
  def run do
    handle({:foo, "foo"})
    handle({:bar, "bar"})
  end
end
```

Match example - `mix dialyzer`

```
lib/match_example_b.ex:12:no_return
Function run/0 has no local return.
---------------------------------------------------------------------------
lib/match_example_b.ex:14:call
The function call will not succeed.

DialyzerExample.MatchExampleB.handle({:bar, <<98, 97, 114>>})

will never return since the 1st arguments differ
from the success typing arguments:

({:foo, _})

---------------------------------------------------------------------------
```

Case Example - non-exhaustive case

```elixir
defmodule DialyzerExample.CaseExample do
  @type tag :: :foo | :bar
  @type tagged_type :: {tag(), term()}

  @spec handle(tagged_type()) :: term()
  def handle(tagged_data)

  def handle({tag, data}) do
    case tag do
      :foo -> IO.inspect(data, label: "got foo")
    end
  end

  def run do
    handle({:baz, "baz"})
  end
end
```

Tips for using Dialyzer

* start using at the beginning of a project
* run the mix task to create Persistent Lookup Table files
* cache PLTs for Erlang, Elixir, and deps in CI
  - https://github.com/team-alembic/staple-actions/tree/main/actions/mix-dialyzer

Gradual Typing with Gradualizer / Gradient

What is Gradual Typing?

* a form of type system that combines static and dynamic types
* a gradually-typed program annotates parts of its code with types
* some of the program will then have known types, other parts will have a `unknown` type
* a gradual type checker ensures parts of values with known types are consistent

Gradualizer

* https://github.com/josefs/Gradualizer
* gradual type checker for Erlang
* relies on type specs
* will only check where types are annotated and known
* by default, does not infer types
* much faster than Dialyzer and no PLTs!

Gradient - an Elixir front-end for Gradualizer

```
# mix.exs
{:gradient, github: "esl/gradient",
  only: [:dev], runtime: false}
```

Type example

```elixir
defmodule GradientExample.TypeExample do
  @spec add(number(), number()) :: number()
  def add(a, b), do: a + b

  def concat(str1, str2), do: str1 <> str2

  @spec wat(number(), String.t()) :: number()
  def wat(a, b), do: a + b

  def run_add, do: add(1, :two)
  def run_concat, do: concat(:not_a_string, "suffix")
end
```

Type example - output

```
lib/type_example.ex: The variable on line 8 is expected to have type number()
but it has type binary()
6
7   @spec wat(number(), String.t()) :: number()
8   def wat(a, b), do: a + b
9
10   def run_add, do: add(1, :two)

lib/type_example.ex: The atom on line 10 is expected to have type number()
but it has type :two
8   def wat(a, b), do: a + b
9
10   def run_add, do: add(1, :two)
11   def run_concat, do: concat(:not_a_string, "suffix")
12 end
```

Inference example - `mix gradient --infer`

```elixir
defmodule GradientExample.InferExample do
  def wat() do
    1 + "2"
  end
end
```

Infer example - output

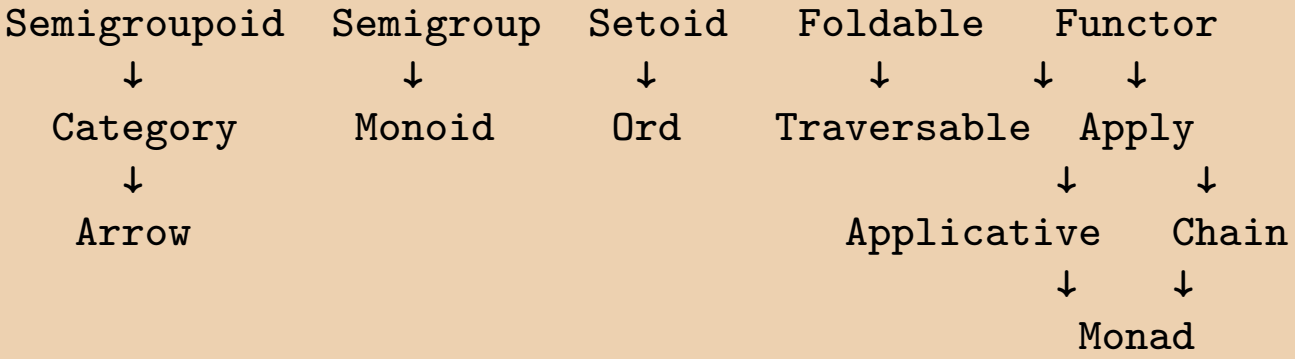
```
lib/infer_example.ex: The operator '+' on line 3 is requires numeric arguments,
but has arguments of type 1 and binary()

Total errors: 1
```

Witchcraft and Algae

Typeclasses with Witchcraft

* https://github.com/witchcrafters/witchcraft
* provides a typeclass hierarchy similar to Haskell, Scala, or FP-TS
* respective operators such as map, apply, lifts, etc
* tools to create typeclass instances for custom data types

Witchcraft Typeclass Hierarchy



Algebraic Data Types with Algae

* https://github.com/witchcrafters/algae
* builds on top of Witchcraft to provide tools to create ADTs
* ADTs: sum and product types
  - sum type: Lists, Trees, Maybe / Option, Either
  - product types: records, maps

Contrived example - TaskEither ADT

```
defmodule TaskEither do
  import Algae

  defsum do
    defdata(Left :: any())
    defdata(Right :: any())
  end
end
```

Contrived example - TaskEither typeclasses

```
definst Witchcraft.Functor, for: TaskEither.Left do
  def map(left, _), do: left
end

definst Witchcraft.Functor, for: TaskEither.Right do
  def map(%Right{right: data}, fun),
   do: data |> fun.() |> Right.new()
end

# Apply, Applicative, Chain, Monad ...
```

Contrived example - TaskEither execution

```
# >>>/2 is the bind function from Haskell's Monad typclass
> g = fn _t -> TaskEither.Left.new(fn -> :error end) end
#Function<42.3316493/1 in :erl_eval.expr/6>
> f = fn t -> TaskEither.Right.new(fn -> t.() end) end
#Function<42.3316493/1 in :erl_eval.expr/6>
> result = a >>> f >>> g
%TaskEither.Left{left: #Function<43.3316493/0 in :erl_eval.exp
```

Thoughts on witchcraftery

* not very active
* straying from idomatic Elixir (e.g `{:ok, data()} | {:error, String.t()}`)
* need to study typeclases, perhaps more of an intellectual curiosity

Sneak-peak at the new Elixir Type System

* PhD project to introduce a native type system
  - Guillaume Duboc under the supervision of Giuseppe Castagna and José Valim
  - Paris Cité University and French National Centre for Scientific Research
* Semantic Subtyping aka Set-Theoretic types
* research is currently underway
  - draft research paper
  - Elixir Conf EU talk
  - demo playground on fly.io

Features

* new syntax for type annotations with some reference to the `@spec` syntax
* type variables
* understands maps, protocols, guards, and pattern matching
* gradual typing with a `dynamic()` type and `strong arrows`

Syntax of a Set-Theoretic type annotation

```
negate :: (integer() -> integer())
  and (boolean() -> boolean())
def negate(x) do ...
```

Type variables

```
map :: ([a], (a -> b) -> [b] when a: term(), b: term())
def map([h | t], fun), do: [fun.(h) | map(t, fun)]
def map([], _fun), do: []

reduce :: ([a], b, (a, b -> b) -> b
  when a: term(), b: term())
def reduce([h | t], acc, fun),
  do: reduce(t, fun.(h, acc), fun)
def reduce([], acc, _fun), do: acc
```

Protocols

```
Enumerable.t(a) and Collectible.t(a) # will be a thing
```

Gradual typing and `dynamic()`

* sometimes typechecker introduces `dynamic()` into the typing
* there will be guarantees of soundness of the typing or a guarantee of a runtime type error using guards (e.g. `is_integer()`)

Thoughts

* early days, but very exciting times
* set operators `and` and `or` take some getting used to
* the dynamic-typing aspect is a bit unclear (to me)
* will be interesting to see what happens to dialyzer and gradient

Resources

* https://www.irif.fr/users/gduboc/index
* https://www.irif.fr/_media/users/gduboc/elixir-types.pdf
* https://www.youtube.com/watch?v=gJJH7a2J9O8
* https://typex.fly.dev/

Typex demo

What did we not get to talk about?

* Gleam
* Purescript
* Property-Based Testing