BROOKLYN ZELENKA

USE THE @TYPES, LUKE

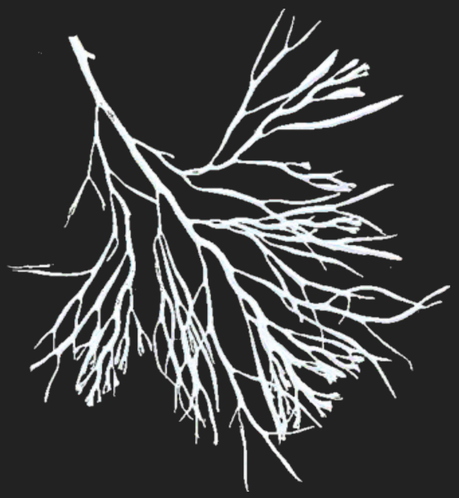
SHAMELESS SELF PROMOTION. SHAMELESS.

- Organizes C&C, VanFP, VanEE
  - Last VanEE in person
- Co-founder of Robot Overlord Inc, developer at MetaLab
- Author of Quark, Algae, and Witchcraft

Witch★craft

Algae

Bootstrapped
algebraic data types
for Elixir



WHAT WE'RE GOING TO COVER

- Why types have a bad rep
- Elixir's type system
- Typespecs
- Dialyxir
- Defining our own types
- Structs
- Parametric polymorphism
- Q&A

WHY TYPES HAVE A BAD REP

THERE ARE TWO KINDS OF TYPE SYSTEMS.
SOME ARE FOR THE COMPILER.
OTHERS ARE FOR THE PROGRAMMER.

Overheard at LambdaConf 2015

TYPES FOR THE COMPILER (EX. C & JAVA)

- Boilerplate
- Don’t add to the code expressivity
- Source of (annoying) warnings

TYPES FOR PROGRAMMERS (EX. HASKELL, ELIXIR, SWIFT)

- Annotate the *meaning* of a piece of code
- Help to structure your code
- Double as documentation
- Catch some bugs before you run your code!

ELIXIR'S TYPE SYSTEM

ELIXIR HAS “WEAK” DYNAMIC TYPES.

“Weak” doesn’t imply bad

WEAK, DYNAMIC TYPES

- Type inference *at run time*
- Static analysis tools do exist (Dialyxr)
- Determine code behaviour through parametric polymorphism

BUILT-IN TYPES

- There's quite a few
- Some contain others
  - ex. **integer** is contained in **number**

```
Type :: any          # the top type, the set of all terms
      | none         # the bottom type, contains no terms
      | pid
      | port
      | reference
      | Atom
      | Bitstring
      | float
      | Fun
      | Integer
      | List
      | Map
      | Tuple
      | Union
      | UserDefined # Described in section "Defining a type"

Atom :: atom
      | ElixirAtom # `:foo`, `:bar`, ...

Bitstring :: <<>>
           | << _ :: M >>              # M is a positive integer
           | << _ :: _ * N >>          # N is a positive integer
           | << _ :: M, _ :: _ * N >>

Fun :: (... -> any)    # any function
     | (... -> Type)   # any arity, returning Type
     | (() -> Type))
     | (TList -> Type)

Integer :: integer
         | ElixirInteger                # ..., -1, 0, 1, ... 42 ...
         | ElixirInteger..ElixirInteger # an integer range

List :: list(Type)                          # proper list ([]-terminated)
      | improper_list(Type1, Type2)         # Type1=contents, Type2=termination
      | maybe_improper_list(Type1, Type2)   # Type1 and Type2 as above
      | nonempty_list(Type)                 # proper non-empty list
      | []                                  # empty list
      | [Type]                              # shorthand for list(Type)
      | [...]                               # shorthand for nonempty_list()
      | [Type, ...]                         # shorthand for nonempty_list(Type)
      | [Keyword]

Map :: map()            # map of any size
     | %{}              # map of any size
     | %Struct{}        # struct (see defstruct/1)
     | %Struct{Keyword}
     | %{Keyword}
     | %{Pairs}

Tuple :: tuple                  # a tuple of any size
       | {}                     # empty tuple
       | {TList}
       | record(Atom)           # record (see Record)
       | record(Atom, Keyword)

Keyword :: ElixirAtom: Type
         | ElixirAtom: Type, Keyword

Pairs :: Type => Type
       | Type => Type, Pairs

TList :: Type
       | Type, TList

Union :: Type | Type
```

TYPESPECS

OPTIONAL, GRADUAL TYPING

- ex. **@spec add(integer, integer) :: integer**
- Similar syntax to @doc, etc.
  - Lives outside of the function definition
  - Has an “@“ before it
  - Generates documentation

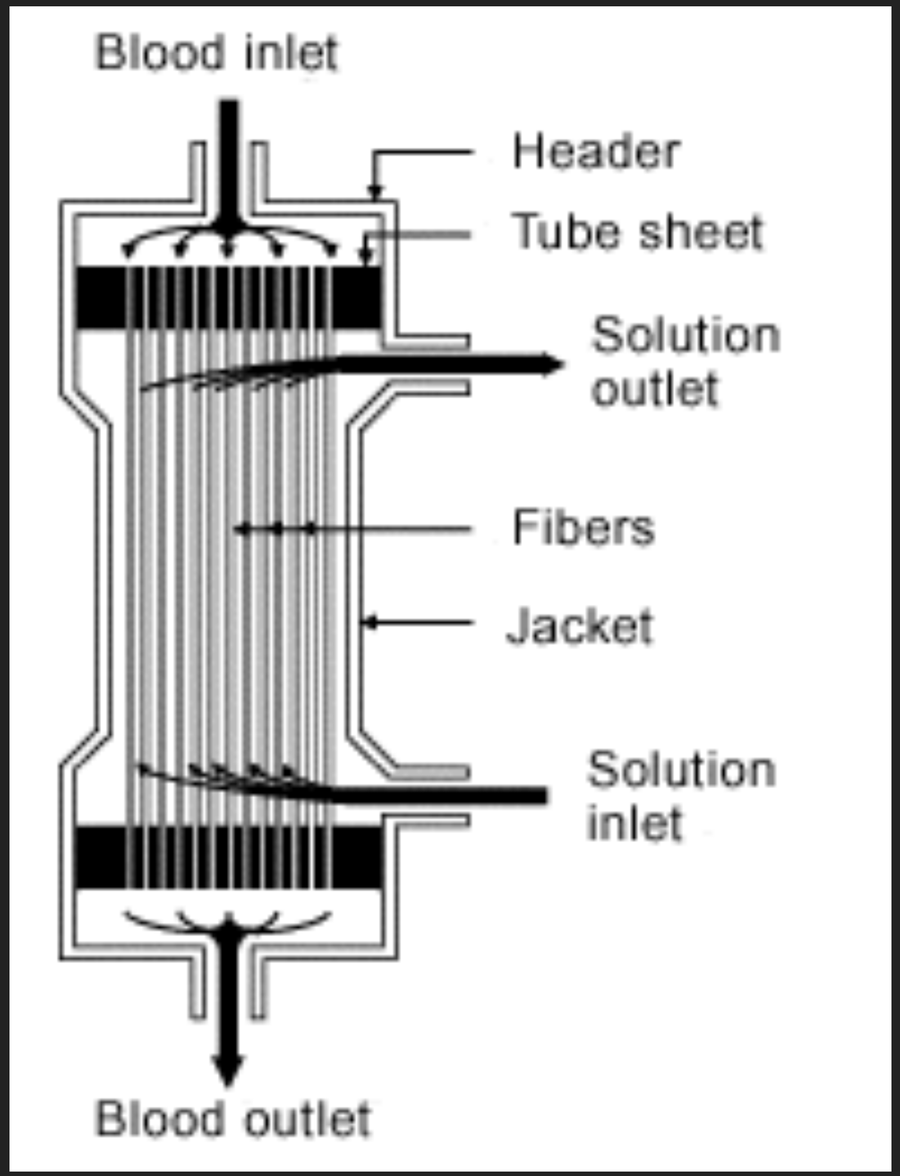
DIALYXIR (DIALYZER)

DIALYZER IS A STATIC ANALYSIS TOOL THAT IDENTIFIES SOFTWARE DISCREPANCIES SUCH AS TYPE ERRORS, UNREACHABLE CODE, UNNECESSARY TESTS, ETC IN SINGLE ERLANG MODULES OR ENTIRE (SETS OF) APPLICATIONS.

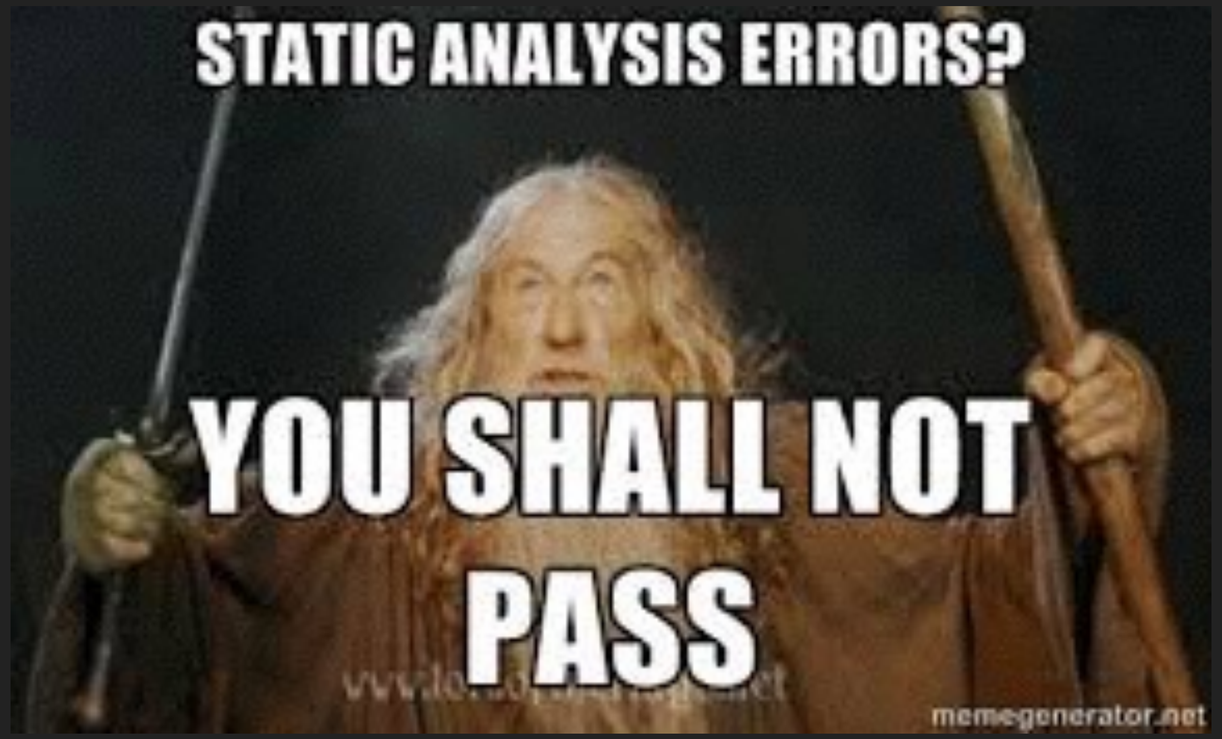
erlang.org

HOW TO DIALYZE

- Add to mix.exs dependencies
- mix deps.get
- mix deps.compile
- mix dialyzer



EXAMPLE OUTPUT

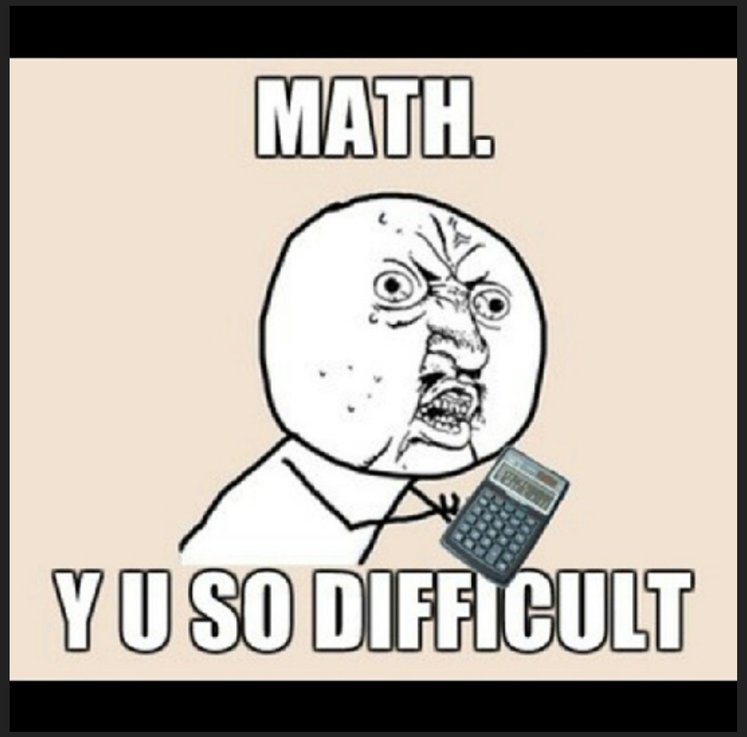



```
✘ ~/D/V/quark  master ±  mix dialyzer
Starting Dialyzer
dialyzer --no_check_plt --plt /Users/expede/.dialyxir_core_18_1.2.1.
plt -Wunmatched_returns -Werror_handling -Wrace_conditions -Wundersp
ecs /Users/expede/Documents/Volunteer/quark/_build/dev/lib/quark/ebi
n
  Proceeding with analysis...
quark.ex:62: Type specification 'Elixir.Quark':m(fun()) -> fun() is
a supertype of the success typing: 'Elixir.Quark':m(fun()) -> fun((_
) -> any())
compose.ex:73: Type specification 'Elixir.Quark.Compose':'<|>'(fun()
,fun()) -> fun() is a supertype of the success typing: 'Elixir.Quark
.Compose':'<|>'(fun(),fun()) -> fun((_) -> any())
curry.ex:69: Overloaded contract for 'Elixir.Quark.Curry':uncurry/2
has overlapping domains; such contracts are currently unsupported an
d are simply ignored
sequence.ex:1: The specification for 'Elixir.Quark.Sequence':'__prot
ocol__'/1 states that the function might also return 'true' but the
inferred return is 'Elixir.Quark.Sequence' | 'false' | [{'origin',1}
 | {'pred',1} | {'succ',1},...]
Unknown functions:
  'Elixir.Quark.Sequence.Atom':'__impl__'/1
  'Elixir.Quark.Sequence.BitString':'__impl__'/1
  'Elixir.Quark.Sequence.Float':'__impl__'/1
  'Elixir.Quark.Sequence.Function':'__impl__'/1
  'Elixir.Quark.Sequence.List':'__impl__'/1
  'Elixir.Quark.Sequence.Map':'__impl__'/1
  'Elixir.Quark.Sequence.PID':'__impl__'/1
  'Elixir.Quark.Sequence.Port':'__impl__'/1
  'Elixir.Quark.Sequence.Reference':'__impl__'/1
  'Elixir.Quark.Sequence.Tuple':'__impl__'/1
 done in 0m1.45s
```

DEFINING OUR OWN TYPES

CUSTOM TYPE EXAMPLE

```
@type number_with_remark :: {number, String.t}
@spec add(number, number) :: number_with_remark
def add(x, y), do: {x + y, "You need a calculator to do that?"}
```



STRUCTS

STRUCT TYPES

- Multiple named fields
- Can get their own type

```elixir
defmodule Algae.Maybe do
  @type t :: Just.t | Nothing.t

  defmodule Nothing do
    @type t :: %Nothing{}
    defstruct []
  end

  defmodule Just do
    @type t :: %Just{just: any}
    defstruct [:just]
  end
end
```

PARAMETRIC POLYMORPHISM

PROTOCOLS

- Give a definition of a function name per data type or struct “type”

```
defimpl Witchcraft.Monoid, for: List do
  def identity(_list), do: []
  def append(as, bs), do: as ++ bs
end

defimpl Witchcraft.Monoid, for: Map do
  def identity(_map), do: %{}
  def append(ma, mb), do: Dict.merge(ma, mb)
end
```

Q&A TIME