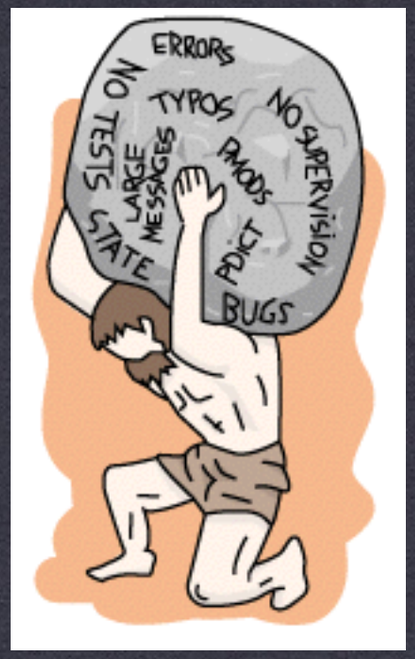
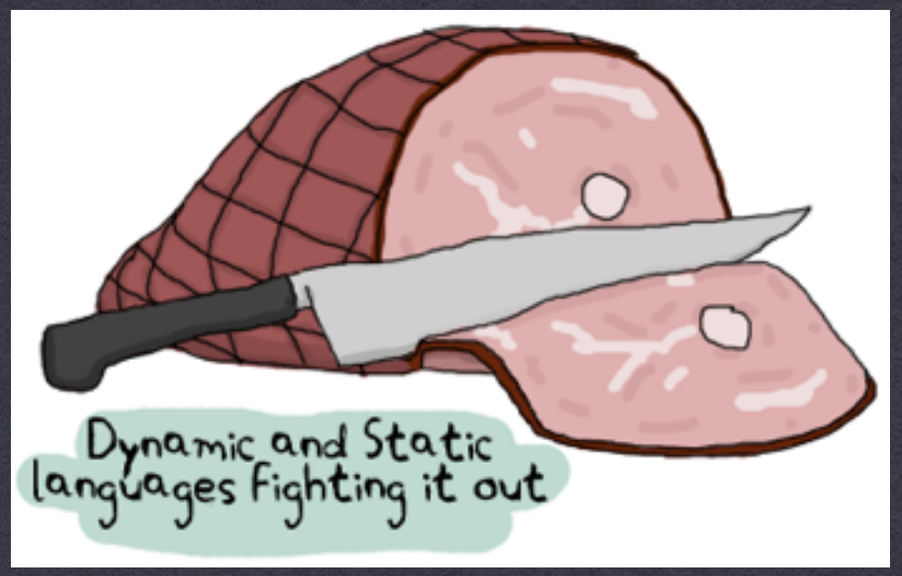
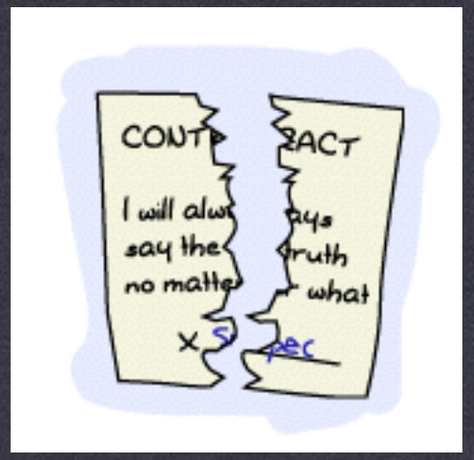


Jason Voegele
@jvoegele

Basho Technologies





DIALYZER

OPTIMISTIC TYPE CHECKING FOR ERLANG AND ELIXIR

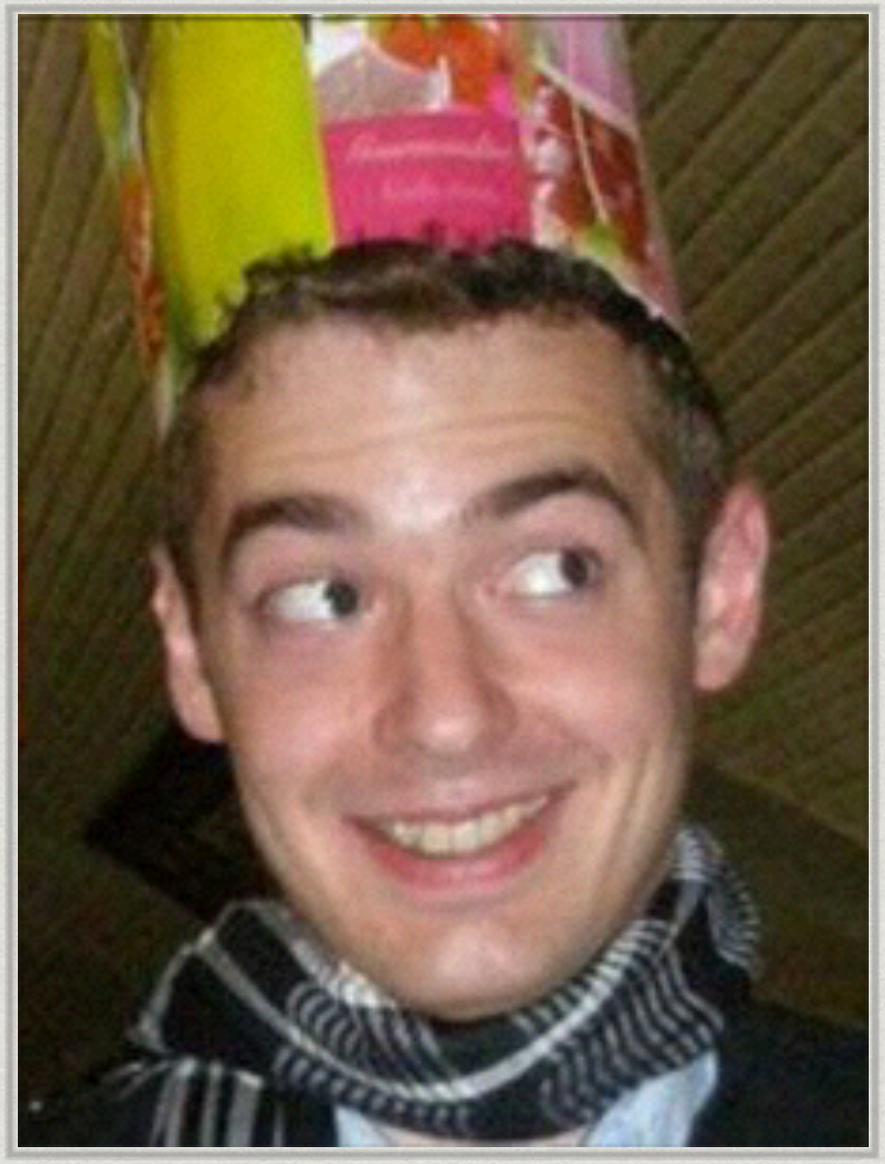
All illustrations from http://LearnYouSomeErlang.com

Holy Wars

- **Emacs vs. Vi(m)**
  - VILE, EVIL, Spacemacs
- **Tabs vs. spaces**
  - spaces won
- **KDE vs. Gnome**
  - The year of Linux on the desktop never happened

Static vs. Dynamic Typing

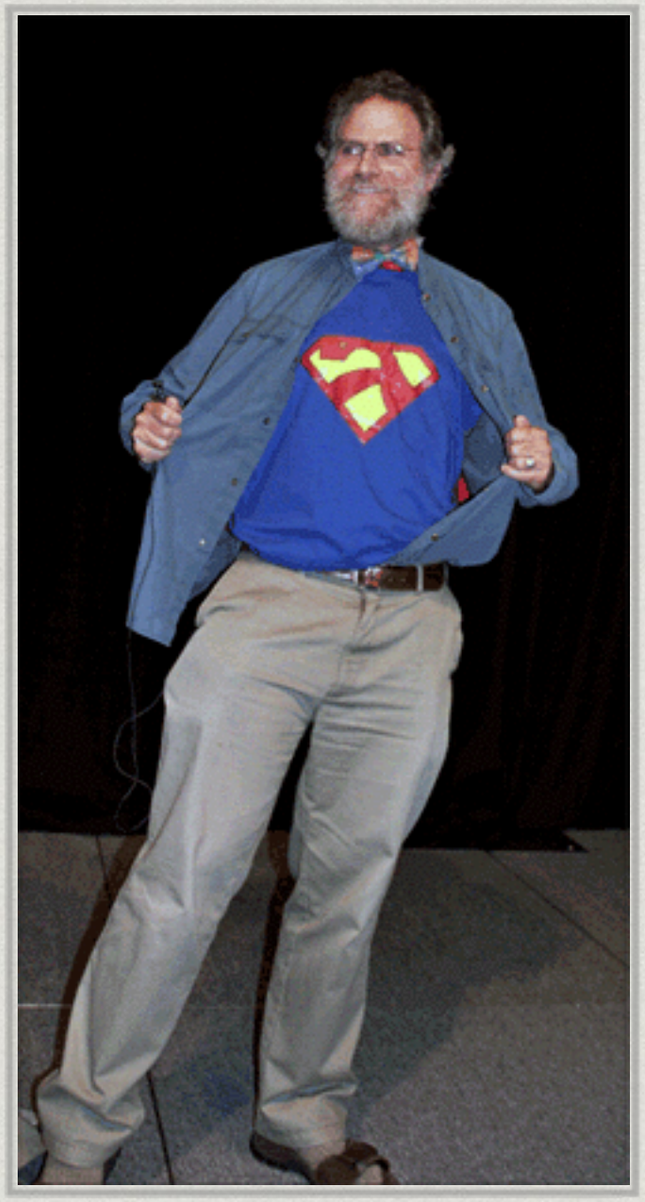
- Gary Bernhardt on *Ideology* (Strange Loop 2015):
  - Type bigots believe that *correctness comes exclusively from* ***categories***.
  - Test bigots believe *correctness comes exclusively from* ***examples***.



Gradual Typing

- Type system where *some* variables and functions have declared types
- Looseness and flexibility of dynamic typing:
  - “Duck Typing” is still a viable option
  - Don’t have to get all the types correct up front
- Benefits similar to static typing:
  - Check for consistency and correctness
  - Aid in documentation and legibility

Type Checking in Erlang



- Erlang is a dynamically but strongly typed language.
  - 5 + “2” -> error
- Pattern matching and guard sequences provide rudimentary type checking (at runtime).
- 1997: Simon Marlow’s and Philip Wadler’s failed attempt to add static typing to Erlang
  - Many large Erlang systems running in production.
  - Type checker must respect Erlang philosophy and idioms.

Enter the Dialyzer

- ★**DI**screpancy **A**nal**YZ**er for **ER**lang programs
- ★Static analysis tool that performs type checking
- ★Type inference plus optional type specifications (gradual typing)
- ★Optimistic type checking model based on “success typing”

Success Typing

- “A success typing is a type signature that over-approximates the set of types for which the function can evaluate to a value. The domain of the signature includes all possible values that the function could accept as parameters, and its range includes all possible return values for this domain.” — Lindahl & Sagonas

- **TL;DR — Optimistic, “Never cry wolf”**

YOU DAMN ELIXIR KIDS!

Type Inference

```
def add(x, y), do: x + y
  # add(number,number) :: number

def divide(x, y), do: x / y
  # divide(number,number) :: float

def and(false, _), do: false
def and(_, false), do: false
def and(true,true), do: true
  # and(any,any) :: boolean
```

Function Type Specs

```
@spec add(number,number) :: number
def add(x, y), do: x + y

@spec divide(number,number) :: float
def divide(x, y), do: x / y

@spec and(boolean,boolean) :: boolean
def and(false, _), do: false
def and(_, false), do: false
def and(true, true), do: true
```

Basic Built-In Types

- `boolean`
- `char`, `binary`, `String.t`
- `atom`
- `pid`, `port`, `reference`
- Literal values: `true`, `:ok`, `42`
- `any` (the “top” type, also known as `term`)
- `none` (the “bottom” type)

Numeric Types

- `integer`
- `float`
- `number` (i.e. `integer | float`)
- `pos_integer`, `neg_integer`, `non_neg_integer`
- Ranges: `1..12`

Lists and Tuples

- Lists:
  - `list`, `[ ]`
  - `list(atom)`, `[atom]`
  - `nonempty_list`, `[…]`
  - `nonempty_list(integer)`, `[integer, …]`
- Tuples: `tuple`, `{}`, `{atom, binary}`

Maps, Structs, & Compound

- Basic maps: `map`, `%{}`, `%{…}`
- Map with required key `:key` with value of `type`:
  - `%{key: type}`
- Map with keys of `type1` with values of `type2`:
  - `%{required(type1) => type2}`
  - `%{optional(type1) => type2}`
- Structs: `%SomeStruct{}`, `%SomeStruct{key: type}`
- Compound: `[{atom, any}]`, `%{atom => [binary]}`

Types to Represent Functions

- 0-arity: `(() -> integer)`
- 1-arity: `(atom -> pid)`
- 2-arity: `(%{atom, integer}, atom -> integer)`
- Any arity: `(… -> boolean)`

Defining Custom Types

```elixir
defmodule PlayingCards do
  @type suit :: :spades | :hearts
              | :diamonds | :clubs
  @type value :: 2..10
               | :jack | :queen | :king | :ace
  @type card :: {suit, value}
  @type deck :: [card, ...]

  @spec suit(card) :: suit
  def suit({s, _v}) do
    s
  end

  def broken do
    suit({10, :spades})
  end
end
```

DEMO

OPAQUE STACK
DIALYZER AND MIX
FUNCTION COMPOSITION

Other Dialyzer Features

- Overloading type specs
- Parameterized & polymorphic types
- Type variables
- Tagged tuples
- Type specs for structs

An Assessment

- Gradual typing is a good compromise
- Type specs make code easier to read
- Can find real errors.
- Cannot find ***all*** real errors.
- Error messages are hard to read.
- Elixir integration is lacking:
  - Type specs for structs
  - Cannot analyze Elixir *script* files