爱因斯坦搞炼丹

Elixir: A Haskeller's Perspective

About me

知乎@祖与占 (FP/Haskell/PL)

CRUD Programmer

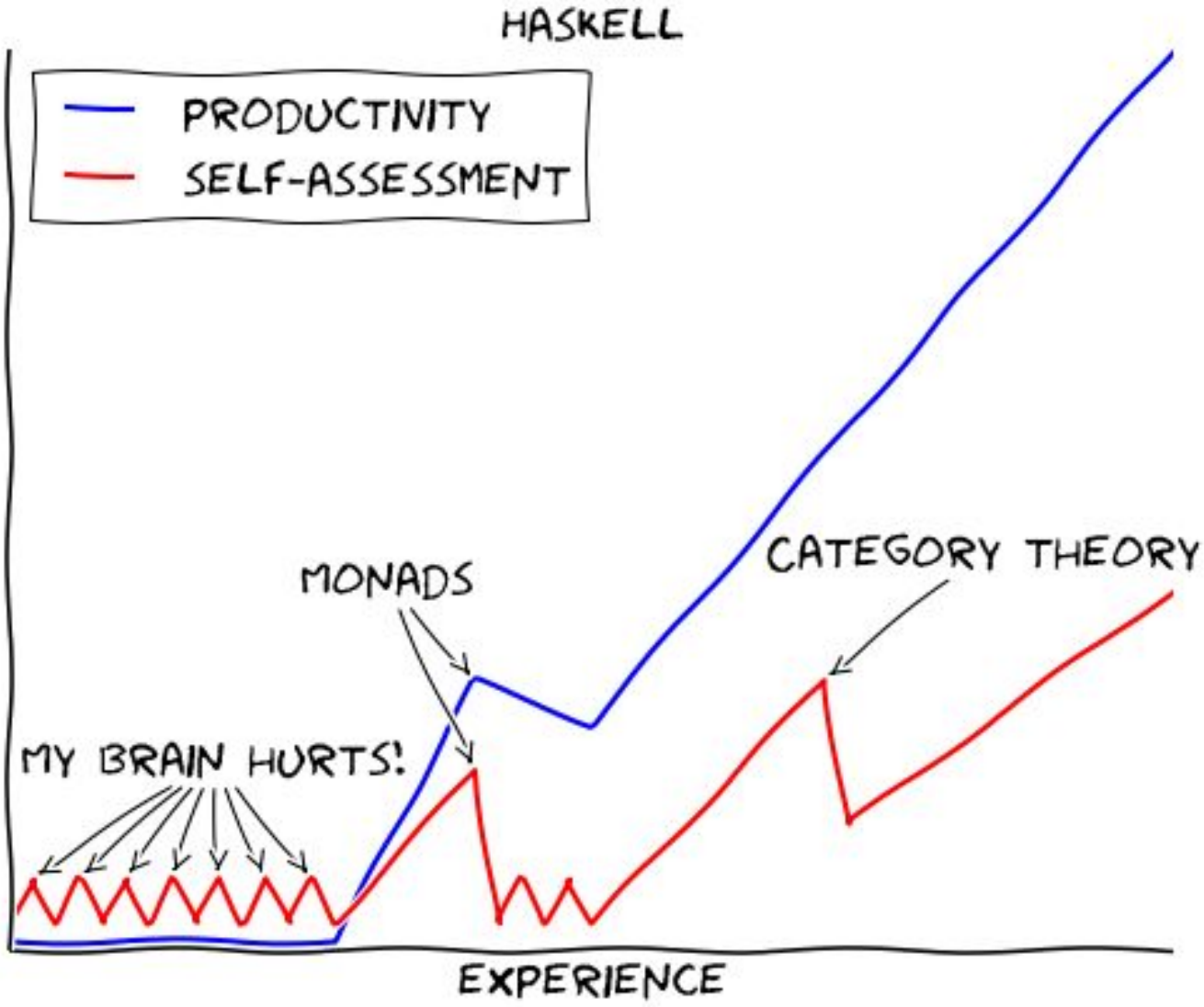
Programming Language Fanboy (PL by UW @Coursera)

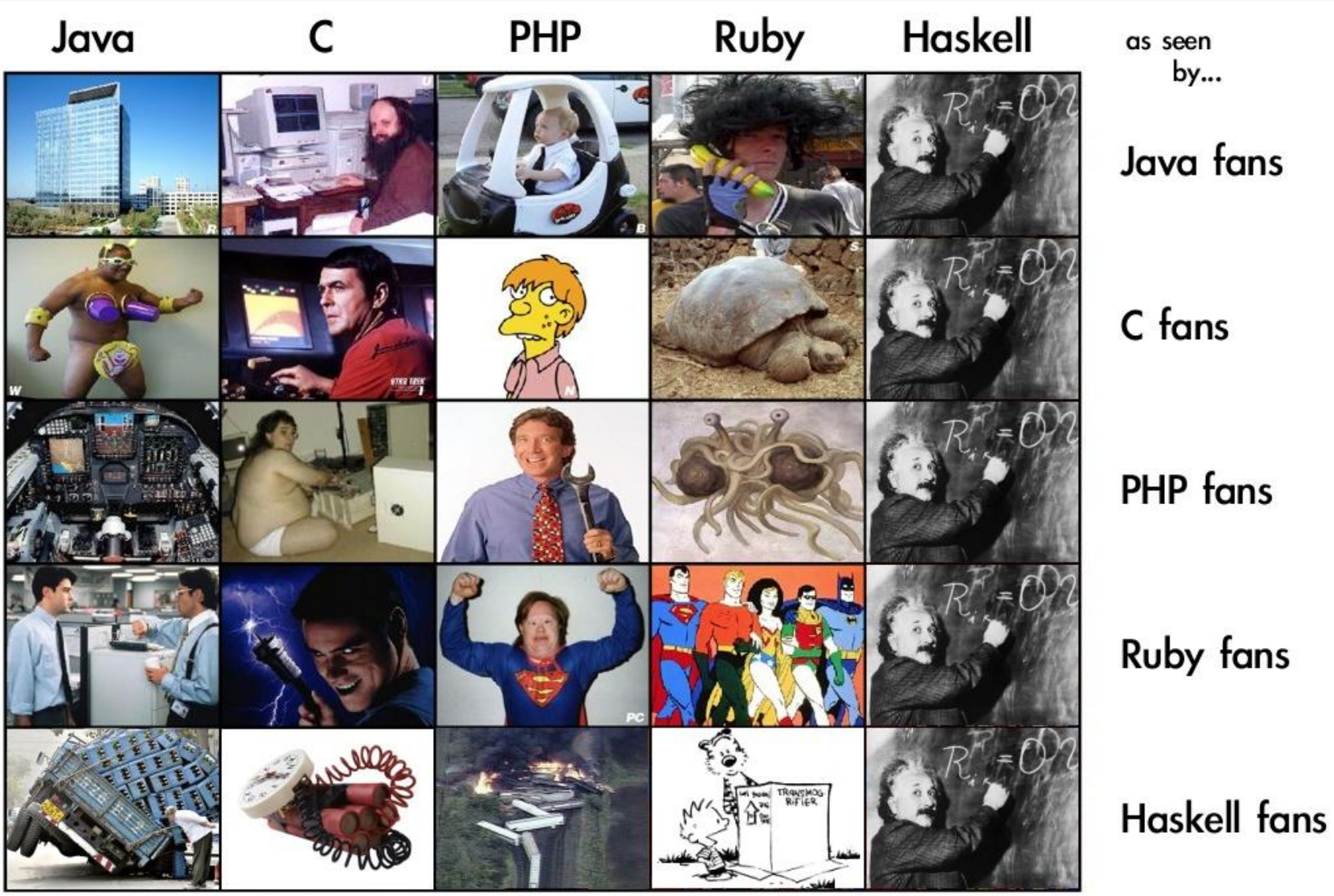
<3 Haskell

Some (webdev) experience of Erlang/Elixir

About this talk

- Not a Machine Learning talk by a physicist
- Not a Introduction of Elixir
  - 欧阳继超 - 函数式 Ruby 编程
  - 邱华 - Rubyists 可以从 Elixir 学到什么
- Elixir: A Haskeller's ~~Perspective~~ Déjà vu


HASKELL
PRODUCTIVITY
SELF-ASSESSMENT
MONADS
CATEGORY THEORY
MY BRAIN HURTS!
EXPERIENCE


Java
C
PHP
Ruby
Haskell
as seen by...
Java fans
C fans
PHP fans
Ruby fans
Haskell fans

elixir

noun [C usually singular] · UK /iˈlɪk.səʳ/ US /iˈlɪk.sɚ/ LITERARY

a substance, usually a liquid, with a magical power to cure, improve, or preserve something

（通常为液体的）万灵药，灵丹妙药，长生不老药

It's yet another health product claiming to be the elixir ***of life/youth*** *(= something to make you live longer/stay young).*

这又是一种号称灵丹妙药的保健产品，据说有延年益寿／永葆青春之奇效。

elixir

Elixir is a dynamic, functional language designed for building scalable and maintainable applications.

elixir-lang.org

Learn Elixir in Y Minute

- Elixir -> Erlang Abstract Format -> BEAM Byte code
- Ruby like, Programmable Syntax (Macro, Compile-time, Special Forms)
- Immutable
- Everything is an expression

```
if  42 do
  "foo"
else
  "bar"
end
# => "foo"
```

```
if  42, do: "foo", else: "bar"
```

```
if(42, [{:do, "foo"}, {:else, "bar"}])
```

Learn Elixir in Y Minute

```
defmodule Foo.Baz do
    @moduledoc “example”
     def baz(arg0, arg1) do
        Enum.map([{:ok, 1}, {:err, 2}], fn {:err, n} -> {:ok, n - 1} ; a -> a end)
      end
   end
end
```

Tricks

Section & chaining:

```
irb(main):001:0> 20.+(1).*(2)
=> 42
```

```
ghci > (*2) . (+1) $ 20
42
```

```
iex > 20 |> Kernel.+(1) |> Kernel.*(2)
42
```

Tricks

Lens:

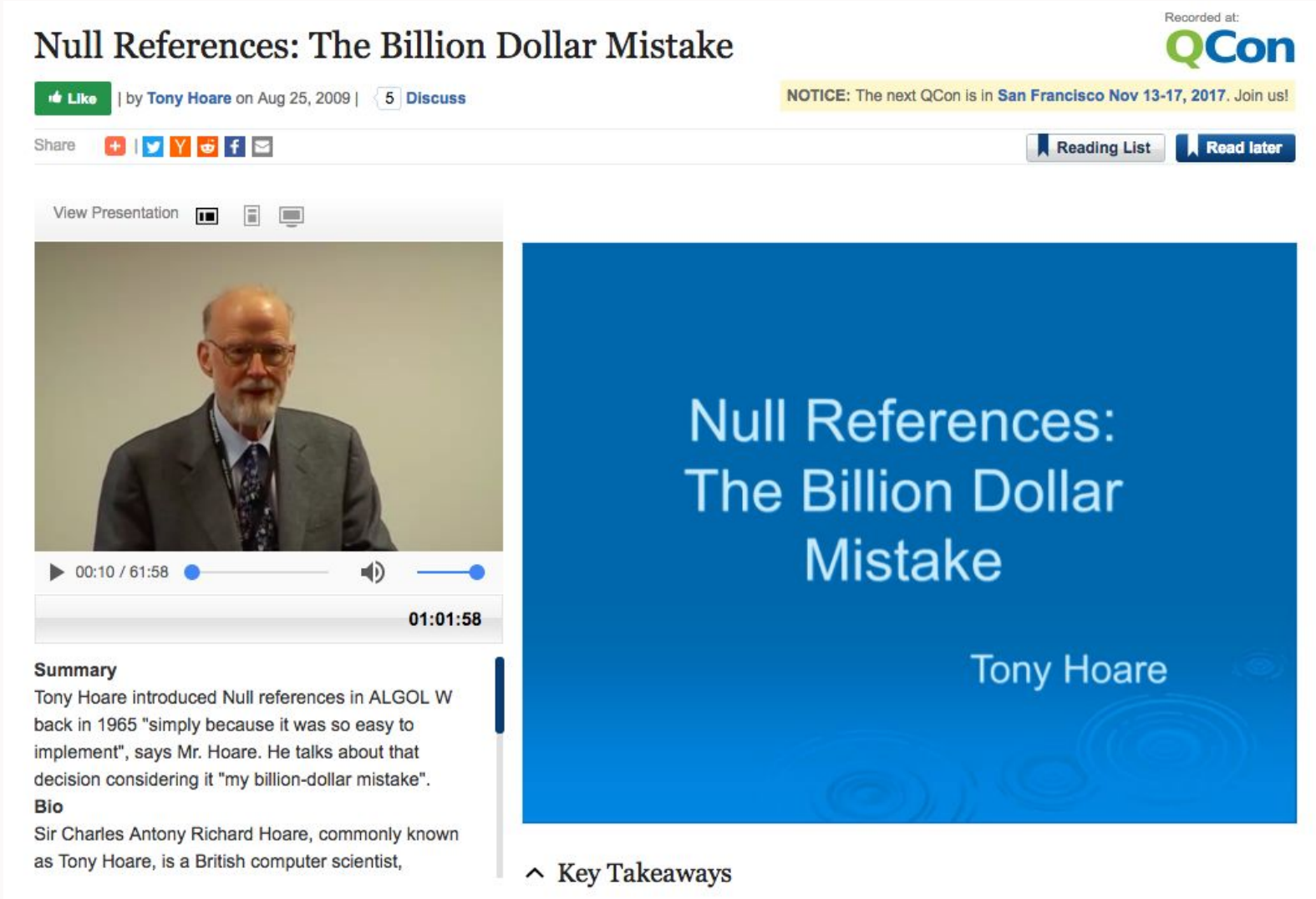
```
iex(1)> marge = %{address: %{street: %{name: "Evergreen Terrace", number: 742}}}
iex(2)> %{marge | address: %{
…(2)>        marge.address | street: %{
…(2)>       marge.address.street | name: "Fake St."
…(2)>     }
…(2)>   }
…(2)> }
…(3)> update_in(marge,  [:address, :street: :name], fn _ -> "Foo St." end)
```

small goal

The Billion Dollar Mistake

Null References: The Billion Dollar Mistake

Null References: The Billion Dollar Mistake



Maybe/Either Monad

Haskell (ADT):

```
data  Maybe a  =  Nothing | Just a

data Either a b = Left a | Right b
```

Ruby 2.3 (Safe navigation operator)

```
name = article&.author&.name
```

Scala (Inheritance):

```
abstract class Option[+A]

case class Some[+A]()  extends Option[A]

case object None extends
Option[Nothing]
```

Poor Elixir :(

- No ADT
- No Inheritance

Right way wrong train

Purescript Erlang Backend:

PureScript type	Erlang type	Notes
Tagged union	Tuple with tag element	e.g. Some 42 is {some, 42}

Elixir: Just a Tuple!

data Maybe a = Nothing | Just a

-> :error | {:ok, 42}

data Either a b = Left a | Right b

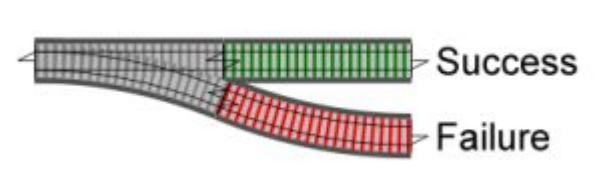
-> {:error, "fail :("} | {:ok, 42}

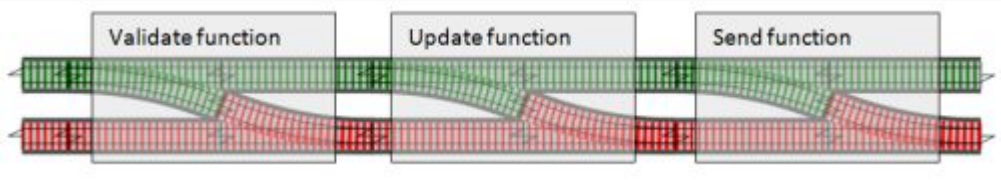
~~Callback~~ Case hell, Where is my do notation?

```
case foo of
    Just bar ->  case bar  of
                    Just baz -> case baz of
                                    Just qux -> …
                                    Nothing -> …
                    Nothing -> …
    Nothing -> ...
```

```
case foo of
  {:ok, bar} -> case bar of
                  {:ok, bar}  -> case baz of
                                   {:ok, qux} -> …
                                   :error -> …
                                 end
                  :error -> …
                End
  :error  ->  ...
end
```

Railway Oriented Programming





ROP in Elixir using With

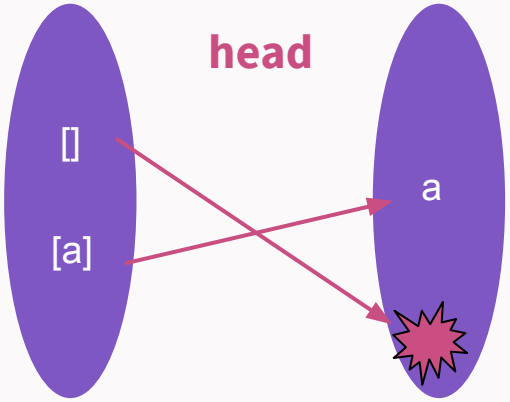
```
with {:ok,  output0} <- do_sth0(input0),
        {:ok, output1} <- do_sth1(input1)
    output1
else
    {:error, reason} -> log_error(reason)
    _ -> handle_ambigous_error
end
```

Derailment

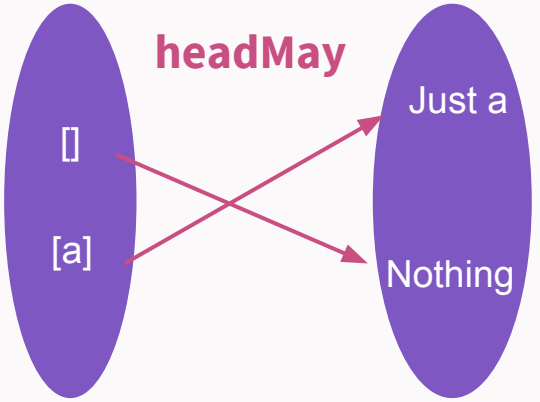
```
defmodule File do
   @spec read(Path.t) :: {:ok, binary} | {:error, posix}
    def read(path) do
      ...
    end

   @spec read!(Path.t) :: binary | no_return
   def read!(path) do  # Scheme, Ruby :D
      …
   end
end
```

Haskell Pitfalls No.1: Partial function

Partial



Total



Partial function: a legacy from Erlang

```
iex(1)> hd []
** (ArgumentError) argument error
   :erlang.hd([])
```

```
@spec hd(nonempty_maybe_improper_list(elem, any)) :: elem when elem: term
def hd(list) do
  :erlang.hd(list)
end
```

A Little Syntax

Sigil

Sigil

In computer programming, a sigil (/ˈsɪdʒəl/) is a symbol attached to a variable name, showing the variable's datatype or scope, usually a prefix, as in $foo, where $ is the sigil. (wikipedia)

Scope:
Ruby(Perl): $GLOBAL_VAR, @instance_var, @@class_var

Data type:
Python: r"[regex]", u"unicode"
C#: @"\\server\share\file.txt"

Elixir: Sigil (Ruby)

```
defmacro sigil_w(term, modifiers) do
  ……
end
```

```
iex> ~w(--source test/enum_test.exs)
["--source", "test/enum_test.exs"]
iex> ~w(foo bar baz)a
[:foo, :bar, :baz]
```

```
iex> ~T[13:00:07] # times
~T[13:00:07]
```

```
iex> Regex.match?(~r(foo), "foo")
true
```

```
iex> Regex.match?(~r/abc/, "abc")
true
```

Haskell: OverloadedStrings?

```
{-# LANGUAGE OverloadedStrings #-}

a :: String

a = "hello"


b :: Text

b = "hello"
```

Template Haskell! Q(uasi)Q(uoter)

regexqq: [$rx|([aeiou]).*(er|ing|tion)([\.,\?]*)$|]

raw-strings-qq: [r|C:\Windows\SYSTEM|] ++ [r|\user32.dll|]

ruby-qq: [x|echo >&2 "Hello, world!"|]

aeason-qq: [aesonQQ| {age: 23, name: "John", likes: ["linux", "Haskell"]} |]

xml-html-qq: [html|<html></html>|] :: Document

Bonus: Record puns in Elixir

```
{-# LANGUAGE NamedFieldPuns #-}

greet IndividualR { person = PersonR { firstName = fn } } = "Hi, " ++ fn

greet IndividualR { person = Person { firstName } } = "Hi, " ++ firstName
```

```
foo = 1

bar = 2

~m(foo bar)a == %{foo: foo, bar: bar}
```

Protocol

Enumerable

Reducees

Protocol & Behaviour

Protocol(...like typeclass?)

- For data type
- Dispatching!

```
defprotocol Size do
  def size(data)
end

defimpl Size, for: Tuple do
  def size(tuple), do: tuple_size(tuple)
end
```

Behaviour(...like interface?):

- For Module
- Compile time

```
defmodule Parser do
  @callback parse(String.t) :: any
  @callback extensions() :: [String.t]
end

defmodule JSONParser do
  @behaviour Parser
  def parse(str), do: # ... parse JSON
  def extensions, do: ["json"]
end
```

Enumerable

Elixir provides the concept of collections, which may be in-memory data structures, as well as events, I/O resources and more. Those collections are supported by the Enumerable protocol, which is an implementation of an abstraction we call "**reducees**".

-- Introducing reducees

defprotocol **Enumerable** do

```
  @type acc :: {:cont, term} | {:halt, term} | {:suspend, term}

  @type reducer :: (term, term -> acc)

  @type result :: {:done, term} | {:halted, term} | {:suspended, term, continuation}

  @type continuation :: (acc -> result)

  @spec reduce(t, acc, reducer) :: result

  def reduce(enumerable, acc, fun)
   …
end
```

Iterator (ask)

```
def next([x|xs]) do
      {x, xs}
end

def next([]) do
   :done
end
```

```
def map(collection, fun) do
   map_next(next(collection), fun)
end

defp map_next({x, xs}, fun) do
   [fun.(x)|map_next(next(xs), fun)]
end

defp map_next(:done, _fun) do
   []
end
```

Iterator: Resource Management Problem(tell)

fs = File.stream(path)

Parse Error!

map parse [Line0, Line1, Line2, ...] #fs

Iterator: halt & try…catch…

```
defp map_next({h, t}, fun) do
  [try do
    fun.(h)
   rescue
     e ->
       halt(t)
       raise(e)
    end|map_next(next(t), fun)]
end
```

```
def take(collection, n) do
   take_next(next(collection), n)
end
...
defp take_next(:done, _n) do: []

defp take_next(value, 0) do
   halt(value)  # side-effect
end
```

Reducer (Clojure)

```
defmodule Reducer do
  def reduce([x|xs], acc, fun) do
    reduce(xs, fun.(x, acc), fun)
  end

  def reduce([], acc, _fun) do
    acc
  end
end
```

"the only thing that knows how to apply a function to a collection is the collection itself"

Reducer: Good & Bad

```
def reduce(file, acc, fun) do
 descriptor = File.open(file)

 try do
  reduce_next(IO.readline(descriptor), acc, fun)
 after
  File.close(descriptor)
 end
end
```

...

```
def take(collection, n) do

 # purely functional way?

end
```

Iteratee (Haskell)

```
defmodule Iteratee do
  def enumerate([h|t], {:cont, fun}) do
     enumerate(t, fun.({:some, h}))
   end

  def enumerate([], {:cont, fun}) do
    fun.(:done)
  end

  def enumerate(_, {:halt, acc}) do
    {:halted, acc}
  end
end
```

Iteratee & map

```
def map(collection, fun) do
 {:done, acc} = enumerate(collection, {:cont, mapper([], fun)})
 :lists.reverse(acc)
end

defp mapper(acc, fun) do
 fn
  {:some, h} -> {:cont, mapper([fun.(h)|acc], fun)}
  :done      -> {:done, acc}
 end
end
```

Iteratee & map

```
def map(collection, fun) do
 {:done, acc} = enumerate(collection, {:cont, mapper([], fun)})
 :lists.reverse(acc)
end

defp mapper(acc, fun) do
 fn
  {:some, h} -> {:cont, mapper([fun.(h)|acc], fun)}
  :done      -> {:done, acc}
 end
end
```

Reducees

```
defmodule Reducee do
def reduce([h|t], {:cont, acc}, fun) do
   reduce(t, fun.(h, acc), fun)
 end

 def reduce([], {:cont, acc}, _fun) do
  {:done, acc}
 end

 def reduce(_, {:halt, acc}, _fun) do
  {:halted, acc}
 end
end
```

Reducees & map

```
def map(collection, fun) do
  {:done, acc} =
    reduce(collection, {:cont, []}, fn x, acc ->
      {:cont, [fun.(x)|acc]}
    end)
 :lists.reverse(acc)
end
```

Reducees & take

```
def take(collection, n) when n > 0 do
  {_, {acc, _}} =
    reduce(collection, {:cont, {[], n}}, fn
      x, {acc, count} -> {take_instruction(count), {[x|acc], n-1}}
    end)
 :lists.reverse(acc)
end

defp take_instruction(1), do: :halt
defp take_instruction(n), do: :cont
```

Binary

Binary Pattern Matching

View Pattern

Char List(Haskell String) and Binary (Bytestring)

```
iex> i 'abc'
Term
  'abc'
Data type
  List

iex> 'hełło'
[104, 101, 322, 322, 111]
```

```
iex> i "abc"
Term
  "abc"
Data type
  BitString

iex> string = "hełło"
"hełło"

iex> byte_size(string)
7

iex> String.length(string)
5
```

Bitstring, Binaries & Strings

Bitstring: sequence of bits

V

Binary: sequence of bytes

V

String: UTF-8 encoded binary

<<args>> - Defines a new bitstring

iex> <<name::binary-size(5), " the ", species::binary>> = <<"Frank the Walrus">>
"Frank the Walrus"

iex> {name, species}
{"Frank", "Walrus"}

```
  def decode_response_header(<<0x81 :: size(8), opcode :: size(8),
                               key_length :: size(16), extras_length :: size(8),
                               _data_type :: size(8), status :: size(16),
                               total_body_length :: size(32), _opaque :: size(32),
                               cas :: size(64)>>) do
```

Ruby: String#unpack

irb(main):001:0> name, _ ,species = "Frank the Walrus".unpack 'a5a5a*'

=> ["Frank", " the ", "Walrus"]

irb(main):002:0> [name, species]

=> ["Frank", "Walrus"]

String Directive	Returns	Meaning
A	String	arbitrary binary string (remove trailing nulls and ASCII spaces)
a	String	arbitrary binary string

Binary Pattern Matching in Haskell?

parse function + Pattern Matching!

View Pattern

View patterns extend our ability to pattern match on variables by also allowing us to pattern match on the result of function application.

-- 24 Days of GHC Extensions: View Patterns

View Pattern (GHC Users Guide)

```
type Typ

data TypView = Unit
             | Arrow Typ Typ

view :: Typ -> TypView

size :: Typ -> Integer
size t = case view t of
  Unit -> 1
  Arrow t1 t2 -> size t1 + size t2
```

```
{-# LANGUAGE ViewPatterns #-}

size (view -> Unit) = 1
size (view -> Arrow t1 t2) = size t1 + size t2
```

Binary Pattern Matching in Haskell

Elixir:

```
<<name::binary-size(5), " the ", species::binary>> = <<"Frank the Walrus">>
```

Haskell:

```
splitAt :: Int -> ByteString -> (ByteString, ByteString)
stripSuffix :: ByteString -> ByteString -> Maybe ByteString
parse (sprintAt 5 -> (name, stripPrefix(" the ") -> Just species)) = (name, species)
```

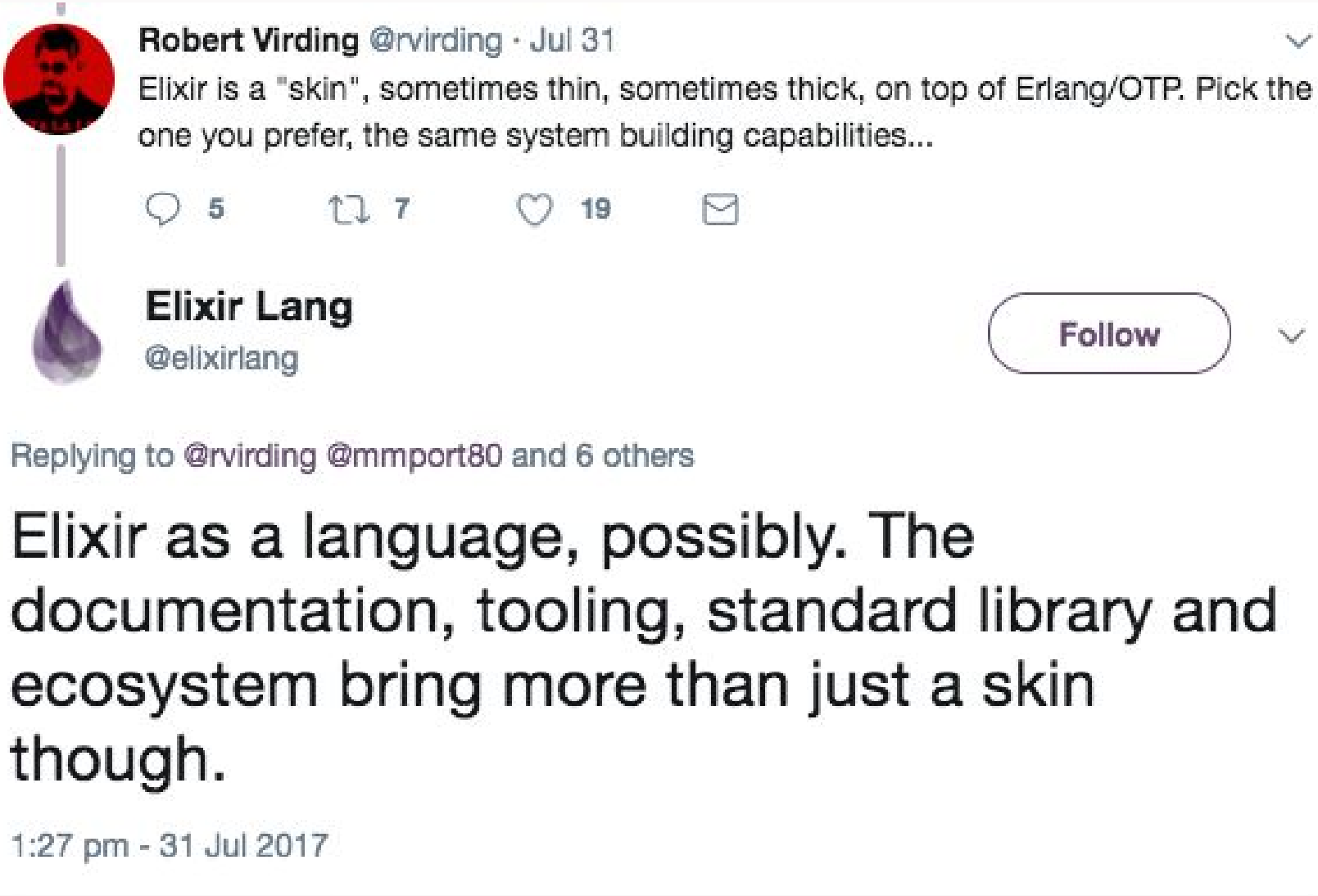
Summary

Haskell is a practical FP language :D

- Smaller community
- Non-strict evaluation
- Sometimes documentation is an academic paper
- Different terminology (Monoid, Functor, Monad, …)

-- What can python learn from Haskell?

Elixir is a practical FP language



Robert Virding @rvirding · Jul 31

Elixir is a "skin", sometimes thin, sometimes thick, on top of Erlang/OTP. Pick the one you prefer, the same system building capabilities...

5 7 19

Elixir Lang
@elixirlang

Follow

Replying to @rvirding @mmport80 and 6 others

Elixir as a language, possibly. The documentation, tooling, standard library and ecosystem bring more than just a skin though.

1:27 pm - 31 Jul 2017

Reference

- Computer Systems: A Programmer's Perspective(深入理解计算机系统)
- Learn X in Y minutes (Where X=elixir)
- Elixir: GETTING STARTED
- Null References: The Billion Dollar Mistake- Tony Hoare
- Purescript Erlang Backend
- Railway Oriented Programming (in Elixir , using with)
- Introducing reducees
- Programming Efficiently with Binaries and Bit Strings
- What Python can learn from Haskell