Introducing Elixir

Functional |> Concurrent |> Pragmatic |> Fun

@/+ pragdave
The future is functional
The future is concurrent
Moore's Law
Moore's Law
Microprocessor Transistor Counts 1971-2011 & Moore’s Law


my laptop
$10^9$ transistors

121 mm$^2$

8.26M transistors/mm$^2$
8.26M transistors/mm²
elixir

The future is **functional**

The future is **concurrent**
elixir

- Immutable data
- Pattern Matching
- Hygenic Macros
- blah blah...
Because all programming is transformation

- data
- code
elixir

transformation
Let’s Code
defmodule Dictionary do
  def signature_of(word) do
    word |> to_char_list |> Enum.sort |> to_string
  end
end
defmodule Dictionary do
  def signature_of(word) do
    word |> to_char_list |> Enum.sort |> to_string
  end
end

iex> Dictionary.signature_of "seat"
"aest"

iex> Dictionary.signature_of "teas"
"aest"
Transformation

- The |> operator makes transformation explicit
- OO focus: information hiding
- FP focus: information transformation
Concurrency

- **Elixir**
  - Behavio(u)rs simplify coding
  - OTP Wrappers
  - Abstractions (e.g., Tasks and Agents)

- **OTP**
  - servers/events/supervisors
  - management
  - hot code loading/recovery

- **Processes**
  - spawn/send/receive
  - synchronization
  - control protocols
Agents

Agent.start_link(fn -> initial_state end, name: SomeName)

Agent.update(SomeName, fn state -> new_state end)

Agent.get(SomeName, fn state -> return_value end)

All functions run in the agent’s process
Concurrency

Caller

Dictionary module

create
add_words
anagram_for

Agent

spawn
update
get

HashDict
defmodule Dictionary do
  
def signature_of(word) do
    word |-> to_char_list |-> Enum.sort |-> to_string
  end
  
def start_link do
    Agent.start_link(fn -> HashDict.new end, name: __MODULE__) end
  end
  
def add_some_words(words) do
    Agent.update(__MODULE__, &add_words(&1, words))
  end
  
def anagrams_of(word) do
    Agent.get(__MODULE__, fn dict -> Dict.get(dict, signature_of(word)) end)
  end
  
defp add_words(dict, words) do
    Enum.reduce words, dict, fn {signature, word}, dict ->
      Dict.update(dict, signature, [word], fn words -> [word|words] end)
    end
  end
end
Tasks

Task.async(fn -> computation end, name: SomeName)

Task.await(SomeName, timeout)
defmodule WordlistLoader do
  def load_from(file_names) do
    file_names
    |> Stream.map(fn name -> Task.async(fn -> load_task(name) end) end)
    |> Enum.map(&Task.await/1)
  end

  defp load_task(file_name) do
    file_name
    |> File.open!
    |> IO.stream(:line)
    |> IO.stream(:line)
    |> Stream.map(&String.strip/1)
    |> Enum.map(fn word -> {Dictionary.signature_of(word), word} end)
    |> Dictionary.add_some_words
  end
end

Dictionary.start_link

1..4

|> Enum.map(&"words/list#{&1}")
|> WordlistLoader.load_from_files

IO.inspect Dictionary.anagrams_of "crate"
Why Elixir?

It doesn’t have to be Elixir
Why Elixir?

But you do need to learn functional programming.
Why Elixir?

But you do need to learn functional programming, concurrent programming.
Why Elixir?

But you do need to learn functional programming, concurrent programming, and distributed programming.
So why not have fun at the same time?
Because STICKERS!
Programing Elixir

Dave Thomas

Foreword by
José Valim,
Creator of Elixir

edited by Lynn Belghley

Functional
> Concurrent
> Pragmatic
> Fun
Introducing Elixir
Functional |> Concurrent |> Pragmatic |> Fun

@/+ pragdave