Introducing Elixir

Alexei Sholik

August 3, 2013 @ kievfprog.net

What is Elixir?

What is Elixir?

Erlang VM

What is Elixir?

Less code

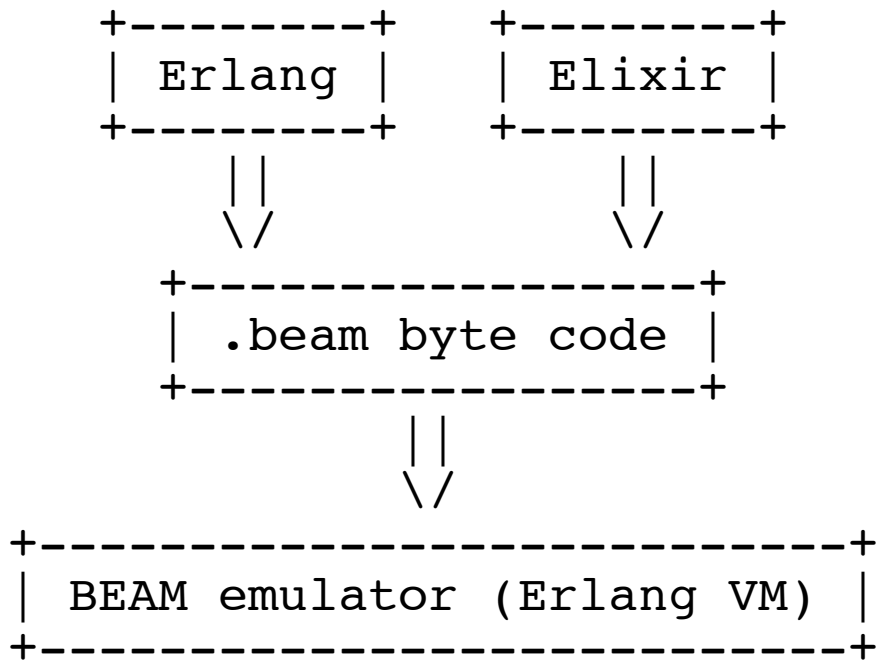
What is Elixir?

More fun

In a nutshell

- Immutable data
- Built-in concurrency
- Pattern matching
- Runtime polymorphism
- Metaprogramming

Erlang VM



Erlang VM

- Same byte code
- Same VM
- Same runtime environment
- Seamless interoperability
- OTP, Elixir on Xen, and more

Why Elixir?

- No repetitive boilerplate

```elixir
@doc "Bright (increased intensity) or Bold"
def bright() do
  "\e[1m"
end
defp escape_sequence(<<"bright", rest :: binary>>) do
  {"\e[1m", rest}
end

@doc "Sets alternative font 1"
def font_1() do
  "\e[11m"
end
defp escape_sequence(<<"font_1", rest :: binary>>) do
  {"\e[11m", rest}
end

@doc "Sets alternative font 2"
def font_2() do
  "\e[12m"
end
defp escape_sequence(<<"font_2", rest :: binary>>) do
  {"\e[12m", rest}
end

@doc "Sets alternative font 3"
def font_3() do
  "\e[13m"
end
defp escape_sequence(<<"font_3", rest :: binary>>) do
  {"\e[13m", rest}
end

@doc "Sets alternative font 4"
def font_4() do
  "\e[14m"
end
defp escape_sequence(<<"font_4", rest :: binary>>) do
  {"\e[14m", rest}
end

@doc "Sets alternative font 5"
def font_5() do
  "\e[15m"
end
defp escape_sequence(<<"font_5", rest :: binary>>) do
  {"\e[15m", rest}
end

@doc "Sets alternative font 6"
def font_6() do
  "\e[16m"
end
defp escape_sequence(<<"font_6", rest :: binary>>) do
  {"\e[16m", rest}
end

@doc "Sets alternative font 7"
def font_7() do
  "\e[17m"
end
defp escape_sequence(<<"font_7", rest :: binary>>) do
  {"\e[17m", rest}
end

@doc "Sets alternative font 8"
def font_8() do
  "\e[18m"
end
defp escape_sequence(<<"font_8", rest :: binary>>) do
  {"\e[18m", rest}
end

@doc "Sets alternative font 9"
def font_9() do
  "\e[19m"
end
defp escape_sequence(<<"font_9", rest :: binary>>) do
  {"\e[19m", rest}
end
```

vs

```elixir
defmacro defsequence(name, code // "", terminator // "m") do
  quote bind_quoted: [name: name, code: code, terminator: terminator] do
    def unquote(name)() do
      "\e[#{unquote(code)}#{unquote(terminator)}"
    end

    defp escape_sequence(<< unquote(atom_to_binary(name)), rest :: binary >>) do
      { "\e[#{unquote(code)}#{unquote(terminator)}", rest }
    end
  end
end

@doc "Bright (increased intensity) or Bold"
defsequence :bright, 1

lc font_n inlist [1, 2, 3, 4, 5, 6, 7, 8, 9] do
  @doc "Sets alternative font #{font_n}"
  defsequence :"font_#{font_n}", font_n + 10
end
```

- No repetitive boilerplate

```
@doc "Bright (increased intensity) or Bold"
defsequence :bright, 1

lc font_n inlist [1, 2, 3, 4, 5, 6, 7, 8, 9] do
  @doc "Sets alternative font #{font_n}"
  defsequence :"font_#{font_n}", font_n + 10
end
```

- Simple APIs

```
%% Erlang
lists:map(fun A -> ... end, List).
dict:map(fun K, V -> ... end, Dict).
gb_trees:map(fun K, V -> ... end, Tree).
lists:map(fun A -> ... end, sets:to_list(Set)).

## Elixir
Enum.map list, fn x -> ... end
Enum.map dict, fn {k, v} -> ... end
Enum.map set, fn x -> ... end
Enum.map 1..10, fn x -> ... end
Enum.map File.stream!("notes.txt"), fn x -> ... end
```

- More simple APIs

```
use ExUnit.Case

test "boolean" do
  assert !:atoms_are_truthy
end

test "comparison" do
  assert 1 == 2
end
```

- More simple APIs

```
1) test boolean (AssertTest)
          expected: !:atoms_are_truthy
             to be: true
       instead got: false

2) test comparison (AssertTest)
                   expected: 1
       to be equal to (==): 2
```

- DSLs

```
@prepare :authenticate_user
get "/users/:user_id" do
  # ...
end

defp authenticate_user(conn) do
  unless conn.session[:user_id] do
    halt! conn.status(401)
  end
end
```

- More DSLs

```elixir
import Ecto.Query

# A query that will fetch the ten first
# post titles
query = from p in Post,
      where: p.id <= 10,
     select: p.title

# Run the query against the database
titles = MyRepo.fetch(query)
```

Elixir

==

PRODUCTIVITY

Why Elixir?

- Practical (code reuse, meta)
- User friendly (syntax; exceptions; REPL)
- Tools (mix, ExUnit, EEx)
- New, consistent stdlib
- Enthusiastic community

Elixir

==

FUN

Demo time!

Demo

recap

Mix: your new project manager

```
defmodule Fprog.Mixfile do
  use Mix.Project

  def project do
    [app: :fprog, deps: deps, ...]
  end

  def application do
    [applications: [:httpotion]]
  end

  defp deps do
    [{ :httpotion, github: "myfreeweb/httpotion" },
     { :jazz, github: "meh/jazz" }]
  end
end
```

jazz: simple API

```
defprotocol JSON.Decoder do
  def from_json(data)
end

defprotocol JSON.Encoder do
  def to_json(self, options)
end

JSON.encode!(dict / list / record)
JSON.decode!(json)
```

jazz: extensible API

```
defimpl JSON.Encoder, for: anything do
  def to_json(...), do: ...
end

defimpl JSON.Decoder, for: anything do
  def from_json(...), do: ...
end

JSON.encode!(anything)
JSON.decode!(json, as: anything)
```

What is Elixir

good for?

What is Elixir good for?

Web apps

What is Elixir good for?

DSLs

...and more

- Distributed computing
- Data transformation
- Scripting
- Compilers
- [here could be your use case]

Everything Erlang can do, Elixir can do too

- Concurrency
- Robustness
- Low latency
- Scalability
- High availability

Who is using

Elixir?

Who is using Elixir?

- Genomu
- con_cache by Saša Jurić
- SoundCloud
- Five9, Inc
- ~100 people in `#elixir-lang`

Development status

- The project is still young
  - version 0.10
  - not too many pure Elixir libraries
  - missing advanced documentation
  - some tools are still WIP

Development status

- But it is quite stable
  - easy to start with
  - used in production
  - few user-level changes expected (so your code won't break)

Getting started

- Getting Started guide / elixir-lang.org
- *Programming Elixir* / pragprog.com
- *Introducing Elixir* / oreilly.com
- *Meet Elixir* / peepcode.com
- Wiki / github.com

Thank you!

Alexei Sholik

@true_droid

github.com/alco

elixir-lang.org | tryelixir.org

Random stuff

- Pipe operator
- Enum and Stream
- Pattern matching
- Docstrings
- Sigils
- Function currying
- REPL
- Nodes and binaries

Pipe operator

```
def process({user, project, count}) do
  Issues.GithubIssues.fetch(user, project)
    |>  decode_response
    |>  convert_to_list_of_hashdicts
    |>  sort_into_ascending_order
    |>  Enum.take(count)
    |>  print_table_for_columns(["number", "created_at", "title"])
end
```

From Programming Elixir by Dave Thomas

Enum and Stream

```elixir
alias Stream, as: S ;  alias Enum, as: E

1..10 |> S.map(to_binary &1) |> E.each(IO.puts &1)
#=> prints numbers from 1 to 10

S.cycle([1, 2, 3]) |> S.map(&1 * &1) |> E.take(10)
#=> [1, 4, 9, 1, 4, 9, 1, 4, 9, 1]

{0, 1}
|> S.iterate(fn {a, b} -> {b, a + b} end)
|> S.map(elem &1, 0)
|> E.take(10)
#=> [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Pattern matching (1)

```
a = 1  #=> 1
1 = a  #=> 1

[a | b] = [1, 2, 3]
# a == 1
# b == [2, 3]

{ :event, _ } = { :event, "kievfprog" }  # OK
{ :event, _ } = { :error, "no event" }
#=> ** (MatchError) no match of right hand side
#                   value: {:error, "no event"}
```

Pattern matching (2)

```
<< a::utf8, _::binary >> = "ßtring"
# a == 223
# <<a>> == <<223>>

{ <<a::utf8>>, "\x{df}" }  # 0xDF == 223
#=> {"ß", "ß"}

binary_to_list "ß"
#=> [195, 159]
```

Pattern matching (3)

```
defmodule M do
  def test([_|_]), do: "list"
  def test([]),    do: "empty list"

  def test({ _, _ }), do: "a pair"

  def test(1234),  do: "one two three four"
  def test(x) when is_number(x),
                   do: "some other number"

  def test(_),     do: "something else"
end
```

Docstrings

```
@moduledoc %B"""
## String and binary operations

The functions in this module act according to the
Unicode Standard, version 6.2.0.
"""

@doc """
Convert all characters in the string to uppercase.
"""
@spec upcase(t) :: t
defdelegate upcase(binary), to: String.Unicode
```

Sigils

```
# %b, %B -- strings
IO.puts %B"hello \"name\""
#=> hello \"name\"

IO.puts %b/hello "name"/
#=> hello "name"

# %r, %R -- regular expressions
"aaab" =~ %r"a+b"  #=> true

# in the shell
iex> h Kernel.sigil_r
```

Function currying

```
Enum.all? [:a, :b, :c], &is_atom(&1)
#=> true

Enum.map  ["a", "B", "c"], &String.upcase &1
#=> ["A", "B", "C"]

Enum.sort [1, 2, 3], &( &2 < &1 )
#=> [3, 2, 1]

fun = &(&1 + &2 + &3)
fun.(1, 2, 3)
#=> 6
```

REPL

- IEx.Helpers.<TAB>
- h()
- c()
- r()
- v()

Nodes and binaries

```
# Node.list
# Node.connect
# a = "Hello, b!"
# Node.spawn :"...", fn -> ... end
#    IO.puts a <> " My name is #{Node.self}"

# :code.load_binary M, 'file', code
```