Meta programming for Erlang

Yurii Rashkovskii,
Erlang Factory Lite Vancouver 2012

Wait, isn't Erlang already perfect?!

- Powers NoSQL databases, message exchanges, etc.
- Fault tolerant
- Functional
- Highly concurrent

Wait, isn't Erlang already perfect?!

- It is also so much **fun** to work with!
- So why people use other languages?

Why Joe prototyped **erl2**?

Well, may be there's something to think about?

```erlang
-module(server).
-behaviour(gen_server).

-export([start_link/0]).

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-define(SERVER, ?MODULE).

-record(state, {}).

start_link() ->
    gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).

init([]) ->
    {ok, #state{}}.

handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

This does... exactly nothing

```elixir
defmodule Server do
  use GenServer.Behavior
  import GenX.GenServer
  alias :gen_server, as: GenServer

  def start_link do
    GenServer.start_link
      {:local, __MODULE__}, __MODULE__,
      [], []
  end

  defrecord State, []

  def init(_), do: {:ok, State.new}
end
```

This does exactly the same (nothing)

```erlang
-module(server).
-behaviour(gen_server).

-export([start_link/0]).

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-define(SERVER, ?MODULE).

-record(state, {}).

start_link() ->
    gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).

init([]) ->
    {ok, #state{}}.
```

Adding calls

```erlang
handle_call({some_call, X},
            _From, State) ->
    {reply, ok, State}.
```

```erlang
handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

```erlang
-module(server).
-behaviour(gen_server).

-export([start_link/0]).

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-define(SERVER, ?MODULE).

-record(state, {}).
```

```erlang
some_call(X) ->
    gen_server:call(?SERVER,
                    {some_call, X}).
```

Adding calls

```erlang
start_link() ->
    gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).

init([]) ->
    {ok, #state{}}.

handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

```
-module(server).
-behaviour(gen_server).

-export([start_link/0]).

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-export([some_call/1]).

-define(SERVER, ?MODULE).

-record(state, {}).

some_call(X) ->
    gen_server:call(?SERVER,
             {some_call, X}.

start_link() ->
    gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).

init([]) ->
    {ok, #state{}}.

handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

Adding calls

```elixir
defmodule Server do
  use GenServer.Behavior
  import GenX.GenServer
  alias :gen_server, as: GenServer

  def start_link do
    GenServer.start_link
        {:local, __MODULE__}, __MODULE__,
        [], []
  end

  defrecord State, []

  def init(_), do: {:ok, State.new}

    defcall some_call(X), state: State do
      {:reply, :ok, State}
    end

end
```

Adding calls

How does Elixir achieve this?

Lisp-style macros.
Simple. Powerful.

What enables it?

Simple code representation structures

```
1 + 2 #=> {:+, <line>, [1,2]}
```

```
IO.puts "Hello, world!" #=>
{{:.,<line>,[{:__aliases__,<line>,
[:IO]},:puts]},<line>,["Hello,
world"]}
```

What enables it?

Quoting and unquoting

```
iex(1)> quote do: 1+2
#=> {:+,0,[1,2]}
```

```
iex(1)> a = quote do: 1+2
#=> {:+,0,[1,2]}
iex(2)> quote do: unquote(a)
#=> {:+,0,[1,2]}
```

```
iex(1)> a = quote do: ["Hello ~s~n",[ "world!"]]
#=> ["Hello ~s~n",["world!"]]
iex(2)> quote do: :io.format(unquote_splicing(a))
#=> {{:.,0,[:io,:format]},0,["Hello ~s~n",
["world!"]]}
```

What enables it?

Macros

```
defmacro unless(condition, opts) do
  quote do
    if !unquote(condition), unquote(opts)
  end
end

iex(1)> unless 2<1, do: IO.puts "Everything's okay"
Everything's okay
:ok
```

The idea is

Productivity

Monkey Patching

Kevin Smith:

“You’re sick. Seek help!”

jsx_to_json.erl

```
encode(string, String, _Opts) ->
    [?quote, String, ?quote];
encode(literal, Literal, _Opts) ->
    erlang:atom_to_list(Literal);
encode(integer, Integer, _Opts) ->
    erlang:integer_to_list(Integer);
encode(float, Float, _Opts) ->
    [Output] = io_lib:format("~p", [Float]), Output.
```

json.ex

```
defprotocol JSON.Encoder do
  def encode(data)
end

defimpl JSON.Encoder, for: String do
  def encode(data), do: ...
end

defimpl JSON.Encoder, for: Integer do
  def encode(data), do: ...
end
```

The idea is

Extensibility

First class shell

```
iex(1)> defmodule MyModule do
...(1)>   def hello_world do
...(1)>     IO.puts "Hello, world!"
...(1)>   end
...(1)> end
{:hello_world,0}
iex(2)> MyModule.hello_world
Hello, world!
:ok
```

Modules are programs

```
defmodule MyModule do
  a = true
  if a do
    IO.puts "Let's do it"
    def hello_world, do: IO.puts "Hello, world!"
  end
end
#=>
Let's do it
{:hello_world,0}
```

First class records

```
iex(1)> defrecord State, report_to: nil,
timeout: :infinity
nil
iex(2)> state = State.new report_to: self
State[report_to: <0.36.0>, timeout: :infinity]
iex(3)> state.report_to
<0.36.0>
iex(4)> State.report_to state
<0.36.0>
iex(5)> defrecord AnotherState, report_to: nil,
timeout: :infinity do
  def optimize(record), do: ...
end
nil
iex(6)> State[report_to: report_to] = state
State[report_to: <0.36.0>, timeout: :infinity]
iex(7)> report_to
<0.36.0>
```

The idea is

Making fucking sense

So, what is Elixir?

- Compiler to BEAM
- Meta programming tool
- A set of libraries
- *An ambitious project*

What else?

- Typespecs: check
- GenX: macro library for gen_*, application and supervisor
- rebar plugin: check
- v0.6.0 is being released right about now

Thanks!

Yurii Rashkovskii,
Erlang Factory Lite Vancouver 2012