Cómo conocí a la metaprogramación (y sobreviví a ello)

Miguel E. Ruiz

29 de febrero de 2024

Contenidos

1. Introducción
2. Metaprogramación
3. Mi TFM: `LogicElixir`

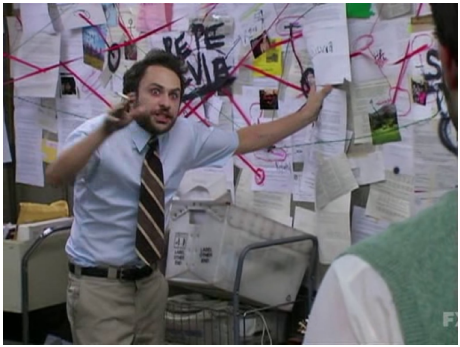
IO.puts("Hello, World!")

- `Miguel [Emilio]`
- Ingenierio Informático
- Intereses: Programación funcional y concurrente
- Erlang y Elixir

La metaprogramación y yo

- Al acabar el Grado (2018), interés en Elixir
  - Ejercicios en Codewars
  - Proyecto Pokedex
- Interés en el libro *Metaprogramming Elixir*
- Sin embargo...
  - Falta de tiempo (y de motivación)
- Arranca el máster (2021) y...
  - recupero el interés en la metaprogramación
  - posibilidad de hacer el TFM sobre ello

Una cosa os prometo...



No acabaremos así (o al menos, no mucho...)

Finalizadas las presentaciones...

Metaprogramación

¿KLK?

Elixir School

Metaprogramming is the process of using code to write code.

Metaprogramación

Definicion(es)

Elixir School

Metaprogramming is the process of using code to write code.

Rust Web Programming

Metaprogramming can generally be described as a way in which the program can manipulate itself based on certain instructions.

Metaprogramación

Definicion(es)

Elixir School

Metaprogramming is the process of using code to write code.

Rust Web Programming

Metaprogramming can generally be described as a way in which the program can manipulate itself based on certain instructions.

ChatGPT

Metaprogramming is a programming technique where a program can manipulate its own structure or behavior at runtime.

Metaprogramación

Técnicas

- Metaclases (Ruby, Python)
- Macros (Scala, Rust)
- Templates (C, C++)

Metaprogramación en Elixir

Mi definición

- Metaprogramación: mecanismo para generar programas a partir de otros programas (en tiempo de compilación).
- Elixir tiene la capacidad de tratar los programas como un tipo de dato y modificarlos o transformarlos.
- Ejemplos de uso:
  - Extensión del lenguaje a través de DSL.
  - Creación de nuevas librerías como *Phoenix* y *Ecto*.

Metaprogramación en Elixir

Ejemplos

```elixir
defmodule Sample.Weather do
  use Ecto.Schema
  schema "weather" do
    field :city     # String by default
    field :temp_lo, :integer
    field :temp_hi, :integer
    field :prcp,    :float, default: 0.0
  end
end
```

```elixir
defmodule MyAppWeb.Router do
  use Phoenix.Router

  get "/login", LoginController, :show

  scope "/" do
    pipe_through [:browser]
    get "/posts", PostController, :index
    get "/posts/:id", PostController, :show
  end
end
```

Metaprogramación en Elixir

Ejemplos

```elixir
defmodule Calculator do
  use ExActor.GenServer

  defstart start_link, do: initial_state(0)

  defcast inc(x), state: state, do: new_state(state + x)
  defcast dec(x), state: state, do: new_state(state - x)

  defcall get, state: state, do: reply(state)

  defcast stop, do: stop_server(:normal)
end
```

Metaprogramación en Elixir

¿Cómo?

- A través de macros, las cuales permiten **extender** el lenguaje

¿Cuántas palabras reservadas hay en Elixir?

- 8
- 15
- 37
- ninguna

Metaprogramación en Elixir

¿Cómo?

- A través de macros, las cuales permiten **expandir** el lenguaje

¿Cuántas palabras reservadas hay en Elixir?[1]

- 8
- 15
- 37
- ninguna

```
true, false, nil, when, and, or, not, in
fn, do, end, catch, rescue, after, else
```

[1] https://hexdocs.pm/elixir/1.16/syntax-reference.html#reserved-words

AST (de Elixir)

- Abstract Syntax Tree
- La estructura interna del código Elixir
- Se representa mediante una tupla de 3 elementos:
  - Un átomo que representa la función invocada u otra tupla que representa un nodo interno del árbol.
  - Una lista de términos de metadatos de la función invocada.
  - Una lista de argumentos de la función del primer argumento.
- El tipo `Macro.t()` representa el AST de Elixir

AST de Elixir

Para obtener el AST de una expresión de Elixir, se utiliza la función `quote/2`:

```
iex> quote do: sum(1, 2, 3)
{:sum, [], [1, 2, 3]}
```

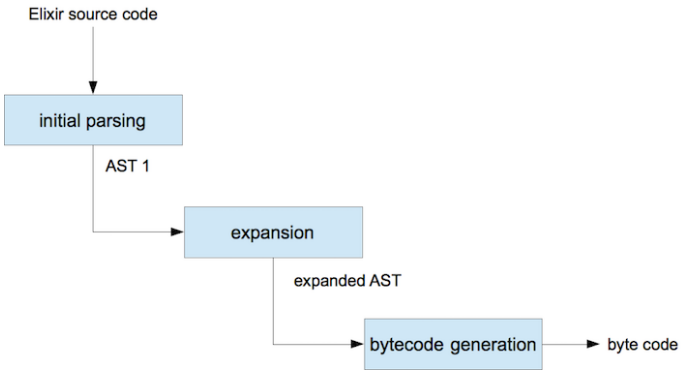
Si se requiere introducir valores dentro de un AST, se dispone de la función `unquote/1`:

```
iex> a = 3
iex> quote do: sum(1, 2, a)
{:sum, [], [1, 2, {:a, [], Elixir}]}
iex> quote do: sum(1, 2, unquote(a))
{:sum, [], [1, 2, 3]}
```

DEMO
(Desmembrando ASTs)

Metaprogramación en Elixir

Proceso de compilación



- Con el operador `defmacro` podremos definir nuestras macros, respetando las reglas[2]:
  - Don't Write macros (WTF?)
  - Use Macros Gratuitously
- El operador **tiene que** devolver un AST

[2] Metaprogramming Elixir: Macro Rules

Definición de macros

Ejemplo básico

```elixir
defmodule ControlFlow do
  defmacro unless(expression, do: block) do
    quote do
      if !unquote(expression), do: unquote(block)
    end
  end
end
```

DEMO

Un DSL para definir mascotas

```elixir
defmodule Template do
  use Pets

  pet "Bucky" do
    species :dog
    hobbies ["sniffing", "eating birds"]
  end
  pet "Gardfield" do
    species :cat
    hobbies ["hating mondays"]
  end
end
```

```
iex> Bucky.greet
"Hello my name is Bucky and I'm a dog"
iex> Gardfield.hobbies
"Gardfield's hobbie is hating mondays"
```

Antipatrones

Sobreutilizar esta técnica podría llevarnos a algunos de estos casos[3]:

- *Large code generation*: Cuando la macro genera demasiado código
- *Unnecessary macros*: ¿Estás seguro que este código debe ser una macro?
- `use` instead of `import`: Propagación de dependencias

[3] `https://hexdocs.pm/elixir/macro-anti-patterns.html` (`>=1.16`)

Mi TFM: LogicElixir

Motivación

- La mayor de todas: aprender sobre metaprogramación (en Elixir).
- Hacer mi primero proyecto (serio) en Elixir.
- Acabar el Máster :D

- Desarrollar una librería para Elixir que permita a un programador especificar un programa lógico
- Proporcionar mecanismos para realizar consultas sobre ese programa lógico

LogicElixir: Prerrequisitos

Programación Lógica

```
% Hechos
human(fry).
mutant(leela).
likes(fry, pizza).

% Reglas
pizza_lover(X):- likes(X, pizza).

member(X, [X | _]).
member(X, [_ | T]):- member(X, T).
```

LogicElixir

Ejemplo

```
defmodule LogicModule do
  use LogicElixir
  defpred human(:fry)
  defpred mutant(:leela)
  defpred likes(:fry, :pizza)
  defpred pizza_lover(X), do: likes(X, :pizza)
  defpred member(X, [X | T])
  defpred member(X, [H | T]) do
    member(X, T)
  end
end
```

Términos lógicos

T	$::=$	*lit*	*literales*
	$\mid$	X	*variables lógicas del conjunto* `Var`
	$\mid$	`[]`	
	$\mid$	`[`T_1 `|` T_2`]`	
	$\mid$	`{`T_1`,` T_2`,` `...,` T_n`}`	
	$\mid$	f`(`T_1`,` T_2`,` `...,` T_n`)`	*llamadas a funciones de Elixir*

Patrones

$$
\begin{array}{rcll}
P & ::= & lit & \\
 & | & X & \textit{variables lógicas} \\
 & | & [\,] & \\
 & | & [P_1 \mid P_2] & \\
 & | & \{P_1,\ P_2,\ \ldots,\ P_n\} &
\end{array}
$$

Objetivos y secuencias de objetivos

G	$::=$	ϵ	
	$\mid$	g, G	
g	$::=$	$p(T_1, T_2, \ldots, T_n)$	*predicados lógicos*
	$\mid$	$T_1 = T_2$	*unificación de términos*
	$\mid$	$G_1 ; \ldots ; G_n$	*disyunción de secuencias*
	$\mid$	$@(e)$	*expresiones de Elixir*

Reglas

$$R ::= p(P_1, ..., P_n)\,[:-G]$$

Definiciones de predicados

$$D = \begin{bmatrix} p(P_{1,1},\ \ldots,\ P_{1,n}) & [:- & G_1] \\ p(P_{2,1},\ \ldots,\ P_{2,n}) & [:- & G_2] \\ \vdots & & \\ p(P_{m,1},\ \ldots,\ P_{m,n}) & [:- & G_m] \end{bmatrix}$$

Lenguaje Core

- El lenguaje Core es el lenguaje de bajo nivel que servirá de nexo entre `LogicElixir` y Elixir.
- Una función Core agrupa todos los hechos y reglas de un predicado `defpred` en una única función `defcore`.
- Posee una sintaxis parecida a `LogicElixir`, pero con matices:
  - No existen los patrones.
  - Ausencia de sintaxis azucarada en las listas.
  - Los predicados tienen que tener un cuerpo de función.
- Los términos de este lenguaje estarán contenidos dentro del conjunto `Term`.

Traducción de LogicElixir a Core

Se considera una definición de predicado D:

$$D = \left[\begin{array}{ccc} p(P_{1,1}, \ldots, P_{1,n}) & :- & G_1 \\ p(P_{2,1}, \ldots, P_{2,n}) & :- & G_2 \\ \vdots & & \\ p(P_{m,1}, \ldots, P_{m,n}) & :- & G_m \end{array}\right]$$

Traducción de LogicElixir a Core

$$
\mathit{generate\_defcore}(D) = \left[\begin{array}{l}
\texttt{defcore } p(X_1, \ldots, X_n) \texttt{ do} \\
\quad \texttt{choice do} \\
\quad\quad P_{1,1} = X_1 \\
\quad\quad \ldots \\
\quad\quad P_{1,n} = X_n \\
\quad\quad G_1 \\
\quad \texttt{else} \\
\quad \ldots \\
\quad \texttt{else} \\
\quad\quad P_{m,1} = X_1 \\
\quad\quad \ldots \\
\quad\quad P_{m,n} = X_n \\
\quad\quad G_m \\
\quad \texttt{end} \\
\texttt{end}
\end{array}\right]
$$

Traducción de LogicElixir a Core

Ejemplos

```
defpred planet(:tatooine)
defpred planet(:hoth)
defpred planet(:dagobah)
```

```
defcore planet(X1) do
  choice do
    :tatooine = X1
  else
    :hoth = X1
  else
    :dagobah = X1
  end
end
```

Traducción de LogicElixir a Core

Ejemplos

```
defpred member(X, [X | T])
defpred member(X, [H | T]) do
  member(X, T)
end
```

```
defcore member(X1, X2) do
  choice do
    X = X1
    [X | T] = X2
  else
    X = X1
    [H | T] = X2
    member(X, T)
  end
end
```

Implementación

logic_elixir.ex

¿Qué supone la expresión `use LogicElixir`?

Se inyecta un AST a través de la función `__using__/1`:

```
defmodule LogicElixir do
  defmacro __using__(_params) do
    quote do
      use LogicElixir.Defpred
      use LogicElixir.Findall
    end
  end
end
```

Implementación

logic_elixir.ex

¿Qué supone la expresión use LogicElixir?

Se inyecta un AST a través de la función __using__/1:

```elixir
defmodule LogicElixir do
  defmacro __using__(_params) do
    quote do
      use LogicElixir.Defpred # <= Dependency injection
      use LogicElixir.Findall
    end
  end
end
```

Implementación

defpred.ex

```elixir
defmodule LogicElixir.Defpred do
  defmacro defpred(head) do
    {pred_name, args_ast} = Macro.decompose_call(head)
    Module.put_attribute(__CALLER__.module, :definitions,
                         {pred_name, args_ast})
  end

  defmacro defpred(head, do: block) do
    {pred_name, args_ast} = Macro.decompose_call(head)
    Module.put_attribute(__CALLER__.module, :definitions,
                         {pred_name, {args_ast, block}})
  end
end
```

Implementación

defpred.ex

```
defmacro __before_compile__(env) do
  VarBuilder.start_link()
  definitions = Module.get_attribute(env.module, :definitions)
                |>
                Enum.reverse()
  grouped_defs = definitions |>
                 Enum.group_by(&elem(&1, 0), &elem(&1, 1))
  for {name, args} <- grouped_defs do
    generate_defcore(name, args)
  end
end
```

Traducción de Core a función de Elixir

- El paso final es convertir la definición Core intermedia a código de Elixir.
- Para ello, es necesario definir una serie de conjuntos, así como una gramática BNF de estos términos disponibles en tiempo de ejecución.

Traducción de Core a función de Elixir

Definición de conjuntos

Además de `Var` y `Term`, se consideran los conjuntos:

- `EVar`, el conjunto de variables de Elixir.
- `Subst`, el conjunto de sustituciones.
- `Expr`, el conjunto de expresiones de Elixir.
- `Eval`, el conjunto de valores de Elixir disponibles en tiempo de ejecución.
- `RTerm`, el conjunto de términos disponibles en tiempo de ejecución.

Traducción de Core a función de Elixir

Términos *ground*

- Para determinar en tiempo $O(1)$ si un término es *ground* o no, utilizamos tuplas `{:ground, _}` en nuestra representación de los elementos `RTerm`.
- Para ello, han de cumplirse los siguientes invariantes:
  - Si está representado de la forma `{:ground, R}`, entonces no contiene variables.
  - Si no está representado de la forma `{:ground, R}`, entonces ha de contener alguna variable.

Traducción de Core a función de Elixir

Estructura de datos

Se manejarán algunas estructuras de datos para facilitar la traducción:

- Δ: Relaciona los términos del conjunto `Var` con los términos del conjunto `EVar`.
- θ: Relaciona los términos del conjunto `Var` con los términos del conjunto `RTerm`.

$$\Delta = [X_1 \to x_1, X_2 \to x_2, \ldots, X_n \to x_n]$$
$$\theta = [X_1 \to R_1,\ X_2 \to R_2,\ \ldots,\ X_n \to R_n]$$

Traducción de Core a función de Elixir

Funciones auxiliares

Se definen funciones auxiliares para facilitar la traducción y la reducción de términos:

- `build_list/2`
- `build_tuple/1`
- `groundify/2`

Traducción de Core a función de Elixir

Términos

Los términos Core serán traducidos por la función `tr_term/3`:

$$
\begin{aligned}
tr\_term(\Delta, x, lit) &= \{\texttt{:ground},\ lit\} \\
tr\_term(\Delta, x, X) &= \{\texttt{:var},\ \Delta(X)\} \\
tr\_term(\Delta, x, [T_1 \mid T_2]) &= \texttt{build\_list}(tr\_term(\Delta, x, T_1),\ tr\_term(\Delta, x, T_2)) \\
tr\_term(\Delta, x, \{T_1, ..., T_n\}) &= \texttt{build\_tuple}([tr\_term(\Delta, x, T_1),\ \ldots,\ tr\_term(\Delta, x, T_n)]) \\
tr\_term(\Delta, x, f(T_1, ..., T_n)) &= \left[\begin{array}{l} x_1 = \texttt{groundify}(x,\ tr\_term(\Delta, x, T_1)) \\ \vdots \\ x_n = \texttt{groundify}(x,\ tr\_term(\Delta, x, T_n)) \\ \{\texttt{:ground},\ f(x_1,\ \ldots,\ x_n)\} \end{array}\right]
\end{aligned}
$$

Traducción de Core a función de Elixir

Objetivos y secuencias de objetivos (I)

$$tr\_goals(\Delta,\ \epsilon) = \texttt{fn}\ th \rightarrow [th]\ \texttt{end}$$

$$tr\_goals(\Delta,\ (g,\ G)) = \left[\begin{array}{l} \texttt{fn}\ th_1 \rightarrow \\ \quad (tr\_goal(\Delta, g)).(th_1) \\ \quad \texttt{|>}\ \texttt{Stream.flat\_map}(\texttt{fn}\ th_2 \rightarrow \\ \qquad (tr\_goals(\Delta, G)).(th_2) \\ \quad \texttt{end}) \\ \texttt{end} \end{array}\right]$$

Traducción de Core a función de Elixir

Objetivos y secuencias de objetivos (II)

$$tr\_goal(\Delta, p(T_1, \ldots, T_n)) = \left[\begin{array}{l} \texttt{fn } th \rightarrow \\ \quad p(tr\_term(\Delta, th, T_1), \ldots, tr\_term(\Delta, th, T_n)).(th) \\ \texttt{end} \end{array}\right]$$

$$tr\_goal(\Delta, T_1 = T_2) = \left[\begin{array}{l} \texttt{fn } th \rightarrow \\ \quad \texttt{unify\_gen}(th, tr\_term(\Delta, th, T_1), tr\_term(\Delta, th, T_2)) \\ \texttt{end} \end{array}\right]$$

$$tr\_goal(\Delta, G_1; \ldots; G_n) = \left[\begin{array}{l} \texttt{fn } th \rightarrow [tr\_goals(\Delta, G_1), \ldots, tr\_goals(\Delta, G_n)] \\ \quad \texttt{|> Stream.flat\_map(fn f} \rightarrow \texttt{f.}(th) \texttt{ end)} \\ \texttt{end} \end{array}\right]$$

$$tr\_goal(\Delta, @(T)) = \left[\begin{array}{l} \texttt{fn } th \rightarrow \\ \quad \texttt{check\_b}(th, \texttt{ groundify}(th, tr\_term(\Delta, th, T)) \\ \texttt{end} \end{array}\right]$$

Traducción de Core a función de Elixir

Predicado Core a función Elixir

$$tr_def(p(X_1, \ldots, X_n) \text{:} - G) = \left[\begin{array}{l}
\texttt{def } p(t_1, \ldots, t_n) \texttt{ do} \\
\quad x_1 = \texttt{VarBuilder.gen_var()} \\
\quad \vdots \\
\quad x_n = \texttt{VarBuilder.gen_var()} \\
\quad y_1 = \texttt{VarBuilder.gen_var()} \\
\quad \vdots \\
\quad y_m = \texttt{VarBuilder.gen_var()} \\
\quad \texttt{fn } th_1 \rightarrow \\
\quad\quad th_2 = \texttt{Map.merge}(th_1, \\
\quad\quad\quad\quad \texttt{Map.new}([\{x_1, t_1\}, \ldots, \{x_n, t_n\}])) \\
\quad\quad (tr_goals(\Delta, G)).(th_2) \\
\quad\quad\quad\quad \texttt{|>} \\
\quad\quad\quad\quad \texttt{Stream.map}(\&\texttt{Map.drop}(\&1, [\\
\quad\quad\quad\quad\quad\quad x_1, \ldots, x_n, y_1, \ldots, y_m])) \\
\quad \texttt{end} \\
\texttt{end}
\end{array}\right]$$

Consulta de predicados

- `LogicElixir` ofrece un mecanismo de consulta sobre los predicados que haya declarado el usuario en su proyecto.
- La macro `findall` sirve de interfaz entre `LogicElixir` y Elixir.

$$F ::= \texttt{findall}\ T\ [\texttt{into}\ E\ ]\ \texttt{do}\ G$$

Donde:

- T es un término `LogicElixir`.
- G es una secuencia de objetivos `LogicElixir`.
- E es un término opcional que implementa el protocolo `Enumerable`.

Consulta de predicados

Traducción

$$
tr\_findall(T, G, E \setminus\setminus \texttt{nil}) = \left[\begin{array}{l}
x_1 = \texttt{VarBuilder.gen\_var()} \\
\ldots \\
x_n = \texttt{VarBuilder.gen\_var()} \\
solutions = (tr\_goals(\Delta, G)).(\texttt{\%\{\}}) \\
\quad \texttt{|> Stream.map(fn } sol \rightarrow \\
\qquad \texttt{t =} \ tr\_term(\Delta, sol, T) \\
\qquad \texttt{groundify(}sol, \texttt{t)} \\
\quad \texttt{end)} \\
\texttt{case } E \texttt{ do} \\
\quad \texttt{nil} \ \rightarrow solutions \\
\quad \texttt{Z} \ \rightarrow solutions \ \texttt{|> Enum.into(Z)} \\
\texttt{end}
\end{array}\right]
$$

DEMO
(Probando mi TFM)

Dificultades encontradas

- Falta de (buena) documentación
- Falta de código de apoyo (o desactualizado)
- Contramedidas: test unitarios

Conclusiones

- La metaprogramación es una área bonita (y MUY compleja)
- Seguramente nunca tengas que usar esta técnica
- No está de más conocer qué hace la “magia”
- Los tests son tus amigos

References

- Chris McCord (2015)
  Metaprogramming Elixir
- José Valim
  Kernel.SpecialForms, Macro, Module
- Leon Sterling and Ehud Shapiro (1986)
  The Art of Prolog
- https://github.com/MiguelERuiz/logic_elixir
- https://github.com/MiguelERuiz/logic_elixir_examples