Métaprogrammation en Elixir

Une introduction naïve

Xavier Van de Woestyne
xaviervdw@gmail.com - *margo.com*

Paris.ex 12 - Mai 2019

Bonjour/Bonsoir !

Xavier Van de Woestyne

- Bruxelles, Lille, Paris
- *Data Engineer* chez Margo Bank

Réseaux

- https://xvw.github.io
- *vdwxv* sur Twitter
- *xvw* sur Github/Gitlab
- *xvw@merveilles.town* sur Mastodon

OCaml, F#, Erlang/Elixir, Kotlin, Io, Ruby, Elm, Racket

LilleFP

- Meetup régulier
- Langages applicatifs
- Programmation fonctionnelle
- Systèmes de types
- Fusion de LilleFP, Lille Elixir, Lille Rust et LilleSUG
- **On recrute des speakers !**

Après presque 10 ans de Erlang

J'appartiens aux gens qui n'ont pas été particulièrement emballé par Elixir

- Je préférais la syntaxe de **Erlang** (amoureux de **Prolog**) ;
- j'étais inquiet de la transmissions d'idiomes (Ruby) ;
- donc je trouvais que Elixir ne servait à rien (à tort…).

Ce qui m'a fait changé d'avis sur Elixir

- **Phoenix** (plus spécifiquement, **Ecto** et **Plug**) ;
- l'unification des types pour les chaines de caractères ;
- les mécanismes de méta-programmation ;
- **mix** ;
- et plus tard : `gen_stage` et `Flow`.

Sommaire et objectifs

- Comprendre ce qu'est *rapidement* la méta-programmation ;
- survoler les perspectives qu'Elixir offre pour *méta-programmer* ;
- observer quelques cas d'usages aux **macros** ;
- conclure sur les usages et avantages.

  Ceci est introduction naïve … pour les non-initiés qui, comme le temps est mon ennemi, n'évoquera malheureusement pas tout ce qu'il est possible de dire sur la méta-programmation en Elixir :'(

- Ceci n'est pas du tout une présentation technique ! (Désolé…)

Meta-programmation

L'écriture de programme qui manipulent des données ***décrivant*** *elles-mêmes des programmes. (d'où le "meta")*

Meta-programmation

On peut méta-programmer de plusieurs manières

- Via la réflexion (implicite ou explicite) ;
- via des raccourcis du langage ;
- via des générateurs de code (par exemple les affreux templates de C++) ;
- via des étapes de compilation (multi-staged metaprogramming) ;
- via des macros ;
- en utilisant un Lisp (aha !)

A quoi ça sert ?

- Limiter le **boilerplate** ;
- limiter le **boilerplate** ;
- limiter le **boilerplate** ;
- étendre le langage (eDSL) ;
- déléguer à une machine des tâches rémanantes.

Elixir offre plusieurs outils de méta-programmation !

Introspection via `Module.__info__(subject)`

```
iex> Map.__info__(:functions)
[delete: 2, drop: 2, ... values: 1]
```

Dispatch dynamique

```
def call(module) do
  apply(module, function, [args])
end
```

Génération de code et dérivation d'interfaces

- **Génération**: Behaviour
- **Dérivation**: Protocol

Ce sont les manières rapides (et un peu cheap*) de faire de la meta-programmation.*

Pré-processeur et macros

On entre enfin dans le vif du sujet !

Une manière de **transformer** une unité exécutable **avant** son exécution

Pré-processeur et macros

Par exemple

- `#include <stdlib.h>` et `#define AGE 29`
- Les préprocesseurs CSS (Less, Sass, etc.)
- Les postprocesseurs CSS (aha)
- `cat myProgram.ex | sed ... | awk ... > myNewProgram.ex`

Cependant, tout le monde n'agit pas comme des sauvages

- **Lisp** introduit la capacité de manipuler des **quasi-quotations**
- elles permettent de manipuler des termes du langages … comme des termes du langages
- c'est en partie possible parce que la grammaire du langage est très simple
- le langage peut être étendu (parfois … de manière suréaliste, **Racket**).

Processus de compilation d'un programme Elixir

La grammaire d'Elixir étant plus complexe, on manipule son **AST**.

Figure 1: Processus de compilation

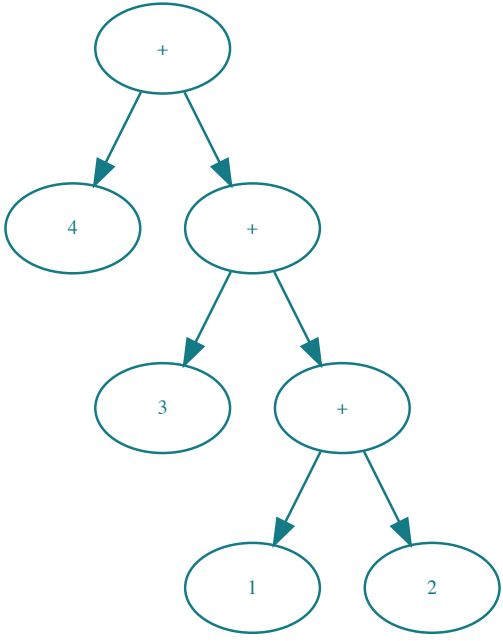
- Il sera possible d'injecter des fragments d'AST avant la phase **d'expansion**
- ces fragments sont des **macros**

Les macros (enfin !)

- On presque la même sémantique qu'une fonction
- concrètement, c'est une fonction qui prend des arguments et qui renvoie un fragment d'AST
- résolue avant la phase **d'expansion**

A quoi ressemble un fragment d'AST ?

```
1 + 2 + 3 + 4
```


+
4
+
3
+
1
2

```
{:+, [context: Elixir, import: Kernel],
 [
   {:+, [context: Elixir, import: Kernel],
    [{:+, [context: Elixir, import: Kernel], [1, 2]}, 3]},
   4
 ]}
```

C’est un peu ennuyant à écrire à la main…

Quotations

Heureusement, Elixir permet, comme Lisp, de manipuler des quotations via la macro `quote/1`

```
iex> quote do: if(true, do: "foo", else: "bar")

{:if, [
   context: Elixir, import: Kernel], [
     true, [do: "foo", else: "bar"]]}
```

- Les macros permettent d'injecter, à la compilation, un fragment d'AST **syntaxiquement valide**
- Les quotations permettent de simplifier l'écriture de fragments d'AST
- On peut **dé-quoter** (via `unquote/1`) des identifiant (qui seront injectés en fonction du contexte)
- Utilisées par beaucoup de bibliothèques majeures (Ecto, Plug, ExUnit etc.) pour étendre le langage (via des DSL's)
- Utilisées pour décrire 80% des expressions du langage

Un premier exemple : implémentons `unless`

Partiellement volé à la documentation d'Elixir

```
if(x) == unless(x)
```

Une première approche au moyen de fonctions

```
def my_unless(predicate, do: expression) do
  if(!predicate, do: expression)
end


iex(2)> my_unless false, do: IO.puts "foo"
foo # Ça semble fonctionner
:ok

iex(1)> my_unless true, do: IO.puts "foo"
foo # Damn, le "foo" s'affiche tout de même !
nil
```

Utilisation de la paresse pour corriger l'approche fonctionnelle

```elixir
def my_unless(predicate, do: expression) do
  if(!predicate, do: expression.())
end
```

```elixir
iex(1)> my_unless false, do: fn() -> IO.puts "foo" end
foo
:ok

iex(2)> my_unless true, do: fn () ->IO.puts "foo" end
nil # Victoire !
```

Mais c'est … un peu laid… on est loin d'un DSL.

Utilisation de macro

```elixir
defmacro my_unless(predicate, do: expression) do
  quote do
    if(! unquote(predicate), do: unquote(expression))
  end
end
```

```
iex(9)> require Sample
iex(10)> Sample.my_unless false, do: IO.puts "foo"
foo
:ok

iex(11)> Sample.my_unless true, do: IO.puts "foo"
nil # Victoire Réelle !
```

Un autre exemple : overloading

```
defmodule Sample do
  def x + y, do: [x, y]
  def x - y, do: [y, x]
end

test "Test for +" do
  import Sample
  assert (1 + 2) == [1, 2]
  assert (1 - 2) == [2, 1]
end
```

```
== Compilation error in file test/sample_test.exs ==
** (CompileError) test/sample_test.exs:8: function +/2 imported from both
                 Sample and Kernel, call is ambiguous
    (ex_unit) expanding macro: ExUnit.Assertions.assert/1
    test/sample_test.exs:8: SampleTest."test Test for +"/1
    (elixir) lib/code.ex:767: Code.require_file/2
    (elixir) lib/kernel/parallel_compiler.ex:211: anonymous fn/4 in
             Kernel.ParallelCompiler.spawn_workers/6
```

Solution

```
test "Test for +" do
  import Kernel, except: [+: 2, -: 2]
  import Sample
  assert (1 + 2) == [1, 2]
  assert (1 - 2) == [2, 1]
end
```

Utilisons nos super-pouvoirs !

```elixir
defmacro overload(methods, from: a, with: b) do
    quote do
      import unquote(a), except: unquote(methods)
      import unquote(b), only: unquote(methods)
    end
end


test "simple overloading" do
    import Scope
    overload [+: 2, -: 2], from: Kernel, with: Test

    assert [2, [1, 3]] == (1 + 3 - 2)
end
```

Et … c'est une très mauvaise idée

- Altération de la syntaxe pour de la **featurite**
- Complexifie la compréhension du code pour une futilité (et ce n'est pas cool pour les co-workers)
- L'utilisation de macros implique un raisonnement en amont !

Gestion complexe d'unités de mesures

On termine sur un exemple plus complexe !

- Pour un projet : manipulation de distances (m/cm/km) et de durée (sec/min/h/d)
- Rappel du projet **Mars Climate Orbiter** (+ de 50 millions partit en fumée)
- Très difficile de tester unitairement l'appartenance à un système métrique
- Tentative d'approche sans macro

Simulation de types fantômes (sans les garanties, mais testables)

```
def cm(x), do: {:cm, x}

def add({base, x}, {base, y}), do: {base, x + y}
```

Enrichissement du système

```
def cm(x), do: {:distance, :cm, x, 100.0}

def m(x), do: {:distance, :m, x, 1.0}

def add({base, ref, x, coeff1}, {base, _,  y, coeff2}) do
   {base, ref, x + (y * (coeff1 / coeff2)), coeff1}
end
```

Achievement unlocked ... sans macro !!!!

(celsius * 1.8) + 32.0

... lol, merci les farenheit's !

Ici typiquement… les macros peuvent servir

```
defmodule Distance do
  use Mizur.System
  type m
  type cm = m / 100
  type mm = m / 1000
  type km = m * 1000
end
```

```
defmodule Time do
  use Mizur.System
  type sec
  type min  = sec * 60
  type hour = sec * 60 * 60
  type day = sec * 60 * (60 * 24)
end
```

```
defmodule Temperature do
  use Mizur.System, intensive: true
  type celsius
  type farenheit = (celsius * 1.8) + 32.0
end
```

- Génération d'une API *typesafe* pour des systèmes métriques
- grâce aux macros, l'expression arithmétique qui défini un type est inversée pour offrir une conversion bi-directionnelle sans coût arithmétique complémentaire.

Concrètement ?

- La méta-programmation est très facile à utiliser
- Le mécanisme de macro est un outil puissant et robuste (et facile à utiliser)
- Intensivement utilisé dans Elixir (pour le langage, Ecto, Phoenix, Plug etc.)
- On peut être tenter d'en mettre partout… et c'est balot car :
  - ça peut complexifier la lecture du code
  - le débogage

Quand s'en servir ?

- Pour évincer du *boilerplate* très lourd (par exemple, une requête SQL via des compréhensions)
- Pour maquiller la plomberie interne d'une bibliothèque.

Merci !