Tipado Gradual en Elixir

Ángel Herranz[1]

Enero 2024

[1]Universidad Politécnica de Madrid

8 de enero de 2024

Elixir es oficialmente[1] *un lenguaje “gradualmente tipado”*

José Valim

[1]Ya se están usando tipos internamente en el compilador.

Gradualmente tipado

- **Tipado estático:** el compilador pasa a ser un demostrador de teoremas (propiedades)

$$\frac{f : A \to B \qquad x : A}{f(x) : B}$$

- **Tipado gradual:** no es obligatorio y el código (legado o no) sin tipos no se verá afectado en modo alguno (ej. tiempos de ejecución)

¿Por qué tipado estático?

¿Por qué tipado estático?

Menos errores

Menos tests

Rendimiento

Documentación (experiencia como desarrollador)

Herramienta de diseño

Demostración de propiedades

¿Por qué tipado estático?

- ✖ Menos errores
- Menos tests
- Rendimiento
- Documentación (experiencia como desarrollador)
- Herramienta de diseño
- Demostración de propiedades

¿Por qué tipado estático?

- ✖ Menos errores
- ✖ Menos tests
- Rendimiento
- Documentación (experiencia como desarrollador)
- Herramienta de diseño
- Demostración de propiedades

¿Por qué tipado estático?

- ✖ Menos errores
- ✖ Menos tests
- ? Rendimiento
- Documentación (experiencia como desarrollador)
- Herramienta de diseño
- Demostración de propiedades

¿Por qué tipado estático?

- ✖ Menos errores
- ✖ Menos tests
- ? Rendimiento
- ✔ Documentación (experiencia como desarrollador)
- ✔ Herramienta de diseño
- ✔ Demostración de propiedades

¿Entusiastas?

¿Entusiastas?

La iniciativa puede fracasar por muchas razones

¿Por qué no Hindley-Milner[2]?

```
def length([]), do: 0
def length([x|xs]), do: 1 + length(xs)
```

[2]Unión disjunta y productos.

¿Por qué no Hindley-Milner[2]?

```
$ list(a) -> integer
def length([]), do: 0
def length([x|xs]), do: 1 + length(xs)
```

[2]Unión disjunta y productos.

¿Por qué no Hindley-Milner[2]?

```
$ list(a) -> integer
def length([]), do: 0
def length([x|xs]), do: 1 + length(xs)
```

```
iex> length [1, :ok, "doofinder"]
** (TypeError) iex:1:
**   No instance type for a in list(a)
```

Inadmisible en Elixir!

[2]Unión disjunta y productos.

Tipos basados en teoría de conjuntos

Set-theoretic types

Keep Calm

Tipos basados en teoría de conjuntos

Set-theoretic types

Keep Calm

Los tipos son conjuntos con unión, intersección y negación

Tipos basados en teoría de conjuntos

```
[1, :ok, "doofinder"]
```

Tipos basados en teoría de conjuntos

```
$ list(integer() or atom() or binary())
[1, :ok, "doofinder"]
```

Semántica: conjuntos

```
$ integer()
```

Semántica: conjuntos

```
$ integer()
```

$$\{x \mid \text{“}x \text{ es un entero”}\}$$

Semántica: conjuntos

`$ integer()`

$$\{x \mid \text{“}x \text{ es un entero”}\}$$

`$ atom()`

Semántica: conjuntos

```
$ integer()
```

$$\{x \mid \text{“}x \text{ es un entero”}\}$$

```
$ atom()
```

$$\{x \mid \text{“}x \text{ es un átomo”}\}$$

```
$ atom()
:ok
```

$$\texttt{:ok} \in \texttt{atom()}$$

Semántica: unión

```
$ integer() or atom()
```

$$\{x \mid \text{“}x \text{ es un entero”}\} \cup \{x \mid \text{“}x \text{ es un átomo”}\}$$

¿Por qué ***or***?

Semántica: unión

```
$ integer() or atom()
```

$$\{x \mid \text{“}x \text{ es un entero”}\} \cup \{x \mid \text{“}x \text{ es un átomo”}\}$$

¿Por qué ***or***?

$$\{x \mid \text{“}x \text{ es un entero”} \ \vee \ \text{“}x \text{ es un átomo”}\}$$

Semántica: intersección

```
$ integer() and atom()
```

$$\{x \mid \text{“}x \text{ es un entero”} \land \text{“}x \text{ es un átomo}\}$$

Semántica: intersección

`$ integer() and atom()`

$$\{x \mid \text{“}x \text{ es un entero”} \land \text{“}x \text{ es un átomo}\}$$

$$\texttt{none()} = \emptyset$$

Semántica: negación

```
$ not integer()
```

$$\{x \mid x \notin \texttt{integer()}\}$$

Semántica: negación

`$ not integer()`

$$\{x \mid x \notin \texttt{integer()}\}$$

`$ any() = not none()`

Semántica: tipos unitarios

```
$ :ok
```

```
{:ok}
```

```
$ 0
```

```
{0}
```

Semántica: diferencia

```
$ integer() and not 0
```

$$\{x \mid x \in \texttt{integer()} \wedge x \notin \texttt{0}\}$$

Semántica: diferencia

```
$ integer() and not 0
```

$$\{x \mid x \in \texttt{integer()} \wedge x \notin 0\}$$

$$\{x \mid x \in \texttt{integer()}\} \setminus 0$$

¿Sensaciones?

¿Fácil?
¿Intuitivo?

“Espacio” de funciones

```
def ineg(x) when is_integer(x) do
  -x
end
```

“Espacio” de funciones

```
$ integer() -> integer()
def ineg(x) when is_integer(x) do
  -x
end
```

“Espacio” de funciones

```
$ integer() -> integer()
def ineg(x) when is_integer(x) do
  -x
end
```

Informalmente

```
$ integer() -> integer()
```

$\{f \mid \text{“}f \text{ es una función que toma un entero y devuelve un entero”}\}$

¿Intuitivo?

```
$ integer() -> integer()
def ineg(x) when is_integer(x) do
  -x
end

$ boolean() -> boolean()
def bneg(x) when is_boolean(x) do
  not x
end
```

¿Intuitivo?

```
def neg(x) when is_integer(x) do
  -x
end
def neg(x) when is_boolean(x) do
  not x
end
```

¿Intuitivo?

```
$ (integer() -> integer()) ??? (boolean() -> boolean())
def neg(x) when is_integer(x) do
  -x
end
def neg(x) when is_boolean(x) do
  not x
end
```

¿Qué ponemos en *???*? ¿*or*? ¿*and*?

¿Intuitivo?

```
$ (integer() -> integer()) ??? (boolean() -> boolean())
def neg(x) when is_integer(x) do
  -x
end
def neg(x) when is_boolean(x) do
  not x
end
```

¿Qué ponemos en *???*? ¿*or*? ¿*and*?

¡ ***and*** !

Semántica: funciones

```
$ integer() -> integer()
```

{*f* | “*f* es una función que toma un entero y devuelve un entero”}

¿`trunc` ∈ `integer() -> integer()`?

Semántica: funciones

`$ integer() -> integer()`

~~$\{f \mid$ "f es una función que toma un entero y devuelve un entero"$\}$~~

Más formal: un contrato

$$\{f \mid x \in \texttt{integer()} \implies f(x) \in \texttt{integer()}\}$$

¿`trunc` $\in$ `integer() -> integer()`?

Semántica: funciones

```
$ integer() -> integer()
```

~~$\{f \mid$ "f es una función que toma un entero y devuelve un entero"$\}$~~

Más formal: un contrato

$$\{f \mid x \in \texttt{integer()} \implies f(x) \in \texttt{integer()}\}$$

¿`trunc` $\in$ `integer() -> integer()`?

¡Sí!

Intuitivo

```
$ (integer() -> integer()) and (boolean() -> boolean())
```

$$\{f \mid x \in \texttt{integer()} \implies f(x) \in \texttt{integer()} \wedge \\ x \in \texttt{boolean()} \implies f(x) \in \texttt{boolean()}\}$$

```
def neg(x) when is_integer(x) do
  -x
end
def neg(x) when is_boolean(x) do
  not x
end
```

¿Más o menos preciso?

```
$ (integer() -> integer()) and (boolean() -> boolean())
def neg(x) when is_integer(x) do
  -x
end
def neg(x) when is_boolean(x) do
  not x
end
```

¿Más o menos preciso?

```
$ integer() or boolean() -> integer()) or boolean()
$ (integer() -> integer()) and (boolean() -> boolean())
def neg(x) when is_integer(x) do
  -x
end
def neg(x) when is_boolean(x) do
  not x
end
```

Objetivo: el tipo más preciso

`$ (integer() -> integer()) and (boolean() -> boolean())`

más preciso que

`$ (integer() -> integer()) or (boolean() -> boolean())`[3]

más preciso que

`$ integer() or boolean() -> integer() or boolean()`

[3] *Type warning: union of function types is useless.*

Objetivo: el tipo más preciso

```
$ (integer() -> integer()) and (boolean() -> boolean())
```

más preciso que ($\subseteq$)

```
$ (integer() -> integer()) or (boolean() -> boolean())
```[3]

más preciso que ($\subseteq$)

```
$ integer() or boolean() -> integer() or boolean()
```

[3] *Type warning: union of function types is useless.*

Menos preciso, más falsos positivos

```
$ integer() or boolean() -> integer() or boolean()
def neg(x) when is_integer(x) do -x end
def neg(x) when is_boolean(x) do not x end
```

Menos preciso, más falsos positivos

```
$ integer() or boolean() -> integer() or boolean()
def neg(x) when is_integer(x) do -x end
def neg(x) when is_boolean(x) do not x end

$ integer(), integer() -> integer()
def sub(x,y), do: x + neg(y)
# Type error:          ^ integer() + boolean()
```

Ejemplo de expresividad

```
def flatten([]), do: []
def flatten([x|xs]), do: flatten(x) ++ flatten(xs)
def flatten(x), do: [x]
```

Ejemplo de expresividad

```
$ tree(a) = a and not list(any()) or list(tree(a))
```

Un árbol de valores de *a* puede ser (1) un valor de *a* siempre y cuando no sea una lista o bien (2) una lista de árboles de *a*

```
$ tree(a) -> list(a)
def flatten([]), do: []
def flatten([x|xs]), do: flatten(x) ++ flatten(xs)
def flatten(x), do: [x]
```

Gradual

- Si no se dice nada se asume que el tipo es *dynamic()*
- *dynamic()* representa un tipo desconocido

```
def f(a,b), do: ...
```

- El sistema de tipos no puede probar “nada” cuando usamos `f`

Gradual

- Si no se dice nada se asume que el tipo es *dynamic()*
- *dynamic()* representa un tipo desconocido

```
$ dynamic(), dynamic() -> ...
def f(a,b), do: ...
```

- El sistema de tipos no puede probar “nada” cuando usamos `f`

Mezclando código anotado y código no anotado

```
$ integer() -> integer()
def id(x), do: x

# Sin declaración de tipos
def debug(x), do: "dato:" <> id(x)
```

Mezclando código anotado y código no anotado

```
$ integer() -> integer()
def id(x), do: x
$ dynamic() -> dynamic()
def debug(x), do: "dato:" <> id(x)
```

Mezclando código anotado y código no anotado

```
$ integer() -> integer()
def id(x), do: x
$ dynamic() -> ¿¡error!?
def debug(x), do: "dato:" <> id(x)
```

Mezclando código anotado y código no anotado

```
$ integer() -> integer()
def id(x), do: x
$ dynamic() -> ¿¡error!?
def debug(x), do: "dato:" <> id(x)

iex> debug("doofinder")
"dato:doofinder"
```

- El tipo estático `integer()` no coincide con el tipo dinámico `binary()` (!?)

Mezclando código anotado y código no anotado

```
$ integer() -> integer()
def id(x), do: x
$ dynamic() -> ¿¡error!?
def debug(x), do: "dato:" <> check_int(id(x))

iex> debug("doofinder")
** TypeError: expected integer
```

Mezclando código anotado y código no anotado

```
$ integer() -> integer()
def id(x), do: x
$ dynamic() -> ¿¡error!?
def debug(x), do: "dato:" <> check_int(id(x))

iex> debug("doofinder")
** TypeError: expected integer
```

- Se modifica la semántica del lenguaje y se introducen problemas de rendimiento

Strong arrows

- En vez de escribir esto

```
$ integer() -> integer()
def id(x), do: x
```

- los programadores más bien escriben esto otro

```
$ integer() -> integer()
def id(x) when is_integer(x), do: x
```

- La declaración del tipo es idéntica
- pero ¡el espacio de funciones es diferente!

Strong arrows: un tipo más preciso

```
$ integer() -> integer()

def id(x) when is_integer(x), do: x

$ binary() -> binary()

def debug(x), do: "dato:" <> x

$ (integer() -> integer()) and (float() -> float())

def inc(x), do: x + 1
```

Strong arrows: un tipo más preciso

```
$ integer() -> integer()
$ and not integer() -> none()
def id(x) when is_integer(x), do: x

$ binary() -> binary()
$ and not binary() -> none()
def debug(x), do: "dato:" <> x

$ (integer() -> integer()) and (float() -> float())
$ and not integer() and not boolean() -> none()
def inc(x), do: x + 1
```

Técnica para probar *strong arrows*

- Supóngase que el tipo de la función *f* es `A -> B`
- El sistema de tipos intenta probar *f* $\in$ `not A -> any()`
- Si lo consigue la función no es *strong* y por lo tanto propaga el tipo `dynamic`

Resumen

- Un sistema de tipos muy expresivo y aún así muy intuitivo
- Correcto (no como TypeScript, por ejemplo)
- Los tipos de las funciones reflejan el contrato
- Inferencia de tipos basada en encaje de patrones y guardas (se apoya en lo que los programadores ya hacen)
- Nadie está haciendo esto

Dudas

- ¿Cómo será los tipos que el comprobador obligue a escribir?
- ¿Cómo de difícil será escribirlos?
- ¿Se entenderán?
- ¿Cuáles serán los nuevos “idioms”?
- ¿Vamos a usarlo?
- ¿Encontrarán los teóricos algún límite serio?
- ¿Cómo afectará al tiempo de compilación?

Referencias

- The Design Principles of the Elixir Type System by G. Castagna, G. Duboc, and J. Valim: artículo científico (*parametric, protocols, maps, structs*, no menos de 5 o 6 artículos si tiramos del hilo)
- ElixirConf 2023 - José Valim - The foundations of the Elixir type system
- Type system updates: moving from research into development

¿Preguntas?