Programación funcional a través de OCaml

José Castillo Lema

https://github.com/josecastillolema/talks

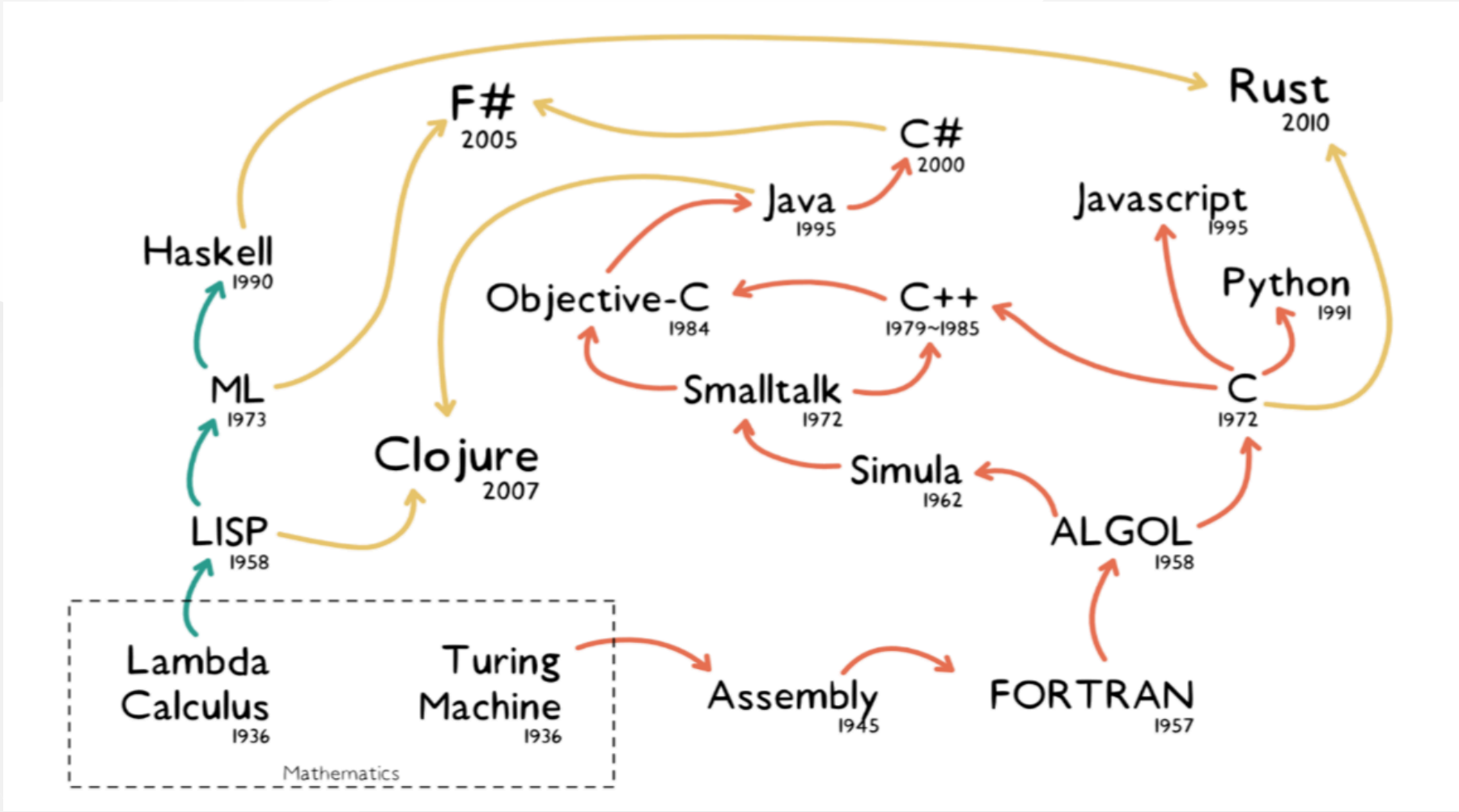
whoami

- Blog
- LinkedIn
- GitHub
- Stack Overflow
- Google Scholar
- ResearchGate
- Email

Índice

- Conceptos de programación funcional (funciones de primera clase, funciones puras, inmutabilidad, recursión)
- Fundamentos de OCaml (tipos de datos, sintax, operaciones básicas)
- Biblioteca estandar de OCaml
- Técnicas de pruebas y depuración en programación funcional
- Ejemplos del mundo real (analizadores, intérpretes, etc.)

Historia


F#
2005
Rust
2010
C#
2000
Java
1995
Javascript
1995
Haskell
1990
Objective-C
1984
C++
1979~1985
Python
1991
ML
1973
Smalltalk
1972
C
1972
Clojure
2007
Simula
1962
LISP
1958
ALGOL
1958
Lambda
Calculus
1936
Turing
Machine
1936
Assembly
1945
FORTRAN
1957
Mathematics

Programación funcional

- Características
  - **Funciones puras**, sin efectos colaterales
  - **Inmutabilidad**, generamos nuevos valores
  - **Recursión**, sin bucles
  - **Funciones de orden superior**
- Ventajas
  - Código más claro y más fácil de razonar sobre
  - Escalabilidad
  - Más fácil de testar

OCaml (Objective Caml)

- Creado en el **1996** en **Inria** proviene de una familia de lenguajes denominados **ML** (Meta Language)
- Propósito **general** y **compilado**
- **Estáticamente** y fuertemente **tipado**
- Implementa **interferencia de tipos** y **evaluación estricta**
- **Alto** nivel con ***garbage collection*** ➡️ ***memory safe***
- Multi **paradigma**
  - **Funcional**
  - Imperativa
  - Orientada a objetos
- Multi **plataforma** (incluyendo JavaScript y WebAssemby)

Tipos de lenguajes

Lenguaje	Imperativo	Funcional
Dinámico	• Python • Ruby • Javascript • PHP	• Lisp • Scheme • Clojure • Erlang • Elixir
Estático	• C • C# • Java • C++	• **OCaml** • Scala • Haskell • **F#**

Energía

	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

Software escrito en OCaml

- **Coq**: un asistente de pruebas
- Partes del cliente **Docker** de macOS
- **Facebook Messenger**: la versión web
- **MirageOS**: para crear *unikernels*
- **MLdonkey**: transferencia de archivos *peer-to-peer*
- **Tezos**: una plataforma de *bitcoin*
- **virt-v2v**: un conversor de servidores a KVM de Red Hat
- **Xen Cloud Platform** y **XenServer**: plataformas de virtualización

Frontend

- Nuevos **lenguajes**
  - **ReasonML** por Facebook
  - **ReScript** por Bloomberg
- **Compiladores** de OCaml a JavaScript
  - **js_of_ocaml**
  - **Melange**
- **Influencias**
  - **Elm**

Usuarios

- Ahrefs
- Bloomberg
- Citrix
- Docker
- Facebook
- Jane Street
- Red Hat

OCaml *vs* Python

En **OCaml**:

```
(* sum : int list → int *)
let rec suma = function
  | [] → 0
  | h::t → h + sum t

assert (suma [1;2;3;4] = 10)
print_endline "Éxito!"
```

En **Python**:

```
def suma (list):
  total = 0
  for num in list:
    total = total + num
  return total

assert (suma([1,2,3,4]) == 10)
print("Éxito!")
```

OCaml *vs* Python

En **OCaml**, no compila:

```
suma ['a'; 'b'; 'c'; 'd']
(* Error: This expression has type char
   but an expression was expected of type int *)

suma [1.1; 2.; 3.; 4.]
(* Error: This expression has type float
   but an expression was expected of type int *)
```

En **Python** compila, pero falla en tiempo de ejecución:

```
suma(['a','b','c','d'])
# Traceback (most recent call last):
#  In sum
#     total = total + num
#             ~~~~~~^~~~
# TypeError: unsupported operand type(s) for +:
#  'int' and 'str'

suma([1.1, 2, 3, 4])
# 10.1
```

Tipos

Tipo	Ejemplo	Operaciones
`unit`	`()`	
`bool`	`true` `false`	`&&` `||`
`int`	`1`	`+` `-` `*` `/` `**`
`float`	`1.`	`+.` `-.` `*.` `/.`
`char`	`'a'`	`^`
`string`	`"hola"`	`^`
`list`	`[1; 2]`	`::` `@`
`'a * 'a`	`(1, 2)`	`fst` `snd`
`ref`	`ref 1`	`!` `:=`
`array`	`[| 1; 2; |]`	`.()` `<-`

OCaml *vs* Python (tipos)

En **OCaml**:

```
[1; 'a']
(* Error: This expression has type char
   but an expression was expected of type int *)

1 + 2
(* -: int = 3 *)

1.1 +. 2.1
(* - : float = 3.2 *)

"a" ^ "b"
(* - : string = "ab" *)

1.1 +. 2
(* Error: This expression has type int
   but an expression was expected of type float *)
```

En **Python**:

```
[1, 'a']
# [1, 'a']

1 + 2
# 3

1.1 + 2.1
# 3.2

"a" + "b"
# 'ab'

1.1 + 2
#3.1
```

Product types (registros) y *union types* (variantes)

Tenemos que representar la siguiente regla de negocio:

> **Los clientes deben tener email o direccion**

En **OCaml**:

```
type contacto =
  | Email of string
  | Direccion of string
  | EmailyDireccion of string * string

type persona = {
  nombre: string;
  contacto: contacto;
}
```

En **Python**:

- Usando clases y herencia?

Aplicación parcial

En **OCaml**:

```
(* suma : int → int → int *)
let suma x y = x + y

(* suma1: suma1 : int → int *)
let suma1 = suma 1

suma1 1
(* - : int = 2 *)
```

En **Python**, por defecto:

```
def suma (x, y):
  return x + y

suma (1)
# TypeError: suma() missing 1
# required positional argument: 'y'
```

Usando la librería [functools]():

```
from functools import partial
suma1 = partial (suma, 1)
suma1 (1)
# 2
```

Pipes

En **OCaml**:

```
suma1 (suma1 4)
(* - : int = 6 *)

suma1 @@ suma1 4
(* - : int = 6 *)

4 ▷ suma1 ▷ suma1
(* - : int = 6 *)
```

En **Python**:

```
suma1 (suma1 (4))
```

Usando la biblioteca *pipe*?

Polimorfismo

OCaml implementa **polimorfismo paramétrico**:

```
(* 'a → 'a → bool *)
let compara input1 input2 =
  input1 = input2
```

OCaml no implementa **polimorfismo *ad-hoc***:

```
1 + 1
(* - : int = 2 *)

1. + 1.
(* Error: This expression has type float
   but an expression was expected of type int *)
```

Python también implementa **polimorfismo paramétrico**:

```
def igual (a, b):
  return (a==b)
```

Python sí implementa **polimorfismo *ad-hoc***:

```
1 + 1
# 2

1. + 1.
# 2.

"a" + "b"
# 'ab'
```

Option

En **OCaml**:

```ocaml
type 'a option =
  | None
  | Some of 'a

(* 'a list → 'a *)
List.hd []
(* Exception: Failure "hd". *)

(* 'a option list → 'a option *)
let hd = function
  | h::t → Some h
  | [] → None

hd [];;
(* - : 'a option = None *)

hd [1; 2; 3]
(* - : int option = Some 1 *)
```

En **Python**:

```python
[][0]
# IndexError: list index out of range

def hd (list):
  try:
    return list[0]
  except:
    return None

hd ([])
#

hd ([1,2,3])
# 1
```

Pattern matching

En **OCaml**:

```
match (hd [2]) with
  | None → 0
  | Some x → x
(* - : int = 2 *)
```

En **Python**, desde la versión **3.10**:

```
match hd ([2]):
  case None: print (0)
  case int(n): print(n)
# 2
```

Map

En **OCaml**:

```
(* ('a → 'b) → 'a list → 'b list *)
List.map ((+) 1) [1; 2; 3]
(* - : int list = [2; 3; 4] *)
```

En **Python**, usando un bucle `for`:

```
lista_nueva = []
for elemento in [1, 2, 3]:
  lista_nueva.append(elemento + 1)
lista_nueva
# [2, 3, 4]
```

Usando *list comprehension*:

```
[x+1 for x in [1, 2, 3]]
# [2, 3, 4]
```

Fold (reduce)

En **OCaml**:

```
(* int list -> int *)
let rec sum = function
  | [] -> 0
  | h :: t -> h + sum t

(* int list -> int *)
let sum' l = List.fold_left ( + ) 0 l
```

En **Python**:

```
# Usando la función incorporada
sum ([1, 2])
# 3

# Usando for
total_sum = 0
for item in mylist:
    total_sum += item

# Usando list comprehension
def sum2 (list):
  acc = 0
  return [acc := acc + x for x in list][-1]
```

Filter

En **OCaml**:

```
List.filter (fun x -> x>2) [1; 2; 3]
(* - : int list = [3] *)
```

En **Python**:

```
l = filter(lambda x: x > 2, [1, 2 ,3])
print(list(l))
# [3]
```

Tooling

- utop: entorno interactivo (REPL)
- dune: sistema de compilación
- opam: gestor de paquetes
- ocaml-lsp: el protocolo de servidor de lenguaje de OCaml
- OCaml platform: extensión para VSCode

Futuro

- OCaml 5
  - Soporte a *multicore*
    - Concurrencia ➡️ *effect handlers*
    - Paralelismo ➡️ *domains*

Referencias

- Why OCaml por Yaron Minsky
- Nekoma Talks #5 - How did we arrive at this mess? por Edil Medeiros

Recursos

- OCaml website
- Curso de la Universidad de Cornell: OCaml Programming: Correct + Efficient + Beautiful
- Libro: Real World OCaml - Functional programming for the masses
- Blogs
  - F# for Fun and Profit
  - Thomas Leonard's blog
- Desafíos
  - 99 problemas inspirados en Ninety-Nine Lisp Problems
  - Learn OCaml
  - Exercism