

PROGRAMACIÓN FUNCIONAL λ

JAVIER ARIAS
LEONARDO MIKEL CERVANTES
ARNAU ROGER SOLÉ

SISTEMAS OPERATIVOS
UNAM
FACULTAD DE INGENIERÍA



PROGRAMACIÓN FUNCIONAL

La programación funcional es un paradigma centrado en el uso de funciones puras y estructuras inmutables. Su enfoque resulta útil para abordar problemas clásicos de concurrencia, como condiciones de carrera o bloqueos, que suelen surgir en modelos imperativos con estado compartido mutable.



Haskell



Agda



Idris

APLICACIONES COMUNES

Aplicaciones
concurrentes



Procesamiento
de datos masivos



Sistemas
financieros



FUNDAMENTOS

DE LA PROGRAMACIÓN FUNCIONAL



FUNCIONES PURAS

Una función es pura si:

- Siempre devuelve la misma salida para una misma entrada.
- No produce efectos secundarios en ningún otro componente del programa, como mutación de datos, consulta a una base de datos o lectura o escritura en un archivo.



```
(* Función pura que duplica los elementos de una lista *)
let duplicar_lista lst =
  List.map (fun x -> x * 2) lst

(* Función pura que filtra los pares *)
let filtrar_pares lst =
  List.filter (fun x -> x mod 2 = 0) lst

let () =
  let original = [1; 2; 3; 4; 5] in

  let duplicados = duplicar_lista original in
  let pares = filtrar_pares duplicados in

  print_endline "Original:";
  List.iter (fun x -> print_int x; print_string " ") original;
  print_newline ();

  print_endline "Duplicados:";
  List.iter (fun x -> print_int x; print_string " ") duplicados;
  print_newline ();

  print_endline "Pares (de los duplicados):";
  List.iter (fun x -> print_int x; print_string " ") pares;
  print_newline ()
```

INMUTABILIDAD

Las estructuras de datos no cambian. Si se necesita modificar algo, se crea una nueva versión. De esta manera es más fácil encontrar donde se está produciendo un error al no presentar estado compartido entre distintos procesos.



```
(* Función pura que duplica los elementos de una lista *)
let duplicar_lista lst =
  List.map (fun x -> x * 2) lst

(* Función pura que filtra los pares *)
let filtrar_pares lst =
  List.filter (fun x -> x mod 2 = 0) lst

let () =
  let original = [1; 2; 3; 4; 5] in

  let duplicados = duplicar_lista original in
  let pares = filtrar_pares duplicados in


  print_endline "Original:";
  List.iter (fun x -> print_int x; print_string " ") original;
  print_newline ();

  print_endline "Duplicados:";
  List.iter (fun x -> print_int x; print_string " ") duplicados;
  print_newline ();

  print_endline "Pares (de los duplicados):";
  List.iter (fun x -> print_int x; print_string " ") pares;
  print_newline ()
```

FUNCIONES COMO VALORES DE PRIMERA CLASE

En comparación con la programación orientada a objetos en la programación funcional no existen las clases, por lo que en este paradigma todo está compuesto por funciones. Es por eso que las funciones pueden pasarse como argumentos, devolverse desde otras funciones y almacenarse en estructuras de datos.



```
let aplicar_doble f x = f (f x)

let doble x = x * 2

let () =
  let resultado = aplicar_doble doble 3 in
  Printf.printf "Aplicar doble: %d\n" resultado
```

COMPOSICIÓN DE FUNCIONES

Programar combinando funciones pequeñas en funciones más complejas. La composición de funciones implica combinar varias funciones para crear una nueva



```
let sumar1 x = x + 1
let por2 x = x * 2

(* Componemos: primero suma 1, luego multiplica por 2 *)
let nueva_funcion = por2 % sumar1

let () =
  let resultado = nueva_funcion 3 in
  Printf.printf "Composición: %d\n" resultado
```


EVALUACIÓN PEREZOSA

Se utiliza en sólo algunos lenguajes, en esta no se calculan los resultados de una función de inmediato sino hasta que realmente sean necesarios. Aunque algunos lenguajes funcionales como Haskell usan evaluación perezosa por defecto, OCaml usa evaluación estricta (Los resultados de las funciones se calculan inmediatamente).

LA CONCURRENCIA SIN BLOQUEO

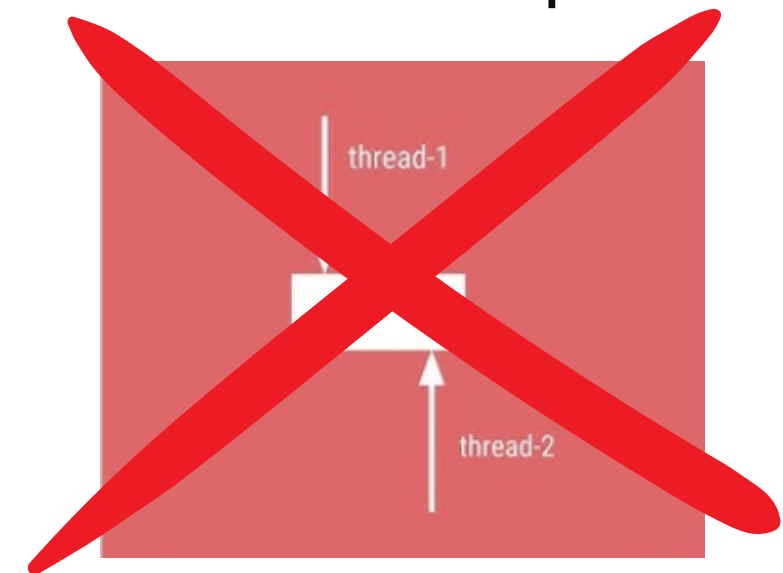
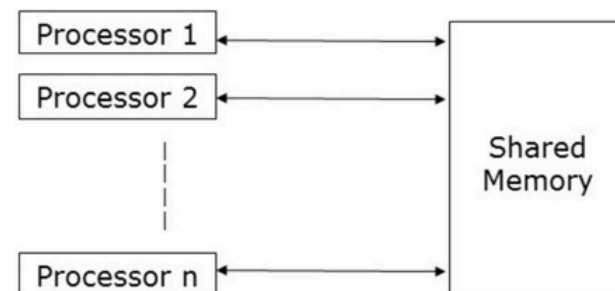
Las funciones puras no tienen efectos secundarios.

Cada función trabaja con sus propios datos.

A diferencia de muchos de los lenguajes de programación imperativos, en la programación funcional no se tiene el estado mutable compartido.

Se eliminan las condiciones de carrera existentes a causa del estado compartido.

Shared Memory



¿ENTONCES QUE HACEMOS?

Cada entidad se modela de manera independiente



Se utilizan mensajes o eventos para la comunicación en lugar de memoria compartida.



Se diseña un sistema basado en comunicaciones explícitas, en lugar de un conjunto de hilos que modifican valores en común.

EJEMPLO PRÁCTICO

```
let num_filosofos = 5

(* Creamos los palillos como canales de eventos *)
let palillos = Array.init num_filosofos (fun _ -> Event.new_channel ())

(* Inicializar cada palillo con un mensaje (disponible) *)
let () =
  Array.iter (fun ch ->
    ignore (Thread.create (fun () -> Event.sync (Event.send ch ()))) ())
  ) palillos

(* Función del filósofo con flushes *)
let filosofo id =
  while true do
    Printf.printf "Filósofo %d está pensando.\n%" id;
    Thread.delay (Random.float 2.0);

    Printf.printf "Filósofo %d tiene hambre.\n%" id;

    let primero, segundo =
      if id mod 2 = 0 then (id, (id + 1) mod num_filosofos)
      else ((id + 1) mod num_filosofos, id)
    in
```

```
Printf.printf "Filósofo %d intenta tomar el palillo %d.\n%" id primero;
Event.sync (Event.receive palillos.(primero));
Printf.printf "Filósofo %d toma el palillo %d.\n%" id primero;

Printf.printf "Filósofo %d intenta tomar el palillo %d.\n%" id segundo;
Event.sync (Event.receive palillos.(segundo));
Printf.printf "Filósofo %d toma el palillo %d.\n%" id segundo;

Printf.printf "Filósofo %d está comiendo.\n%" id;
Thread.delay (Random.float 2.0);

Printf.printf "Filósofo %d libera los palillos.\n%" id;
Event.sync (Event.send palillos.(primero) ());
Event.sync (Event.send palillos.(segundo) ());
done

(* Lanzar filósofos *)
let () =
  let rec lanzar_filosofos = function
    | 5 -> ()
    | n ->
      ignore (Thread.create filosofo n);
      lanzar_filosofos (n + 1)
  in
  lanzar_filosofos 0;

(* Mantener el programa activo *)
while true do
  Thread.delay 1.0
done
```

DESCRIPCIÓN DEL PROGRAMA

```
let num_filosofos = 5

(* Creamos los palillos como canales de eventos *)
let palillos = Array.init num_filosofos (fun _ -> Event.new_channel ())

(* Inicializar cada palillo con un mensaje (disponible) *)
let () =
  Array.iter (fun ch ->
    ignore (Thread.create (fun () -> Event.sync (Event.send ch ())) ()))
  ) palillos
```

```
(* Función del filósofo con flushes *)
let filosofo id =
  while true do
    Printf.printf "Filósofo %d está pensando.\n%" id;
    Thread.delay (Random.float 2.0);

    Printf.printf "Filósofo %d tiene hambre.\n%" id;

    let primero, segundo =
      if id mod 2 = 0 then (id, (id + 1) mod num_filosofos)
      else ((id + 1) mod num_filosofos, id)
    in

    Printf.printf "Filósofo %d intenta tomar el palillo %d.\n%" id primero;
    Event.sync (Event.receive palillos.(primero));
    Printf.printf "Filósofo %d toma el palillo %d.\n%" id primero;

    Printf.printf "Filósofo %d intenta tomar el palillo %d.\n%" id segundo;
    Event.sync (Event.receive palillos.(segundo));
    Printf.printf "Filósofo %d toma el palillo %d.\n%" id segundo;

    Printf.printf "Filósofo %d está comiendo.\n%" id;
    Thread.delay (Random.float 2.0);

    Printf.printf "Filósofo %d libera los palillos.\n%" id;
    Event.sync (Event.send palillos.(primero) ());
    Event.sync (Event.send palillos.(segundo) ());
  done
```

DESCRIPCIÓN DEL PROGRAMA

```
Printf.printf "Filósofo %d intenta tomar el palillo %d.\n%!" id primero;
Event.sync (Event.receive palillos.(primero));
Printf.printf "Filósofo %d toma el palillo %d.\n%!" id primero;

Printf.printf "Filósofo %d intenta tomar el palillo %d.\n%!" id segundo;
Event.sync (Event.receive palillos.(segundo));
Printf.printf "Filósofo %d toma el palillo %d.\n%!" id segundo;

Printf.printf "Filósofo %d está comiendo.\n%!" id;
Thread.delay (Random.float 2.0);

Printf.printf "Filósofo %d libera los palillos.\n%!" id;
Event.sync (Event.send palillos.(primero) ());
Event.sync (Event.send palillos.(segundo) ());
done
```

```
(* Lanzar filósofos *)
let () =
  let rec lanzar_filosofos = function
    | 5 -> ()
    | n ->
        ignore (Thread.create filosofo n);
        lanzar_filosofos (n + 1)
  in
    lanzar_filosofos 0;

(* Mantener el programa activo *)
while true do
  Thread.delay 1.0
done
```

VENTAJAS Y LIMITACIONES

DE LA PROGRAMACIÓN FUNCIONAL



VENTAJAS

- Simpleza
- Implementación de pruebas
- Reusabilidad del código
- Escalabilidad

LIMITACIONES

- Uso de memoria
- Rendimiento
- Cantidad de usuarios y herramientas

REFERENCIAS

- Neumann, J. (2022). Fundamentals of functional programming. Medium.
<https://medium.com/twodigits/fundamentals-of-functional-programming-e808918c8b47>
- Neumann, J. (2022). Advantages and disadvantages of functional programming. Medium.
<https://medium.com/twodigits/advantages-and-disadvantages-of-functional-programming-52a81c8bf446>
- Jeffery, J. (2017). JavaScript: What are pure functions and why use them? Medium.
<https://medium.com/@jamesjefferyuk/javascript-what-are-pure-functions-4d4d5392d49c>
- Spolsky, J. (2006). Can your programming language do this? Joel on Software.
<https://www.joelonsoftware.com/2006/08/01/can-your-programming-language-do-this/>

