

Arias Hernández Javier

Cervantes Mateos Leonardo Mikel

Solé Pi Arnau Roger

Introducción

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.

Esta exposición presenta los principios básicos de la programación funcional y explora cómo su modelo facilita la concurrencia sin necesidad de sincronización explícita. A través de un ejemplo práctico en OCaml, se ilustran sus ventajas y también se discuten sus limitaciones en el contexto de la ejecución concurrente.

Algunos campos donde se utiliza comúnmente la programación funcional son:

- Aplicaciones concurrentes: Al no haber mutaciones de estado en la programación funcional es más fácil trabajar en paralelo al evitar las condiciones de carrera.
- Procesamiento de datos masivos: Permite aplicar funciones puras a grandes volúmenes de datos en paralelo sin provocar efectos secundarios.
- Sistemas financieros: Es necesario obtener resultados precisos y fiables con grandes volúmenes de datos en muy poco tiempo.

Fundamentos de la programación funcional

Funciones puras

Una función es pura si:

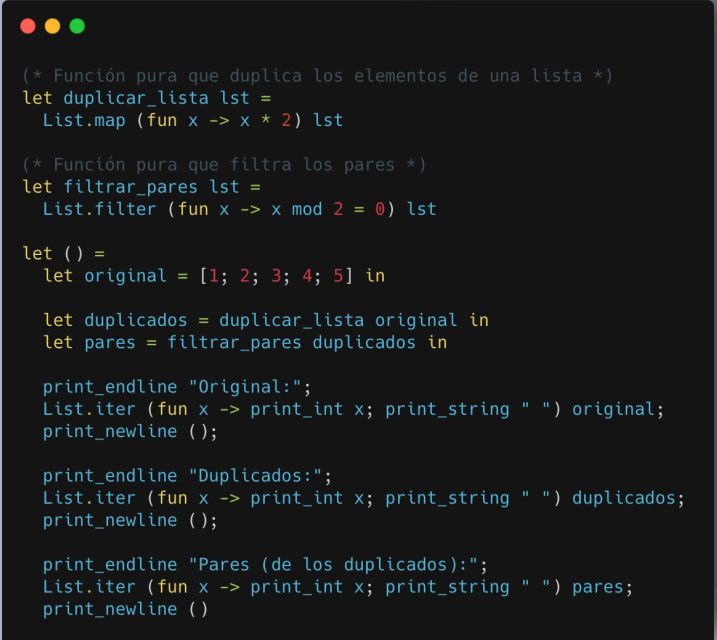
- Siempre devuelve la misma salida para una misma entrada. Esto quiere decir que el resultado debe depender únicamente de la entrada y no de algún

estado o valor externo. Por ejemplo si tenemos una función $f(x) = x + 5$, si la entrada x es 3 el resultado siempre será igual a 8.

- 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.

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 ();

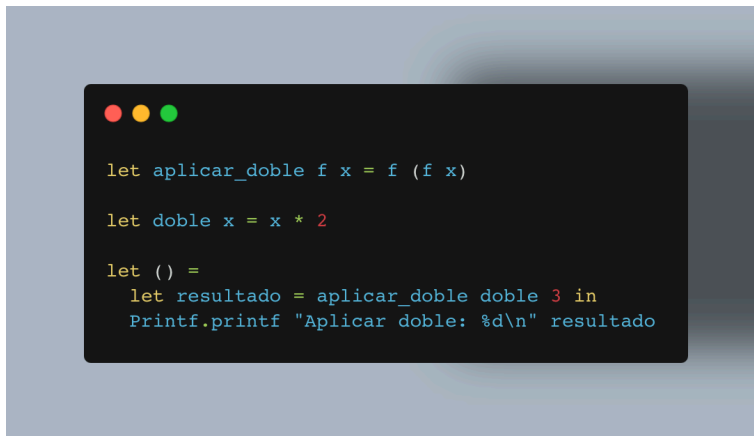
  print_endline "Duplicados:";
  List.iter (fun x -> print_int x; print_string " ") duplicados;
  print_newline ();

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

Ejemplo de funciones puras e inmutabilidad.

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

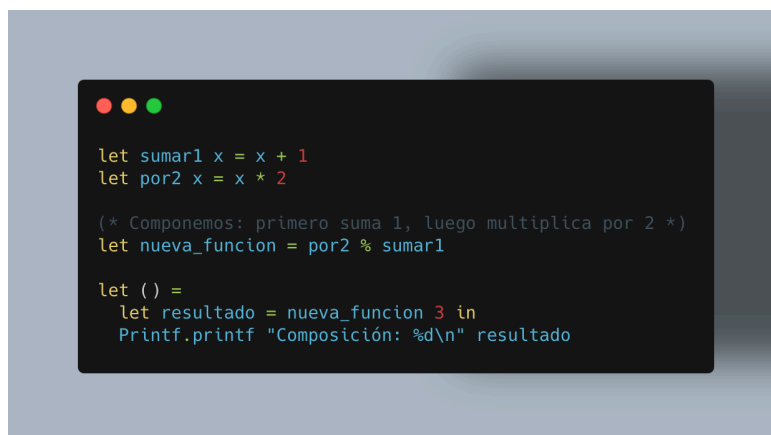
let () =
  let resultado = aplicar_doble doble 3 in
  Printf.printf "Aplicar doble: %d\n" resultado

```

Ejemplo de funciones como valores de primera clase.

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

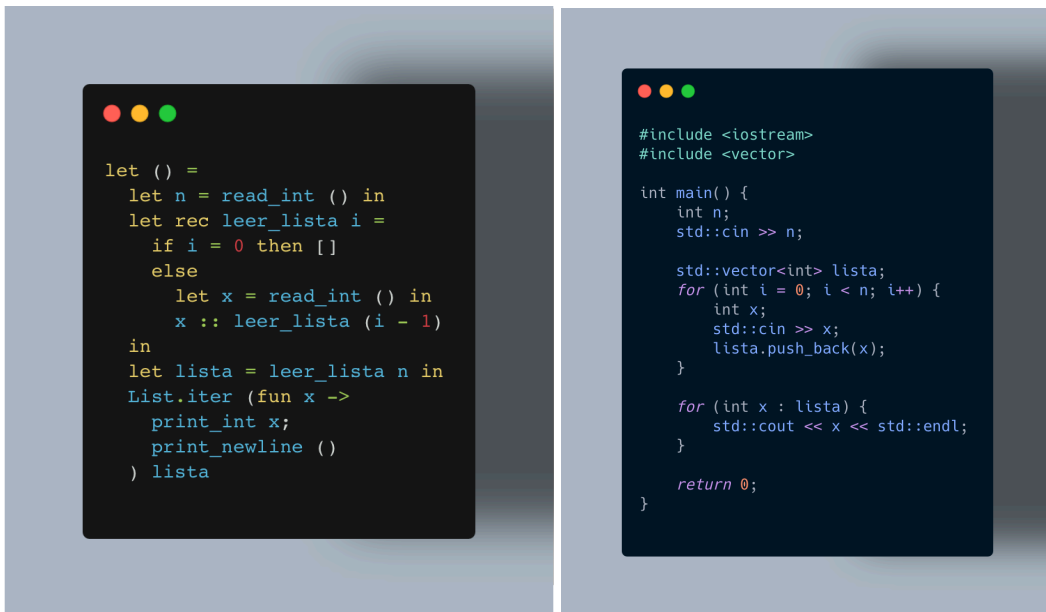
let () =
  let resultado = nueva_funcion 3 in
  Printf.printf "Composición: %d\n" resultado

```

Ejemplo de composición de funciones.

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). Se pueden usar estructuras perezosas si se requiere, pero no son clave para la concurrencia sin bloqueo.



Modelo funcional, la concurrencia sin bloqueo

En la programación funcional las funciones puras **no tienen efectos secundarios** y no hay estado mutable compartido, por lo que **cada función trabaja con sus propios datos**. Esto elimina muchas causas de errores en programas concurrentes ya que la inmutabilidad elimina las condiciones de carrera existentes a causa del estado compartido.

En la programación imperativa (como en C, Java o Python con **threading**), es común resolver problemas de concurrencia **compartiendo memoria** entre hilos y **usando bloqueos** (**mutexes**, **semaphores**, etc.) para evitar condiciones de carrera. Esto permite pensar el programa como si todo ocurriera en una única línea de ejecución, solo que con protección.

En la programación funcional no se tiene el estado mutable compartido, por lo que es necesario:

- Modelar cada entidad como independiente.
- Utilizar mensajes o eventos para la comunicación en lugar de memoria compartida.

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

Ejemplo Práctico

Veremos cómo funciona esto en el problema de la cena de los filósofos.

En clase el ejemplo que vimos con python utiliza...

```
#!/usr/bin/python3
import threading

num_filosofos = 5
palillos = [threading.Semaphore(1) for i in range(num_filosofos)]

def filosofo(yo):
    while True:
        piensa(yo)
        # ¡Hace hambre!
        come(yo)

def piensa(yo):
    print(f'{" " * yo}{yo} tengo asuntos muuuuy importantes por pensar')
    print(f'{" " * yo}{yo} como que ya hace hambre...')

def come(yo):
    # Para evitar los bloqueos mutuos, los filósofos diestros toman primero el
    # palillo derecho, y los zurdos toman primero el derecho: ya no se pueden
    # formar ciclos de todos esperando al de junto.
    if yo % 2 == 0:
        print(f'{" " * yo}{yo}: Diestro. Tomo palillo derecho')
        palillos[yo].acquire()
        print(f'{" " * yo}{yo}: Diestro. Tomo palillo izquierdo')
        palillos[(yo + 1) % num_filosofos].acquire()
    else:
        print(f'{" " * yo}{yo}: Zurdo. Tomo palillo izquierdo')
        palillos[(yo + 1) % num_filosofos].acquire()
        print(f'{" " * yo}{yo}: Zurdo. Tomo palillo derecho')
        palillos[yo].acquire()

    print(f'{" " * yo}{yo}: ¡Rico! Arroz amargo')

    # Ya terminamos de comer, dejamos ambos palillos. No reportamos, porque esta
    # operación no puede llevar a bloqueo de ningún tipo.
    palillos[yo].release()
    palillos[(yo + 1) % num_filosofos].release()

for i in range(num_filosofos):
    threading.Thread(target=filosofo, args=[i]).start()
```

```

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

```

Cuando queremos aplicar el **modelo funcional** al problema de la Cena de los Filósofos, **debemos cambiar totalmente la forma de resolverlo**. En lugar de compartir palillos como estado mutable protegido con locks, **lo que haremos será:**

- **Modelar cada entidad de forma separada**

Cada filósofo y cada palillo vive en su propio hilo, con su propio canal de comunicación, sin compartir memoria.

- **Usar mensajes para adquirir y liberar palillos**

Los filósofos “piden” un palillo enviando un mensaje al proceso (hilo) que lo representa, y esperan la respuesta (sin bloquear ningún recurso global).

- **Usar mensajes para coordinar el ciclo de comer**

Una vez que el filósofo recibe confirmación de ambos palillos, procede a comer; al terminar, “envía” de vuelta la señal de liberación a esos mismos canales.

Descripción del programa en OCaml

1. Inicialización

- Se crean `num_filosofos = 5` canales (`Event.new_channel ()`), uno por palillo.
- Un pequeño hilo inicial por canal envía un mensaje vacío (`()`) para marcar cada palillo como “disponible” desde el arranque.

2. Función filósofo `id`

- Corre indefinidamente en su propio hilo (`Thread.create`).
- **Pensar**: pausa aleatoria.
- **Hambre**: decide en qué orden tomará los palillos (pares primero el izquierdo, nones primero el derecho) para evitar deadlocks.

Tomar palillos:

```
Event.sync (Event.receive palillos.(primer_palillo));
```

```
Event.sync (Event.receive palillos.(segundo_palillo));
```

- Cada `receive` hace que el filósofo espere hasta que el canal “libere” la señal.

- **Comer:** pausa aleatoria.

Liberar palillos:

Event.sync (Event.send palillos.(primer_palillo) ());

Event.sync (Event.send palillos.(segundo_palillo) ());

- Devolvemos la señal de disponibilidad.

3. Ejecución concurrente

- Cinco hilos independientes lanzan filosofo 0...filosofo 4.
- Un bucle principal mantiene el programa vivo.

Gracias a este rediseño, **no hay estado compartido mutable** ni locks: la sincronización ocurre de manera explícita y segura por **pasaje de mensajes**, ilustrando cómo la programación funcional facilita una concurrencia sin bloqueo ni condiciones de carrera.

Ventajas y limitaciones

Las ventajas que presenta la programación funcional son:

- **Simpleza:** El uso de funciones puras e inmutabilidad obligan a elaborar funciones simples y claras, por lo que es más fácil entender el código del programa. Esto no indica que sea más fácil de escribir, al contrario, se requiere de un mayor esfuerzo para poder implementar este tipo de funciones cumpliendo con el objetivo necesario.
- **Pruebas:** Es más sencillo realizar pruebas para funciones puras ya que dependen únicamente de la entrada. Esto permite implementar pruebas unitarias fácilmente.
- **Reusabilidad:** Gracias a las funciones como valores de primera clase y la composición de funciones es posible reutilizar las funciones en distintos escenarios.

- Escalabilidad: Las funciones puras y la inmutabilidad impiden las condiciones de carrera por lo que cada uno de los hilos puede ejecutarse libremente sin bloqueos para ser sincronizados. Esto permite la escalabilidad del programa al poder agregar más hilos de una manera más sencilla.

De igual manera se presentan ciertas desventajas y limitaciones como:

- Uso de memoria y rendimiento: La inmutabilidad tiene la desventaja de tener que crear una copia cada vez que se quiera modificar una variable, por lo que será necesario un mayor uso de la memoria. Esto también puede generar un impacto en el rendimiento si no se realiza de la manera correcta debido al procesamiento necesario para copiar los datos. Es por esto que algunos programas utilizan estrategias como la evaluación perezosa para reducir estos efectos.
- Cantidad de usuarios y herramientas: Debido a que la programación funcional no es un paradigma utilizado comúnmente no existen tantas herramientas y frameworks como en otros lenguajes más populares como los son Python y JavaScript. De igual manera la cantidad de usuarios y expertos es menor.

Conclusiones

Referencias

- Neumann, J. (2022). *Fundamentals of functional programming*. Medium. <https://medium.com/twodigits>
- Neumann, J. (2022). *Advantages and disadvantages of functional programming*. Medium. <https://medium.com/twodigits>
- Jeffery, J. (2017). *JavaScript: What are pure functions and why use them?* Medium. <https://medium.com/@jamesjefferyuk>
- Spolsky, J. (2006). *Can your programming language do this?* Joel on Software. <https://www.joelonsoftware.com>