Programación Declarativa.

Mezclando estilo imperativo y funcional.

Selene Linares Arévalo.

Facultad de Ciencias, UNAM

2016-2.

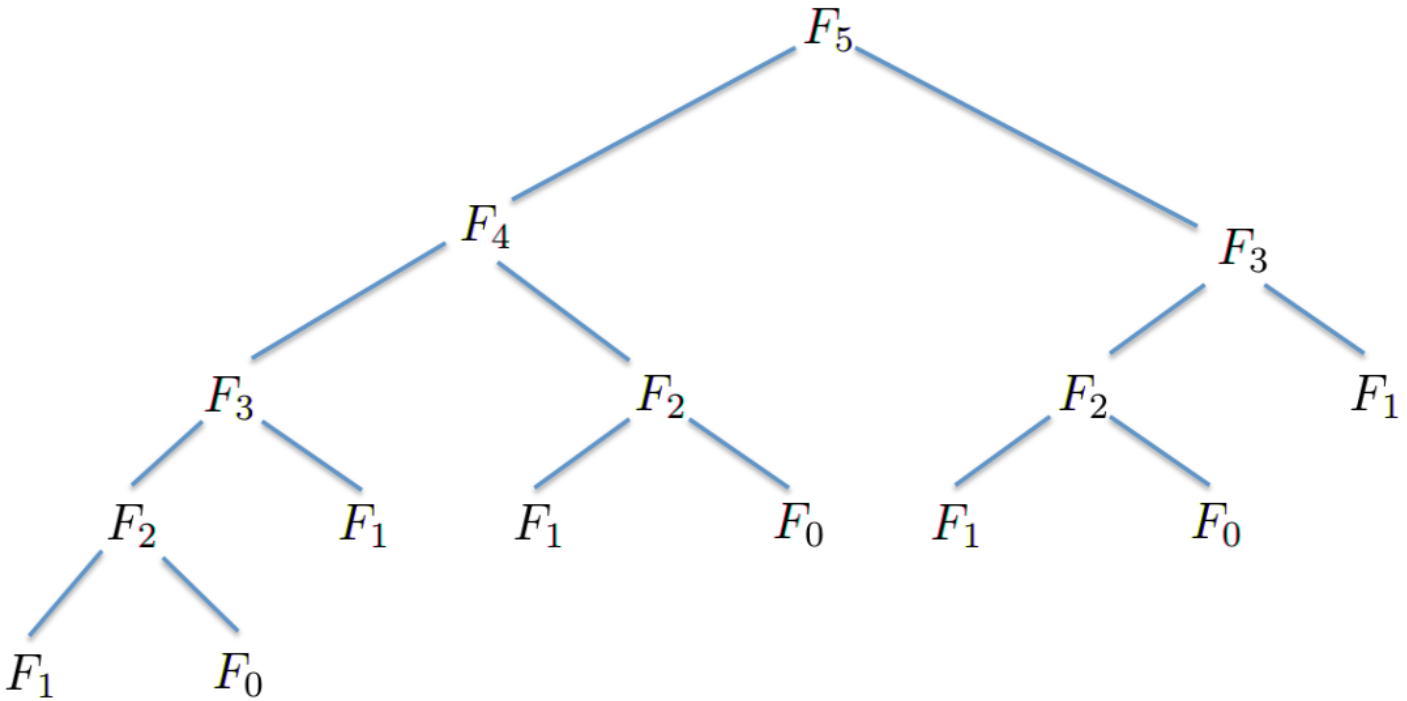
Memoización

Consideremos la función para calcular el n-ésimo número de Fibonacci:

```
# let rec fibR i = match i with
                   | 0 -> 0
                   | 1 -> 1
                   | _ -> fibR (i-1) + fibR (i-2) ;;
val fibR : int -> int = <fun>
```

Memoización

La anterior definición es ineficiente. Para calcular `fib n` se requiere evaluar `fib (n-2)` y `fib (n-1)`, éste último nuevamente calculará `fib (n-2)`, y dentro de cada llamada a `fib (n-2)` habrá dos llamadas a `fin (n-4)`.



Memoización

¿Cómo podríamos evitar estos cálculos repetidos?
Observemos que para obtener el n-ésimo número, necesitamos los dos valores previos; podríamos entonces, regresar ambos valores en el resultado:

```
# let rec fibP i = match i with
                   | 0 -> (0,1)
                   | _ -> let (x,y) = fibP (i-1)
                          in (y,x+y) ;;
val fibP : int -> int * int = <fun>
```

Memoización

La **Memoización** es una técnica de optimización de programas a nivel de ejecución que evita recalcular resultados durante la evaluación de expresiones, lo cual sucede particularmente en algunas soluciones recursivas a ciertos problemas.

Una función *memoizada* es similar a una función ordinaria, excepto que recuerda algunos o todos los argumentos a los que ha sido aplicada junto con los resultados obtenidos. Estos pares, se almacenan en una tabla llamada *tabla de memoización*. Entonces cuando una función memoizada es evaludada reutiliza, si es posible, resultados antes calculados.

Cualquier función puede ser memoizada pero puede suceder que no mejore su desempeño a pesar de estarse empleando una técnica de optimización.

Memoización

Ejemplos.

Evaluación perezosa

En OCaml, el valor de la expresión condicional
`if true then e1 else e2` es el valor de `e1`, y `e2` nunca es evaluada.

```
# List.hd [];;
Exception: Failure "hd".

# if true then 4 else List.hd [];;
- : int = 4
```

Esto es a lo que llamamos *evaluación perezosa* pero no es el caso en la mayoría de expresiones, de hecho OCaml emplea evaluación *ansiosa*.

Evaluación perezosa

Cuando una función es aplicada a argumentos, éstos son evaluados de forma ansiosa, y la función se aplica a los valores obtenidos. Por ejemplo:

```
# let if_a x e1 e2 = match x with
                     | true -> e1
                     | false -> e2;;
val if_a : bool -> 'a -> 'a -> 'a = <fun>

# if_a true 4 (List.hd []);;
Exception: Failure "hd".
```

Evaluación perezosa

Es posible simular evaluación perezosa a través de **thunks**. Un *thunk* es una función de la forma `fun () -> ...` . Éste toma ventaja del hecho de que el cuerpo de una función no se evalúa al definir la función sino cuando es aplicada, es decir, el cuerpo de la función es evaluado de forma perezosa.

```
# let f = fun () -> List.hd [];;
val f : unit -> 'a = <fun>

# f ();;
Exception: Failure "hd".
```

Evaluación perezosa

Con lo anterior podemos redefinir `if` utilizando thunks:

```
# let if_l x e1 e2 = match x with
                     | true -> e1()
                     | false -> e2();;
val if_l:bool -> (unit ->'a) -> (unit ->'a) -> 'a =<fun>

# ite true (fun () -> 4) (fun () -> List.hd []);;
- : int = 4
```

En OCaml una función `fun x ->` e es considerada un valor y no se realiza ningún intento de evaluarla hasta que se aplica la función.

Evaluación perezosa

Sin embargo, emplear *thunks* no siempre es la mejor solución para simular evaluación perezosa. Consideremos la siguiente expresión:

```
# List.map (fun x -> x + (expr_costosa)) xs
```

donde `expr_costosa` no depende de `x`.

En este caso, nos gustaría realizar la evaluación de `expr_costosa` fuera del cómputo de `map`, digamos:

```
# let w = expr_costosa
      in List.map (fun x -> x + w ) xs
```

Evaluación perezosa

En un lenguaje con evaluación perezosa, la expresión `w` sería evaluada exactamente cero o una vez como queremos, pero no es nuestro caso y utilizar *thunks* no funciona en esta ocasión:

```
# let w () = expr_costosa
        in map (fun x -> x + w ()) xs
```

¿Por qué no nos sirve esta definición?.

Evaluación perezosa

Consideremos las siguientes definiciones:

```
let fibonacci n = let rec fib a b i =
                   if i = n then a else fib b (a+b) (i+1)
                  in  fib 0 1 0;;

let millionth_fib = fun() -> fibonacci 1000000;;

let show_the_fib fib = function
  | true -> Some (fib())
  | false -> None;;

let print_the_fib fib = function
  | true -> print_int (fib())
  | false -> ();;
```

Evaluación perezosa

Buscamos definir `millionth_fib_lazy` con las siguientes características:

- `million_fib_lazy` debe ser una función thunk si nunca se ha evaluado.
- `million_fib_lazy` debe ser un número inmediatamente después de la primera vez que es evaluada.
- En futuros intentos de evaluar `million_fib_lazy` debe regresarse el número directamente.
- `million_fib_lazy` debe tener un tipo asociado al cual vamos a llamar `lazy_state`.

Evaluación perezosa

Declaramos tipos para representar un valor perezoso.

```
# type 'a lazy_state =
    | Delayed of (unit -> 'a)
    | Value of 'a
    | Exn of exn ;;
type 'a lazy_state = Delayed of (unit -> 'a)
                   | Value of 'a | Exn of exn

# type 'a lazy_value = 'a lazy_state ref;;
 type 'a lazy_value = 'a lazy_state ref

# let lazy_from_fun f = ref (Delayed f);;
val lazy_from_fun : (unit -> 'a) ->
                    'a lazy_state ref = <fun>
```

Evaluación perezosa

Utilizaremos la función `force` para evaluar un elemento de tipo `lazy_value`. La lógica es simple:

- Si el elemento `lazy_value` ya es un valor calculado, se devuelve como resultado.
- Si el elemento `lazy_value` es una excepción, se propaga.
- En otro caso, se evalúa el elemento `lazy_value` para obtener `Value` o bien `Exc` y se actualiza la referencia.

```
# let force x_lazy =
  match !x_lazy with
    | Value x -> x
    | Exn e -> raise e
    | Delayed f ->  try
        let r = f() in  x_lazy := Value r; r
      with e -> x_lazy := Exn e; raise e;;
```

Evaluación perezosa

Considerando las siguientes definiciones:

```
# let millionth_fib_lazy =
      lazy_from_fun (fun() ->
      print_endline "calculate millionth fib";
      fibonacci 1000000);;

# let show_the_fib fib = function
  | true -> Some (force fib)
  | flase -> None;;

# let print_the_fib fib = function
  | true -> print_int (force fib)
  | flase -> ();;
```

Evaluación perezosa

Obtendríamos el siguiente resultado:

```
# show_the_fib millionth_fib_lazy true;;
calculate millionth fib
- : int option = Some (1884755131)

# print_the_fib millionth_fib_lazy true;;
1884755131 : unit = ()

# show_the_fib millionth_fib_lazy true;;
- : int option = Some (1884755131)

# print_the_fib millionth_fib_lazy true;;
- : int option = Some (1884755131)
```