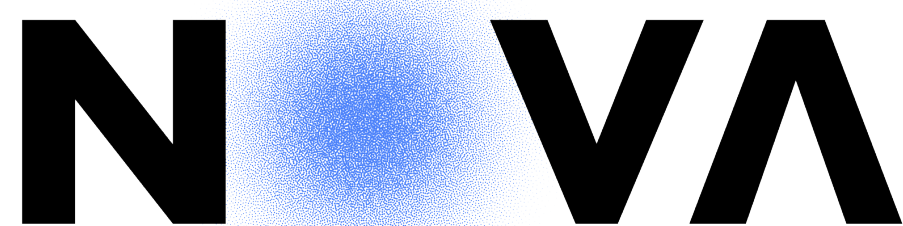


Linguagens e Ambientes de Programação (Aula Teórica 14)

LEI - Licenciatura em Engenharia Informática

João Costa Seco (joao.seco@fct.unl.pt)



NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Agenda

- Mutability
- Memoization

Mutability

- A linguagem OCaml não é uma linguagem *pura*, é possível programar com efeitos laterais.
- Exemplos de efeitos laterais: I/O (printing), leitura e escrita de ficheiros, variáveis de estado, bases de dados, comunicação em geral.
- Mutabilidade permite a implementação de estruturas de dados mais eficientes do que as puras (ex: hash table, doubly linked lists, cyclic graphs)
- A mutabilidade torna a programação mais difícil. Não é fácil raciocinar sobre a mudança de estado (sem enumerar todos os passos).

Refs!

- Um valor de tipo ref é como um apontador em C, ou uma referência para um objecto ou array em Java.
- As operações de criação e desreferenciação (na heap) são explícitas, como o malloc, ou o “*” em C.
- Criação e desreferenciação são diferentes da utilização de variáveis de estado (stack) nas linguagens imperativas (C, Java, etc.)

```
[1] let x = ref 0;;
    ✓ 0.0s
... val x : int ref = {contents = 0}

[2] !x ;;
    ✓ 0.0s
... - : int = 0

[3] x := !x + 1
    ✓ 0.0s
... - : unit = ()

▶ [4] !x ;;
    ✓ 0.0s
... - : int = 1
```

Igualdade “física”

- O operador de igualdade do OCaml (=) testa os valores pela sua estrutura.
- A função Stdlib.(==) testa se duas referências são a mesma. Interessante para algoritmos sobre estruturas de dados dinâmicas (references, arrays, byte sequences, records com campos mutáveis, etc.)

```
▷ ▾  
    let r1 = ref 42  
    let r2 = ref 42  
  
    let _ = assert (not (r1 = r2))  
    let _ = assert (r1 ≠ r2)  
    let _ = assert (!r1 = !r2)  
    let _ = assert (r1 = r2)  
[35] ✓ 0.0s  
... val r1 : int ref = {contents = 42}  
... val r2 : int ref = {contents = 42}  
... - : unit = ()  
... - : unit = ()  
... - : unit = ()  
... - : unit = ()
```

Aliasing

- Com referências teremos aliasing: “Os nomes da stack permitem mais que um caminho para uma mesma célula de memória.”
- Com aliasing o raciocínio fica ainda mais difícil. Para analisar a independência de dois segmentos de código (threads, caller/callee) é preciso eliminar/controlar o aliasing.
- O problema do aliasing em linguagens imperativas é “resolvido” em linguagens como o RUST.



```
let x = ref 42 ;;  
let y = ref 42 ;;  
let z = x ;;  
x := 43 ;;  
let w = !y + !z ;;
```

[7]

✓ 0.0s

```
... val x : int ref = {contents = 42}  
... val y : int ref = {contents = 42}  
... val z : int ref = {contents = 42}  
... - : unit = ()  
... val w : int = 85
```


Exemplo de um contador

- A função `next_val` retorna um valor diferente cada vez que é chamada. Tem efeitos laterais.
- A criação de uma variável tem que ser separada da função que incrementa.

```
let next_val_broken = fun () →
  let counter = ref 0 in
  incr counter;
  !counter

[14] ✓ 0.0s
... val next_val_broken : unit → int = <fun>

▷ next_val_broken ();;
next_val_broken ();;

[15] ✓ 0.0s
... - : int = 1
... - : int = 1
```

```
let counter = ref 0

let next_val =
  fun () →
    counter := !counter + 1;
    !counter

[10] ✓ 0.0s
... val counter : int ref = {contents = 0}
... val next_val : unit → int = <fun>

▷ next_val ();;
next_val ();;

[11] ✓ 0.0s
... - : int = 1
... - : int = 2
```

Exemplo de um contador

- A função `next_val` retorna um valor diferente cada vez que é chamada. Tem efeitos laterais.
- A criação de uma variável tem que ser separada da função que incrementa.

```
▷ ▾  
module Counter1 = Counter.Make();;  
Counter1.next_val ();;  
Counter1.next_val ();;  
module Counter2 = Counter.Make();;  
Counter2.next_val ();;  
[28] ✓ 0.0s
```

```
... module Counter1 :  
    sig type t = int ref val counter : int ref val next_val : unit → int end  
... - : int = 1  
... - : int = 2  
... module Counter2 :  
    sig type t = int ref val counter : int ref val next_val : unit → int end  
... - : int = 1
```

```
▷ ▾  
module Counter = struct  
  module Make() = struct  
    type t = int ref  
    let counter = ref 0  
    let next_val () =  
      counter := !counter + 1;  
      !counter  
  end  
end  
[25] ✓ 0.0s
```

```
... module Counter :  
  sig  
    module Make :  
      functor () →  
        sig  
          type t = int ref  
          val counter : int ref  
          val next_val : unit → int  
        end  
  end  
end
```


Recursão pela memória (Landin's knot)

- A implementação de recursão pode ser feita sem recurso à recursão nativa.
- Uma variável de estado (ref) pode guardar a continuação
- Bom para substituir e interceptar chamadas (ex: para implementar memoization)

```
▷ ▾  
let rec fact_rec n = if n = 0 then 1 else n * fact_rec (n - 1)  
  
let fact0 = ref (fun x → x + 0)  
  
let fact n = if n = 0 then 1 else n * !fact0 (n - 1);;  
fact0 := fact  
  
let _ = fact 5  
  
[19] ✓ 0.0s  
... val fact_rec : int → int = <fun>  
... val fact0 : (int → int) ref = {contents = <fun>}  
... val fact : int → int = <fun>  
... - : unit = ()  
... - : int = 120
```

Hashtbl

- O módulo Hashtbl tem uma interface genérica com uma função de hash built-in, e também tem um functor Make que permite a sua customização.

```
let t = Hashtbl.create 16;;
List.fold_left (fun () x → Hashtbl.add t x (string_of_int x)) () (List.init 16 (fun x → x));;
Hashtbl.mem t 5;;

[15] ✓ 0.0s

... val t : ('_weak3, '_weak4) Hashtbl.t = <abstr>

... - : unit = ()

... - : bool = true
```

Hashtbl

- O módulo Hashtbl tem uma interface genérica com uma função de hash built-in, e também tem um functor Make que permite a sua customização.

```
module PairHash = struct
  type t = string * int
  let equal (a1, b1) (a2, b2) = a1 = a2 && b1 = b2
  let hash (a, b) = Hashtbl.hash (a, b)
end
```

```
module PairHashtbl = Hashtbl.Make(PairHash)
```

[4] ✓ 0.0s

```
... - : bool = true
```

```
... module PairHash :
  sig
    type t = string * int
    val equal : 'a * 'b → 'a * 'b → bool
    val hash : 'a * 'b → int
  end

... module PairHashtbl :
  sig
    type key = PairHash.t
    type 'a t = 'a Hashtbl.Make(PairHash).t
    val create : int → 'a t
    val clear : 'a t → unit
    val reset : 'a t → unit
    val copy : 'a t → 'a t
    val add : 'a t → key → 'a → unit
    val remove : 'a t → key → unit
    val find : 'a t → key → 'a
    val find_opt : 'a t → key → 'a option
    val find_all : 'a t → key → 'a list
    val replace : 'a t → key → 'a → unit
    val mem : 'a t → key → bool
    val iter : (key → 'a → unit) → 'a t → unit
    val filter_map_inplace : (key → 'a → 'a option) → 'a t → unit
    val fold : (key → 'a → 'b → 'b) → 'a t → 'b → 'b
    val length : 'a t → int
    val stats : 'a t → Hashtbl.statistics
    val to_seq : 'a t → (key * 'a) Seq.t
    val to_seq_keys : 'a t → key Seq.t
    val to_seq_values : 'a t → 'a Seq.t
    val add_seq : 'a t → (key * 'a) Seq.t → unit
    val replace_seq : 'a t → (key * 'a) Seq.t → unit
    val of_seq : (key * 'a) Seq.t → 'a t
  end
```

Memoization

```
let rec fib n = if n < 2 then 1 else fib (n - 1) + fib (n - 2)
```

```
let _ = fib 45
```

[7] ✓ 38.2s

... val fib : int → int = <fun>

... - : int = 1836311903

Memoization

```
[7] ✓  
... val  
... - :
```

```
let fib_memo n =  
  let memo = Hashtbl.create 16 in  
  let rec fib_memo' n =  
    if n < 2 then 1  
    else  
      match Hashtbl.find_opt memo n with  
      | Some v → v  
      | None →  
        let v = fib_memo' (n - 1) + fib_memo' (n - 2) in  
        Hashtbl.add memo n v;  
        v  
  in  
  fib_memo' n  
  
let _ = fib_memo 45
```

```
[8] ✓ 0.0s  
... val fib_memo : int → int = <fun>  
... - : int = 1836311903
```

```
- 2)
```


General Memoization



```
let fib_0 = ref (fun x → x)
let rec fib_rec n = if n < 2 then 1 else (!fib_0) (n - 1) + (!fib_0) (n - 2)
let _ = fib_0 := fib_rec

let _ = fib_rec 45
```

[25] ✓ 40.2s

... val fib_0 : ('_weak24 → '_weak24) ref = {contents = <fun>}

... val fib_rec : int → int = <fun>

... - : unit = ()

... - : int = 1836311903

General Memoization

▷ ▾

[25] ✓

... val

... val [24]

... - :

... - :

```
let fib_hash = Hashtbl.create 16
let fib_mem =
  fun n →
    match Hashtbl.find_opt fib_hash n with
    | Some v → v
    | None → let v = fib_rec n in
             Hashtbl.add fib_hash n v;
             v
let _ = fib_0 := fib_mem
let _ = fib_rec 45
```

(n - 2)

```
[24] ✓ 0.0s
... val fib_hash : ('_weak22, '_weak23) Hashtbl.t = <abstr>
... - :
... val fib_mem : int → int = <fun>
... - :
... - : unit = ()
... - : int = 1836311903
```