Programmation Fonctionnelle

Adrien Durier

03 octobre 2023

DM & TP noté

TP noté le mercredi 11 octobre
Présence obligatoire !

Modalités sur *ecampus*.

Mini-DM
Pour mardi prochain (10/10/2023)

Simplement rendre le fichier `.ml` complété, avec les réponses aux questions en commentaire.

Mini-DM

On utilise le type 'option' pour représenter des fonctions partielles (polymorphes) :

```
let partiel_pairs : int -> int option = fun n ->
 if n mod 2 = 0 then Some (n/2) else None
```

Cette fonction n'est définie que sur les entiers pairs : elle retourne alors $n/2$. Sur les nombres impairs, elle n'est pas définie. Test :

```
let _ = partiel_pairs 4 (* Some 2 *)
let _ = partiel_pairs 5 (* None *)
```

Mini-DM

On utilise le type 'option' pour représenter des fonctions partielles (polymorphes) :

```
let partiel_pairs : int -> int option = fun n ->
 if n mod 2 = 0 then Some (n/2) else None
```

Fonction qui met à jour une fonction partielle sur la valeur x

```
let update_partial (f : 'a -> 'b option) (x : 'a) (n : 'b)
        : 'a -> 'b option =
  fun y -> if x = y then Some n else (f y)
```

Attention !
Pas d'effets de bord...

Il faut utiliser la fonction retournée !

Curryfication

```
let add x y = x + y
val add : int -> int -> int = <fun>
```

```
let add' t = fst t + snd t
let add' (x, y) = x + y
val add' : int * int -> int = <fun>
```

```
let curry f x y = f (x, y)
let uncurry f (x, y) = f x y
```

```
val curry : ('a * 'b -> 'c)
 -> 'a -> 'b -> 'c = <fun>
```

Haskell Curry

(1900 - 1982 – comme le langage Haskell !)

```
let uncurried_add = uncurry add
val uncurried_add : int * int -> int = <fun>
```

L'évaluation Paresseuse

Permet de définir des **objets infinis** (arbres, listes. . .)

- Haskell, Scala . . .

Et de ne calculer les valeurs des éléments uniquement quand c'est nécessaire !

Evaluation Paresseuse en OCaml

```
type 'a glaçon =
| Gelé of unit -> 'a
| Connu of 'a
```

```
type 'a lazy_list =
| Nil
| Cons of 'a cellule
and 'a cellule = { hd : 'a; mutable tl : 'a lazy_list glaçon};;
```

Evaluation Paresseuse en OCaml

```
let force cellule =
 let glaçon = cellule.tl in
  match glaçon with
   | Connu val -> val
   | Gelé g ->
     let val = g () in
     cellule.tl <- Connu val;
     val
```

```
let rec lazy_map f = function
 | Nil -> Nil
 | Cons ({hd = x; } as cellule) ->
    Cons {
          hd = f x;
          tl = Gelé (function () -> lazy_map f (force cellule))
         }
lazy_map : ('a -> 'b) -> 'a lazy_list -> 'b lazy_list = <fun>
```

Evaluation Paresseuse en OCaml

La liste de tous les entiers :

```
let rec nat = Cons {
                    hd = 0;
                    tl = Gelé (fun () ->
                          lazy_map (fun n -> n + 1) nat)
                   }
nat : int lazy_list = Cons {hd=0; tl=Gelé <fun>}
```

```
let rec lazy_do_list f n = function
 | Nil -> ()
 | Cons ({hd = x; } as cellule) ->
    if n > 0
    then begin
           f x;
           lazy_do_list f (n - 1) (force cellule)
         end
lazy do list : ('a -> 'b) -> int -> 'a lazy_list -> unit = <fun>
```

Evaluation Paresseuse en OCaml

```
lazy_do_list print_int 3 nat;;
012- : unit = ()

nat;;
- : int lazy_list =
```