

LIFPF – Programmation fonctionnelle

TD4 – Fonctions d'ordre supérieur

Licence informatique UCBL – Printemps 2025

Exercice 1 : Décorateurs

Un *décorateur* est une fonction d qui va prendre (entre autres) une fonction f en argument et renvoyer une nouvelle fonction dont le comportement sera une "modification" de celui de f . Pour les décorateurs suivants, donner le type puis le code du décorateur. Dans la description de ces décorateurs, la fonction passée au décorateur sera nommée f et la fonction obtenue g .

1. *lift_option* : g pourra prendre une `option` en argument. Si cette option est `None`, alors le résultat sera également `None`, sinon cela sera `Some` avec le résultat de l'application de f à la valeur de l'option.
2. *default* : f renvoie une option et g renvoie une valeur par défaut r si f renvoie `None`, ou la valeur de l'option sinon. r doit être un paramètre de *default*.
3. *except* : g renverra le même résultat que f sauf pour une valeur particulière a pour laquelle le résultat sera r . *except* devra être paramétrée par a et r .
4. *power* : g applique f n fois à son argument (c'est-à-dire $g = f^n$).

Exercice 2 : Portée de variable, fermetures et applications partielles

Soit le code suivant :

```
let f =  
  let x = 5 in  
  fun y -> x + y  
in  
f 2
```

1. Ce code s'évalue-t-il sans erreur ? Si oui quel est le résultat ?
2. Détailler l'évaluation de l'expression en la traduisant auparavant en λ -calcul. On pourra traduire `let a = e_1 in e_2` par $(\lambda a.e_2 \ e_1)$. On effectuera cette évaluation avec une stratégie avide en s'interdisant de réduire à l'intérieur d'un λ (ce qui correspond au mode d'évaluation de OCaml). Indiquer quelle(s) réduction(s) correspond(ent) à un calcul de fermeture.
3. On considère maintenant le code suivant :

```
let f x =  
  let y = g x in  
  fun z -> y + z  
  
let f' x z =  
  let y = g x in  
  y + z
```

```
let v =
  let h = f 3 in
  [ h 10; h 12; h 14; h 16 ]
```

```
let v' =
  let h' = f' 3 in
  [ h' 10; h' 12; h' 14; h' 16 ]
```

En supposant qu'un appel à `g` prenne 1 min de calcul, combien de temps prend le calcul de `v`? Combien de temps prend le calcul de `v'`?

4. On souhaite, en utilisant `List.map`, implémenter une fonction qui prend une liste de liste d'`int` et renvoie la liste de liste d'`int` contenant des listes où les int de départ ont été élevés au carré. Aurait-on pu facilement coder cette fonction si `List.map` avait pris en une fois une paire (fonction de transformation, liste à transformer)?

Exercice 3 : Parcours génériques d'arbres

On souhaite implémenter une structure d'arbre binaire de recherche pour implémenter une structure d'association clé-valeur, une clé pouvant être associée à plusieurs valeurs. Les données stockées aux nœuds seront donc de la forme (clé, liste de valeurs).

1. Définir le type de cette structure.
2. Définir la fonction `insere` qui ajoute une valeur en fonction de la clé et d'une fonction de comparaison des clés. On utilisera le type `type cmp_result = Lt | Eq | Gt` (plus petit, égal, supérieur) pour le résultat des comparaisons.
3. Définir une fonction `map_valeurs` qui prend en argument une fonction et transforme toutes les valeurs de l'arbre en utilisant cette fonction. On utilisera `List.map` pour transformer les listes le cas échéant.
4. Définir une fonction `fold_valeurs` qui prend en argument une fonction `f` et une valeur initiale `v`. `f` effectuera un calcul sur un accumulateur et une valeur pour renvoyer une nouvelle valeur d'accumulateur. `fold_valeurs` renverra la valeur de $f(f(\dots(f\ v\ x_n)\dots)x_2)x_1$ si $\{x_1, x_2, \dots, x_n\}$ sont les valeurs stockées dans l'arbre. On utilisera `List.fold_left` le cas échéant. Est-il possible d'avoir plusieurs implémentations différentes de cette fonction produisant des résultats différents?
5. Quelles difficultés vont se poser si on souhaite implémenter une fonction `map_cles` qui transforme les clés? Proposer une implémentation de `map_cles` qui respecte la contrainte que le résultat doit être un arbre binaire de recherche.
6. Définir une fonction `fold_arbre` qui va travailler sur les données des nœuds et pas simplement sur les valeurs. La fonction passée en argument à `fold_arbre` prendra deux valeurs d'accumulateur (pourquoi?). Recoder `map_valeur` à partir de cette fonction.

Corrections

Solution de l'exercice 1

```
(* 1 *)
let lift_option (f : 'a -> 'b) (o : 'a option) : 'b option =
  match o with None -> None | Some x -> Some (f x)

(* existe dans le module Option de la Stdlib *)
let lift_option = Option.map

(* 2 *)
let default (r : 'b) (f : 'a -> 'b option) (x : 'a) : 'b =
  match f x with None -> r | Some y -> y

(* existe dans le module Option de la Stdlib,
   mais avec des paramètres nommés et dans un autre ordre *)
let default f r x = Option.value (f x) ~default:r

(* 3 *)
let except (a : 'a) (r : 'b) (f : 'a -> 'b) (x : 'a) : 'b =
  if a = x then r else f x

(* 4 *)
let rec power (n : int) (f : 'a -> 'a) (x : 'a) : 'a =
  if n <= 0 then x else f (power (n - 1) f x)
(* en fait c'est un fold sur les entiers de Peano,
   avec succ = f et zero = x dont les paramètres
   sont réordonnés, on aurait écrit plutôt :

   int_rec (s : 'b -> 'b) (z : 'b) (n : int) : 'b *)
```

Solution de l'exercice 2

1. Le code s'évalue sans erreur et donne 7 comme résultat.

2. $(\lambda f.(f\ 2)\ (\lambda x.\lambda y.(x + y)\ 5))$
 $\xrightarrow{x} (\lambda f.(f\ 2)\ \lambda y.(5 + y))$
 $\xrightarrow{f} (\lambda y.(5 + y)\ 2)$
 $\xrightarrow{y} (5 + 2) \xrightarrow{+} 7$

C'est lors de la première réduction sur x que la fonction λy est produite et c'est à ce moment là que OCaml va effectuer la capture de la valeur de x (qui dans le λ -calcul est faite par le jeu de la substitution).

3. On suppose que hors du temps d'exécution de g , les calculs sont quasi instantanés.

Dans le calcul de v , l'appel à g se fait au moment du calcul de h . g est appelée une fois et la valeur est utilisée dans les différents appels à h . On a donc un peu plus d'une minute de temps de calcul pour v .

Dans le calcul de v' , l'appel à g se fait lors des appels à h' car le `let` est dans la définition de la fonction qui prend z en argument (on rappelle que `let f' x z = ...` est équivalent à `let f' = fun x -> fun z -> ...`). g est donc appelée 4 fois et le calcul de v' prend un peu plus de 4 minutes.

Pour se convaincre, on peut définir une fonction avec un effet de bord et compter le nombre d'appels comme `let g x = let _ = print_endline "g" in x + 1`.

4. On peut par exemple utiliser le code suivant :

```
let tous_au_carre : int list list -> int list list =  
  List.map (List.map (fun x -> x * x))
```

La concision de ce code vient du fait qu'on peut faire des applications partielles de `List.map`. Si `List.map` prenait tous ses arguments en une fois, il aurait fallu introduire un `fun` pour gérer explicitement le passage de la liste en paramètre pour chacun des usages de `List.map`.

Solution de l'exercice 3

```
(* 1 *)  
type ('a, 'b) arbre =  
  | Vide  
  | Noeud of 'a * 'b list * ('a, 'b) arbre * ('a, 'b) arbre  
  
(* 2 *)  
type cmp_result = Lt | Eq | Gt  
  
let rec insere (cmp : 'a -> 'a -> cmp_result) (k : 'a) (v : 'b)  
  (a : ('a, 'b) arbre) : ('a, 'b) arbre =  
  match a with  
  | Vide -> Noeud (k, [ v ], Vide, Vide)  
  | Noeud (k', vs', fg, fd) -> (  
    match cmp k k' with  
    | Lt -> Noeud (k', vs', insere cmp k v fg, fd)  
    | Gt -> Noeud (k', vs', fg, insere cmp k v fd)  
    | Eq -> Noeud (k', v :: vs', fg, fd))  
  
(* 3 *)  
let rec map_valeurs (f : 'b -> 'c) (a : ('a, 'b) arbre) :  
  ('a, 'c) arbre =  
  match a with  
  | Vide -> Vide  
  | Noeud (k, vs, fg, fd) ->  
    Noeud (k, List.map f vs, map_valeurs f fg, map_valeurs f fd)  
  
(* 4 *)  
let rec fold_valeurs (f : 'c -> 'b -> 'c) (v : 'c)  
  (a : ('a, 'b) arbre) : 'c =  
  match a with  
  | Vide -> v  
  | Noeud (_, vs, fg, fd) ->  
    fold_valeurs f  
      (List.fold_left f (fold_valeurs f v fd) vs)  
      fg  
  
(* On peut faire varier l'ordre des appels a fold dans le cas  
   du noeud.  
   Si f n'est pas commutative (par exemple avec la concatenation
```

de string) le resultat sera different. *)

(* 5 *)

(* Si on transforme naivement les cles les problemes suivants
peuvent se poser:
- les cles resultat ne sont pas forcement dans le meme ordre
- certaines cles peuvent avoir la meme transformation

Un moyen d'éviter ces problemes est de reconstruire un
arbre resultat par insertions, mais cela a un cout en
termes de calcul *)

```
let map_cles (cmp : 'c -> 'c -> cmp_result) (f : 'a -> 'c)
  (a : ('a, 'b) arbre) : ('c, 'b) arbre =
  let rec insere_map a' a'' =
    match a' with
    | Vide -> a''
    | Noeud (k, vs, fg, fd) ->
      let k' = f k in
      List.fold_left
        (fun a''' v -> insere cmp k' v a''')
        (insere_map fg (insere_map fd a''))
        vs
  in
  insere_map a Vide
```

(* 6 *)

```
let rec fold_arbre (f : 'c -> 'c -> 'a -> 'b list -> 'c) (v : 'c)
  (a : ('a, 'b) arbre) : 'c =
  match a with
  | Vide -> v
  | Noeud (k, vs, fg, fd) ->
    f (fold_arbre f v fg) (fold_arbre f v fd) k vs

let map_valeur' (f : 'b -> 'c) (a : ('a, 'b) arbre) :
  ('a, 'c) arbre =
  fold_arbre
    (fun fg fd k vs -> Noeud (k, List.map f vs, fg, fd))
    Vide a
```