

LIFPF – Programmation fonctionnelle

TD2 – λ -calcul et OCaml

Licence informatique UCBL – Printemps 2025

Exercice 1 : λ -calcul et OCaml

En OCaml, on peut utiliser la notation `fun x -> ...` pour décrire une fonction anonyme qui prend en argument `x` et renvoie ce qui est calculé par l'expression se trouvant après la flèche. Cette notation est donc un bon moyen de traduire la notation $\lambda x. \dots$. Par exemple, $\lambda x. x$ se traduit par `fun x -> x`, $\lambda x. \lambda y. (x + y)$ se traduit par `fun x -> fun y -> x + y` et $\lambda x. \lambda y. (x y)$ se traduit par `fun x -> fun y -> x y`.

1. Typé, puis traduire les termes suivants du λ -calcul vers OCaml.
 - a. $\lambda x. (x + 2)$
 - b. $\lambda x. \lambda y. ((2 * x) - y)$
 - c. $\lambda x. ((x 2) + (x 3))$
 - d. $\lambda x. (x \lambda y. (y + 3))$

2. Soit le code OCaml suivant :

```
(fun u -> fun y -> u (fun v -> v)) (fun x -> x y) 3
```

Ce code produit l'erreur suivante : `Error: Unbound value y.`

- a. Traduire ce code en λ -calcul, puis donner les variables libres du terme obtenu et faire le lien avec l'erreur ci-dessus.
- b. β -réduire le terme en prêtant une attention particulière aux renommages nécessaires. Que peut-on en déduire vis-à-vis des différentes occurrences de `y` dans le code OCaml de départ ?

Exercice 2 : Codage d'arguments multiples

On ajoute au λ -calcul le constructeur de type \times . On considère de plus les fonction prédéfinies suivantes, avec leur type et les règles de réduction dédiées :

- $\text{paire} : \tau \rightarrow \tau' \rightarrow (\tau \times \tau')$
- $\text{fst} : (\tau \times \tau') \rightarrow \tau$
- $\text{snd} : (\tau \times \tau') \rightarrow \tau'$
- $(\text{fst} (\text{paire } a \ b)) \rightsquigarrow a$
- $(\text{snd} (\text{paire } a \ b)) \rightsquigarrow b$

1. Montrer que $(\lambda p. ((\text{fst } p) + (\text{snd } p)) ((\text{paire } x) \ y))$ se réduit sur le même terme que $((\lambda p_1. \lambda p_2. (p_1 + p_2) \ x) \ y)$. Discuter sur les similitudes et les différences entre ces deux stratégies de passage d'arguments multiples dans une même fonction.
2. Coder `paire`, `fst` et `snd` en OCaml en utilisant un `match` pour les deux derniers.
3. (Pour aller plus loin) On se donne le codage suivant en λ -calcul :

— $\text{paire} = \lambda x. \lambda y. \lambda s. ((s \ x) \ y)$

- $fst = \lambda p.(p \ \lambda u.\lambda w.u)$
- $snd = \lambda p.(p \ \lambda u.\lambda w.w)$

Montrer que ce codage correspond aux règles de réductions données au début de l'exercice.

Exercice 3 : Filtrage de motif OCaml

On considère les types OCaml définis par le code suivant :

```
type couleur = Pique | Coeur | Carreau | Trefle
type valeur = As | Roi | Dame | Valet | Nombre of int
type carte = Normale of couleur * valeur | Joker
```

1. Donner un exemple de carte qui n'existe pas mais qui est représentable avec ces types.
2. Écrire une fonction `valeur_blackjack` de type `carte -> int` qui donne la valeur d'une carte au Blackjack, à savoir : 11 pour les as, 10 pour chaque figure, autant que leur nombre pour les cartes numérotées et enfin 0 pour les jokers ou les cartes qui n'existent pas. On fera deux versions de cette fonction : une avec des `match` imbriqués pour gérer les as, les cartes inexistantes et les figures et une qui utilise le filtrage de motif en profondeur, c'est-à-dire qui filtre sur des constructeurs imbriqués. Dans les deux cas, on pourra utiliser un `if` pour écarter les cartes inexistantes. Pour la deuxième version, proposer une valeur correspondant à chaque cas du `match`.
3. Écrire une fonction `valeur_belote` de type `carte -> couleur -> int` qui donne la valeur d'une carte à la belote en fonction de la couleur d'atout. Par défaut l'as vaut 11, le 10 vaut 10, le roi 4, la dame 3, le valet 2 et les autres cartes 0. Cependant si la couleur de la carte est la couleur d'atout, le valet vaut 20 et le 9 vaut 14 (les autres cartes gardent la même valeur). On codera la fonction avec un seul `match`, mais avec des `if` quand cela est nécessaire. Expliquer pourquoi on ne peut pas utiliser le paramètre de la couleur d'atout directement dans les cas du `match`.

Corrections

Solution de l'exercice 1

1.
 - a. Type : `number → number`, OCaml : `fun x -> x+2`
 - b. Type : `number → number → number`, OCaml : `fun x -> fun y -> 2 * x - y`
 - c. Type : `(number → number) → number`, OCaml : `fun x -> x 2 + x 3`
 - d. Type : `((number → number) → τ) → τ`, OCaml : `fun x -> x (fun y -> y+3)`
2.
 - a. En λ -calcul : $((\lambda u. \lambda y. (u \lambda v. v) \lambda x. (x y)) 3)$. y est la seule variable libre, elle n'est pas définie dans l'expression, ce qui veut dire qu'en OCaml il faut qu'elle soit définie globalement auparavant.
 - b. Il faut commencer par réduire λu , mais le y libre dans $\lambda x. (x y)$ est en conflit avec le λy . On commence donc par un renommage du λy en utilisant w , puis on réduit jusqu'au moment où on est bloqué car on a pas la valeur de y .

$$\begin{aligned} & ((\lambda u. \lambda y. (u \lambda v. v) \lambda x. (x y)) 3) \\ \equiv_{\alpha} & ((\lambda u. \lambda w. (u \lambda v. v) \lambda x. (x y)) 3) \\ \xrightarrow{u} & (\lambda w. (\lambda x. (x y) \lambda v. v) 3) \\ \xrightarrow{w} & (\lambda x. (x y) \lambda v. v) \\ \xrightarrow{x} & (\lambda v. v y) \xrightarrow{v} y \end{aligned}$$

En λ -calcul, cela ne pose pas de problème de s'arrêter sur y , mais en OCaml, on chercherait à récupérer la valeur de y ce qui provoquerait une erreur, d'où le fait qu'OCaml refuse dès le départ d'exécuter ce code.

Solution de l'exercice 2

1. D'un côté :

$$\begin{aligned} & (\lambda p. ((fst p) + (snd p)) ((paire x) y)) \\ \xrightarrow{p} & ((fst ((paire x) y)) + (snd ((paire x) y))) \\ \xrightarrow{fst} & (x + (snd ((paire x) y))) \\ \xrightarrow{snd} & (x + y) \end{aligned}$$

de l'autre :

$$\begin{aligned} & ((\lambda p_1. \lambda p_2. (p_1 + p_2) x) y) \\ \xrightarrow{p_1} & (\lambda p_2. (x + p_2) y) \\ \xrightarrow{p_2} & (x + y) \end{aligned}$$

On a d'un côté la paire (x, y) passée en une fois via p à la fonction et de l'autre une fonction qui prend d'abord le premier composant de la paire puis renvoie une fonction qui prend le second composant et renvoie le calcul voulu (ici la somme). Le premier mode permet de gérer un seul argument avec les deux valeurs, mais force à l'extraire avec *fst* et *snd* plus tard. C'est ce mode qui est utilisé par défaut dans la plupart des langages impératifs comme C, Java, Python, etc. Le second mode nécessite de fabriquer une deuxième fonction pour gérer le deuxième argument, mais, contrairement à la première version, on peut décider de passer chaque argument indépendamment car $(\lambda p_2. (x + p_2) y)$ est une valeur que l'on peut utiliser elle-même ailleurs avant à un moment de lui passer la deuxième valeur. Par exemple on peut écrire : $(\lambda u. ((u 2) + (u 3)) (\lambda p_1. \lambda p_2. (p_1 + p_2) x))$ dans laquelle $(\lambda p_2. (x + p_2) y)$ viendra remplacer u à deux endroits avant de se voir passer son deuxième argument.

```

2. (* paire est simplement codee avec la notation (,) *)
   let paire x y = (x, y);;

   (* on extrait le premier composant de la paire avec le match *)
   let fst p =
       match p with
       | (x, _) -> x;;

   (* on extrait le deuxieme composant de la paire avec le match *)
   let snd p =
       match p with
       | (_, y) -> y;;

```

3. On vérifie simplement que les réductions dédiées se font naturellement avec le codage :

```

(fst (paire a b))
= (λp.(p λu.λw.u) ((λx.λy.λs.((s x) y) a) b))
 $\xrightarrow{x}$  (λp.(p λu.λw.u) (λy.λs.((s a) y) b))
 $\xrightarrow{y}$  (λp.(p λu.λw.u) λs.((s a) b))
 $\xrightarrow{p}$  (λs.((s a) b) λu.λw.u)
 $\xrightarrow{s}$  ((λu.λw.u a) b)
 $\xrightarrow{u}$  (λw.a b)
 $\xrightarrow{w}$  a

```

et

```

(snd (paire a b))
= (λp.(p λu.λw.w) ((λx.λy.λs.((s x) y) a) b))
 $\xrightarrow{x}$  (λp.(p λu.λw.w) (λy.λs.((s a) y) b))
 $\xrightarrow{y}$  (λp.(p λu.λw.w) λs.((s a) b))
 $\xrightarrow{p}$  (λs.((s a) b) λu.λw.w)
 $\xrightarrow{s}$  ((λu.λw.w a) b)
 $\xrightarrow{u}$  (λw.w b)
 $\xrightarrow{w}$  b

```

Au final on voit que le λ -calcul est suffisamment expressif pour coder les paires sans ajouter de construction native particulière.

Solution de l'exercice 3

```

(* correction exercice 3 *)
type couleur = Pique | Coeur | Carreau | Trefle
type valeur = As | Roi | Dame | Valet | Nombre of int
type carte = Normale of couleur * valeur | Joker;;

(* 3.1 *)
Normale (Pique, Nombre 42)

(* Ceci n'est pas une vraie carte *)

(* Exercice 3.2 version match imbriquées *)

```

```

(** Cette fonction calcule la valeur d'une carte au blackjack
    @param c la carte
    @return la valeur de la carte *)
let valeur_blackjack (c : carte) : int =
  match c with
  | Joker -> 0
  | Normale (_, v) -> (
      match v with
      | As -> 11
      | Nombre n -> if n > 0 && n <= 10 then n else 0
      | _ -> 10)

```

```

(* Exercice 3.2 version match profond *)

```

```

(** Cette fonction calcule la valeur d'une carte au blackjack
    @param c la carte
    @return la valeur de la carte *)
let valeur_blackjack (c : carte) : int =
  match c with
  | Normale (_, As) -> 11
  | Normale (_, Nombre n) -> if n > 0 && n <= 10 then n else 0
  | Normale (_, _) -> 10
  | _ -> 0
;;

```

```

valeur_blackjack (Normale (Pique, As));;
valeur_blackjack (Normale (Pique, Nombre 7));;
valeur_blackjack (Normale (Pique, Dame));;
valeur_blackjack Joker

```

```

(* Exercice 3.3 *)

```

```

(** Cette fonction donne la valeur d'une carte a la belote en fonction de la
    couleur d'atout.

```

```

    @param c la carte
    @param a la couleur d'atout
    @return la valeur de la carte, ou 0 si ce n'est pas une carte *)
let valeur_belote c a =
  match c with
  | Normale (_, As) -> 11
  | Normale (_, Roi) -> 4
  | Normale (_, Dame) -> 3
  | Normale (co, Valet) -> if a = co then 20 else 2
  | Normale (_, Nombre 10) -> 10
  | Normale (co, Nombre 9) -> if a = co then 14 else 0
  | _ -> 0
(* on ne peut mettre a dans les cas du match car cela
   introduirait une nouvelle variable a, alors qu'on
   veut en fait comparer la couleur de la carte a la
   valeur de a. *)

```