

Exercice 1 [Typage]

Que répond l'interpréteur Caml aux questions suivantes ?

1. `let f g x y = g (x, y);;`
`val f : (('a * 'b) → 'c) → 'a → 'b → 'c = <fun>`
2. `let j x y = x y y;;`
`val j : ('a → 'a → 'b) → 'a → 'b = <fun>`
3. `let h x y = x x y;;`
Type error.
4. `let jj f (x, y) = f (f x x) y;;`
`val jj : ('a → 'a → 'a) → ('a * 'a) → 'a = <fun>`

Exercice 2 [Démonstration de propriétés]

1. Définir la fonction de composition de fonctions

```
val compose : ('a → 'b) → ('c → 'a) → ('c → 'b) = <fun>
  let compose f g = (fun x → f (g x));;
```

2. Démontrer que pour toutes fonctions $f: 'a \rightarrow 'b$, $g: 'c \rightarrow 'a$ et toute liste $l: 'a \text{ list}$, on a :

$$\text{map } f \text{ (map } g \text{ l)} = \text{map (compose } f \text{ g) l}$$

Cas de base : $l = []$

$$\begin{aligned} \text{map } f \text{ (map } g \text{ [])} &= \text{map } f \text{ []} = [] \\ \text{map (compose } f \text{ g) []} &= [] \end{aligned}$$

Donc le cas de base est **vérifié ✓**.

Cas inductif : $l = t::q$ et on suppose que $\text{map } f \text{ (map } g \text{ q)} = \text{map (compose } f \text{ g) q}$

$$\begin{aligned} \text{map } f \text{ (map } g \text{ l)} &= \text{map } f \text{ (map } g \text{ t)::q)} \\ &= \text{map } f \text{ ((g t)::(map } g \text{ q))} \\ &= (f \text{ g t)::(map } f \text{ (map } g \text{ q))} \\ &= ((\text{compose } f \text{ g) t)::(map (compose } f \text{ g) q)} \\ &= \text{map (compose } f \text{ g) t)::q} = \text{map (compose } f \text{ g) l} \end{aligned}$$

Donc le cas inductif est **vérifié ✓**.

Exercice 4

1. Nous souhaitons inventer la notion d'associations. Une association sera une structure permettant d'associer des valeurs à des données en entrée. Afin de simplifier le problème, il existera un type générique des associations du type `int` vers un type `'a`. Nous souhaitons avoir une structure permettant les opérations suivantes :

- association vide (aucune valeur associée)
- test de vacuité de la structure
- ajout d'une association de la clef `k` à la valeur `v` dans une structure `m`
- modification de la valeur associée à une clef
- recherche de la valeur associée à une clef (si elle existe)
- suppression d'une clef
- `fold` sur l'association entière prenant en compte les clefs et les valeurs

Proposer une interface pour un module prenant en compte les clefs et les valeurs.

```
module type I_assoc = sig
  type 'a assoc
  exception Already_in
  exception Not_found
  val empty: 'a assoc
  val is_empty: 'a assoc → bool
  val add: 'a assoc → int → 'a → 'a assoc (* raises Already_in *)
  val edit: 'a assoc → int → 'a → 'a assoc (* raises Not_found *)
  val del: 'a assoc → int → ('a assoc * 'a)
  val fold: (int → 'a → 'b → 'b) → 'a assoc → 'b
  val find: 'a assoc → int → 'a
end;;
```

2. À l'aide de vos connaissances, proposer une ou deux solutions rapides (pas besoin de donner le code) de cette structure.

```
type 'a assoc = int list * 'a list;;
type 'a assoc = (int * 'a) list;;
type 'a assoc = (int * 'a) avl;;
```