Création de types en Caml

Cours – TP

I Types alias.

I.1 Exemples

C'est un type existant mais auquel on attribue un nouveau nom avec la syntaxe `type ... = ...`

Exemple 1 On peut définir le type `mot` des listes de caractères :

```
type mot = char list;;
```

On peut imposer le type d'une fonction définie sur ou à valeurs dans un type alias.
Un exemple avec la fonction qui calcule le miroir d'un mot :

```
let miroir (m : mot) : mot = List.rev m;;
```

Un type peut être polymorphe, c'est-à-dire être paramétré par un ou plusieurs autres types.

Exemple 2 On peut définir le type `'a biglist` des listes de listes d'éléments de type 'a :

```
type 'a biglist = 'a list list;;
```

I.2 Travaux pratiques

Exercice 1. Écrire les fonctions `string_of_mot : mot→ string` et `mot_of_string : string→ mot` réalisant les conversions d'un mot en chaine de caractère et inversement, en forcant leur types.

```
type mot = char list;;

let rec mot_of_string (m: string): mot = if m = ""
    then []
    else m.[0] :: (mot_of_string (String.sub m 1 (String.length m - 1)))

let mot_of_string__terminale m: mot =
    let rec aux i l = match i with
    | -1 -> l
    | _ -> aux (i+1) (m.[i]::l) in
    aux (String.length m-1) []

let rec string_of_mot (m: mot): string = match m with
    | [] -> ""
    | h::t -> (String.make 1 h) ^ (string_of_mot t);;

let string_of_mot__terminale (m: mot): string =
    let rec aux (m: mot) (_acc: string) = match m with
    | [] -> _acc
    | h::t -> aux t (_acc ^ (String.make 1 h)) in
    aux m "";;
```

II Types somme

II.1 Exemples

Le mot "somme" dans ce contexte désigne une réunion disjointe, et se note |. Chaque terme de la réunion disjointe est introduit par un *constructeur* (qui peut avoir 0, 1 ou plusieurs arguments) dont le nom commence par une majuscule. Le *pattern-matching* s'applique précisément aux types somme (on filtre suivant le constructeur utilisé pour définir l'élément ayant le type somme).

Exemple 3 Le type `couleur` suivant a trois constructeurs qui n'ont aucun argument.

```
type couleur = Bleu | Rouge | Vert;;
```

Exemple 4 Le type `nombre` suivant a deux constructeurs qui ont chacun un argument.

```
type nombre = Entier of int | Flottant of float;;
```

II.2 Travaux pratiques

Exercice 2. Écrire une fonction `somme : nombre→ nombre→ nombre` pertinente

```
type nombre = Entier of int | Flottant of float

let (+) (a: nombre) (b: nombre): nombre = match a, b with
    | Entier a,   Entier b   -> Entier (a + b)
    | Flottant a, Flottant b -> Flottant (a +. b)
    | Entier a,   Flottant b -> Flottant ((float_of_int a) +. b)
    | Flottant a, Entier b   -> Flottant (a +. (float_of_int b))
```

III Types produits et enregistrements

III.1 Exemples

Types produits :
Si `'a` et `'b` sont deux types alors `'a*'b` est le type des couples `(x,y)` avec `x` de type `'a` et `y` de type `'b`.

Remarque 1

On peut définir de même de types produits avec $n \geqslant 3$ facteurs plutôt que 2.

Exemple 5 On peut définir le type `complexe` comme un alias du type `float * float`

Types enregistements :
C'est le même principe mais en permettant de *nommer* les différents éléments du couple (ou du n-uplet). Ainsi pour les complexes, il est plus agéable de pouvoir clairement spécifier ce qu'est la partie réelle et ce qu'est la partie imaginaire.

Exemple 6

```
type complexe = {re : float ; im : float};;
```

III.2 Travaux pratiques

Exercice 3. Définir le nombre complexe **i**, et les fonctions somme et produit sur les complexes.

```
type complexe = {re: float; im: float}

let i = { re = 0. ; im = 1. }

let ( + ) (a: complexe) (b: complexe) =
    { re = a.re +. b.re; im = a.im +. b.im }

let ( * ) (a: complexe) (b: complexe) =
    {
        re = a.re *. b.re -. a.im *. b.im;
        im = a.re *. b.im +. a.im *. b.re
    }

(* TODO: exponentiation rapide *)
let ( ^ ) (a: complexe) (b: int) =
    {
        re =
    }
```

III.3 Mutabilité

Par défaut, un type enregistement est un type *immuable*. On ne peut pas modifier les champs des enregistements d'un même élément après l'avoir défini, on peut seulement créer un nouvel élément avec des champs différents (de ce point de vue, les types enregistrements se comportent comme les listes et pas comme les tableaux).

Il est néanmoins possible d'y remédier en spécifiant que les champs concernés peuvent être mutables.

Exemple 7

```
type cplx = {mutable re : float ; mutable im : float};;

let z = {re = 2.0; im = 5.0}
z.re <- 3.0
```

IV Types inductifs

Dans le contexte des types, le mot inductif signifie récursif.

IV.1 L'exemple-type

On connaît déjà un type inductif : le type (polymorphe) `'a list`. Redéfinissons-le !

Exemple 8 Le type `'a liste` :

```
type 'a liste = Vide | Cons of 'a*('a liste);;
```

IV.2 Travaux pratiques

Exercice 4. Écrire une fonction qui convertit une `'a liste` en 'a list.

```
type 'a liste = Vide | Cons of ('a * 'a liste)

let rec list_of_liste l = match l with
    | Vide -> []
    | Cons (h, t) -> h::list_of_liste t

let rec liste_of_list l = match l with
    | [] -> Vide
    | h::t -> Cons (h, liste_of_list t)

let rec longueur (l: 'a liste): int = match l with
    | Vide -> 0
    | Cons (h, t) -> 1 + longueur t
```

Exercice 5. Proposer d'autres types inductifs.

```
type ('k, 'v) dict = Vide | Union of ('k * 'v) * (('k, 'v) dict)

let nombre_de_plus = Union (
        ("El Baki", 18), Union(
        ("Barriac", 19), Vide)
);;
```