

Colle 6 : Construction de types

Le langage OCAML permet de définir soit même des types. On va alors découvrir la syntaxe pour écrire un type dit "produit" (enregistrement dans le vocabulaire du langage OCAML), un type "somme" (distinctions de cas) et des alias.

1 UN PREMIER EXEMPLE DE TYPE ENREGISTREMENT

```
type couleur = {rouge : float ; bleu : float ; jaune : float}
```

Par exemple on peut définir :

```
let orange = {rouge=50.0;bleu=0.0;j jaune=50.0};;
```

La commande `orange.bleu` renverra 0.

▷ **Question 1.** Ecrire un objet vert. ◁

▷ **Question 2.** Ecrire une fonction `taux_de_bleu : couleur -> float` qui renvoie le pourcentage de bleu dans la couleur passée en argument. ◁

▷ **Question 3.** Ecrire une fonction `melange : couleur->couleur->couleur` qui prend en entrée deux couleurs et renvoie la couleur obtenue en mélangeant les deux couleurs en quantité égales c'est-à-dire en déterminant le pourcentage de chaque couleur primaire dans le mélange des deux couleurs passées en argument. ◁

2 TYPE SOMME

On définit un type somme par distinction de cas. Par exemple :

```
type valeur = As | Roi | Reine | Valet | Val of int
type symbole = Coeur | Carreau | Trefle | Pique
```

Le symbole | signifie ici ou.

Les différents champs sont des constructeurs et doivent nécessairement commencer par une majuscule. Si on leur associe une valeur (comme ici pour le champ Val) alors on utilise le mot clé `of` dans la déclaration du type et des parenthèses dans la déclaration de l'objet. Par exemple :

```
let ma_carte = {couleur = Coeur ; figure = Val(9)};;
```

On peut en déduire le type carte :

```
type carte = {couleur : symbole; figure : valeur}
```

Les types somme sont particulièrement adaptés au filtrage. Par exemple :

```
let noir = {rouge = 33.3; bleu=33.4; jaune = 33.3};;
let rouge = {rouge=100.; bleu=0.;jaune=0.};;
let rn carte =
match carte.couleur with
|Coeur->rouge
|Carreau-> rouge
|_-> noir;;
```

▷ **Question 4.** Ecrire une fonction `gagne : carte->carte->bool` qui renvoie `true` si la carte passée en premier argument est plus forte que celle passée en second et `false` sinon (on renvoie donc `false` en cas d'égalité). ◀

Le type `somme` peut être récursif comme nous l'avons déjà vu pour les listes : On a déjà croisé le type `int list` prédéfini qui correspond à :

```
type int liste = Vide | Const of int*(int liste)
```

On peut aussi définir des types polymorphes comme le type `'a liste` suivant :

```
type 'a liste = Vide | Const of 'a * 'a liste
```

▷ **Question 5.** Ecrire une fonction `maximum : 'a liste -> 'a` qui renvoie le maximum d'une liste passée en argument avec le type défini ci-dessus. ◀

▷ **Question 6.** Toujours en utilisant la syntaxe du type défini ci-dessus, écrire une fonction `verb+miroir : 'a liste->'a liste` qui retourne le miroir de la liste passée en argument. Cette fonction devra être linéaire. ◀

On avait aussi croisé le type suivant :

```
type programme_robot = Stop | Move of (int * int) * programme_robot ;;
```

▷ **Question 7.** Ecrire une fonction `deplace_robot` qui prend en entrée un `programme_robot` et renvoie les coordonnées du point où se trouve le robot après avoir effectué l'ensemble des déplacements à partir du point de coordonnées $(0, 0)$. ◀

3 ALIAS

On a vu dans la colle 3 le type polynôme que l'on aurait pu définir ainsi : `type polynome = (int*int) list.`

4 MANIPULATION DE LIMITES

L'objectif de cet exercice est d'écrire des fonctions permettant de retrouver les règles de calcul usuelles sur les limites des suites réelles.

On définit les deux types `somme` suivants :

```
type signe =  
  | Plus  
  | Moins
```

```
type limite =  
  | Infini of signe  
  | Zero of signe  
  | Nombre of float  
  | FormeIndeterminee
```

1. Définir une fonction `changement_signe` prenant en entrée un signe `signe1` et renvoyant le signe opposé à `signe1`. Donner le type de cette fonction.
2. Définir une fonction `regle_des_signes` prenant en entrée deux signes, `signe1` et `signe2`, et évaluant le signe obtenu en appliquant la règle des signes à `signe1` et `signe2`. Par exemple :

```
regle_des_signes Plus Moins;;  
- : signe = Moins
```

On cherche à écrire une fonction `oppose`, de type `limite ->limite`, qui renvoie la valeur de la limite de l'opposée d'une suite réelle de limite `limite1`. Un élève peu attentif en cours propose le programme situé ci-dessous.

```

let oppose limite1 =
  match limite1 with
  | FormeIndeterminee = FormeIndeterminee
  | Infini signe = Infini changement_signe(signe)
  | Zero signe = Zero changement_signe(signe)
  | Nombre valeur = Nombre(-valeur)

```

1. Proposez une version entièrement corrigée de la tentative de l'élève pour avoir un programme correctement typé et fonctionnel. Attention, il y a pas mal d'erreurs différentes !

On suppose désormais que ce programme a été correctement corrigé et qu'il est maintenant fonctionnel.

1. Que renvoie `oppose (Infini Plus);;` ?
2. Que renvoie `oppose (Nombre 48);;` ? *Attention, il y a un piège.*
3. Définir une fonction `inverse` prenant en entrée une limite notée `limite1` et retournant la limite de l'inverse d'une suite de limite `limite1`.
Attention à l'inverse de `Nombre 0.0` qui n'est pas la même que celle de `Zero Plus` ni celle de `Zero Moins`.
4. Définir une fonction `somme` prenant en entrée deux limites notées `limite1` et `limite2` et renvoyant la limite de la somme de deux suites de limites respectives `limite1` et `limite2`. On devra obtenir:

```

somme (Zero Moins) (Zero Plus);;
- : limite = Nombre 0.0
somme (Zero Plus) (Zero Plus);;
- : limite = Zero Plus

```

5. Définir une fonction `difference` prenant en entrée deux limites notées `limite1` et `limite2` et retournant la limite de la différence de deux suites de limites respectives `limite1` et `limite2`. *Il convient d'être astucieux.*

5 REPRÉSENTATION D'ENSEMBLES AVEC DES LISTES

En informatique, un *ensemble* est une structure de données qui permet de stocker une collection d'objets, sans ordre particulier et sans doublons. Il s'agit d'une mise en œuvre informatique de la notion mathématique d'ensemble fini. Dans cet exercice on propose de représenter un ensemble d'entiers à l'aide d'une liste *sans doublons*.

En OCAML, on peut définir des alias de type : `caml type ensemble = int list` Dans toute la suite le type `ensemble` est maintenant *synonyme* de `int list`. Cependant, il est interdit ici d'utiliser les fonctions prédéfinies dans la bibliothèque `List`.

1. Par quoi peut-on représenter un ensemble vide ?
2. Proposer une fonction `est_vide : ensemble -> bool` qui teste si un ensemble est vide ou non.
3. écrire une fonction `cardinal : ensemble -> int` qui renvoie le cardinal d'un ensemble (on rappelle que les listes sont supposées être sans doublons).
4. écrire une fonction `appartient : int -> ensemble -> bool` qui vérifie si un élément est dans un ensemble.
5. écrire une fonction `ajoute : int -> ensemble -> ensemble` qui ajoute un élément dans un ensemble. Insérer un élément déjà présent doit être sans effet.
6. écrire une fonction `supprime : int -> ensemble -> ensemble` qui élimine un élément s'il est présent et est sans effet sinon.
7. Proposer une fonction réalisant l'intersection de deux ensembles, en indiquant son type.
8. Proposer une fonction réalisant l'union de deux ensembles, en indiquant son type.
9. Proposer une fonction permettant de tester l'égalité de deux ensembles, en indiquant son type.

6 Arbres

On suppose défini le type arbre de la manière suivante :

```
type arbre =  
  | E  
  | Noeud of arbre * arbre
```

On définit la hauteur h d'un arbre par : $h(E) = 0$ et $h(\text{Noeud}(a1, a2)) = 1 + \max(h(a1), h(a2))$.

▷ **Question 8.** Ecrire une fonction `hauteur : arbre -> int` qui renvoie la hauteur de l'arbre passé en argument. ◁

Définissons maintenant le type :

```
type arbre_ord =  
  | E  
  | arbre_ord list
```

On généralise la notion de hauteur : $h(E) = 0$ et $h([a1, a2, \dots, an]) = 1 + \max(h(a1), h(a2), \dots, h(an))$.

▷ **Question 9.** Ecrire une fonction `hauteur_ord : arbre_ord -> int` qui renvoie la hauteur de l'arbre passé en argument. (on pourra écrire une fonction qui calcule le maximum d'une liste d'entiers et on utilisera la fonction `List.map`) ◁

7 Fractions

On définit le type suivant : `type fraction = {Num : int ; Denom : int}`. Ainsi `{Num = 2; Denom = 3}` représente $\frac{2}{3}$.

1. Ecrire une fonction récursive `pgcd` qui calcule le `pgcd` entre deux entiers passés en argument en utilisant l'algorithme d'Euclide.
2. La fonction `pgcd` peut-elle produire des erreurs de type `stack overflow` ? Justifier.
3. Ecrire une fonction `simplifie : fraction -> fraction` qui simplifie une fraction en divisant le numérateur et le dénominateur par leur `pgcd`.
4. Ecrire une fonction `ajoute : fraction -> fraction -> fraction` qui renvoie la fraction simplifiée correspondant à la somme des deux fractions passée en argument.