Chapitre 9 - Type Produit

Introduction

Dans ce chapitre, nous allons à présent voir comment :

- Continuer de définir et nommer nos propres types
- Étendre la notion de n-uplet pour obtenir des types regroupant différents champs de types différents. Ce sont les **types produits**.

Les types produits

Définition 1

Les types produits permettent de regrouper dans un même type plusieurs objets différents appelés **champs** de l'enregistrement, désignés par des noms appelés **étiquettes de champs**.
La syntaxe est la suivante :

```
type id = c_1 : t_1 ; ... ; c_n : t_n
```

où les `c_i` sont des identificateurs (les étiquettes de champs) et les `t_i` sont des types. Cette fois, les étiquettes de champs doivent commencer par des minuscules.
Pour définir un objet de type `id`, on utilisera la syntaxe :

```
c_1 = exp1 ; ... ; c_n = exp_n
```

où les `expi` sont des expressions de type `t_i`. Noter que dans la définition du type, on utilise les `:`, alors que lorsque l'on définit un objet de ce type, on utilise `=`.

Exemple 1

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

let z = {re = 1. ; im = 0.} ;;
val z : complexe = {re = 1.0 ; im = 0.0}
```

Pour accéder aux champs d'un type produit, on écrit alors simplement :

```
z.im ;;
- : float = 0.0
```

Et pour écrire des fonctions...

```
let memePartieRe = function (t,z) -> t.re = z.re  ; ;
memePartieRe  : complexe * complexe -> bool = <fun>
```

Ou alors, il est également possible d'utiliser les constructeurs de type produits et dans un filtrage :

```
let conjugue = function {re = a  ; im = b} -> {re = a  ; im = -. b}  ; ;
conjugue  : complexe -> complexe = <fun>
```

Exercice . Exercice TP 1 - Un peu de géométrie

On caractérise un point du plan à l'aide d'un type produit regroupant son abscisse et son ordonnée :

```
type Point = {abs  : float  ; ord  : float} ; ;

let p1 = {abs = 0.0 ; ord = 0.0} and
p2 = {abs = 2.0 ; ord = 0.0} and
p3  = {abs = 1.0 ; ord = 2.0} and
p4  = {abs = 0.0 ; ord = 1.0} ; ;
```

Un polygone du plan est alors défini comme une liste de points. On regroupe dans un type somme `Forme` les cercles (définis par leur centre et leur rayon) et les polygones :

```
type Forme = Cercle of Point*float | Polygone of Point list ; ;

let p=Polygone [p1 ;p2 ;p3 ;p4 ;p1] ; ;
```

1. Écrire une fonction `distance  : Point * Point -> float = <fun>` calculant la distance entre deux points.

   ```
   dis(p3,p2) ; ;
   -  : float = 2.2360679775
   ```

2. Écrire une fonction `longueur  : Point list -> float = <fun>` qui calcule la longueur d'une ligne brisée définie par la liste de ses sommets.

   ```
   longueur([p1 ;p2 ;p3]) ; ;
   -  : float = 4.2360679775
   ```

3. Une liste de points ne définit un polygone que si elle forme une figure fermée, c'est à dire, si la liste contient au moins quatre points et que le premier et le dernier point de la liste sont égaux.

   Écrire une fonction `bonPoly  : 'a list -> bool = <fun>` qui à une liste associe vrai si et seulement si elle possède au moins quatre éléments et que son premier et son dernier éléments sont égaux.

   ```
   bonPoly ([p1 ;p2 ;p3 ;p4 ;p1]) ; ;
   ```

```
-  : bool = true
bonPoly ([p1 ;p2 ;p3 ;p4 ;p3]) ; ;
-  : bool = false
bonPoly ([p1 ;p2 ;p3]) ; ;
-  : bool = false*)
```

4. Écrire une fonction `perimetre : Forme -> float = <fun>` calculant le périmètre d'un objet de type `Forme`. Dans le cas d'un polygone, on vérifiera que la liste de points donnée vérifie bien les conditions de la question précédente. Dans le cas d'un cercle de rayon r, rappelons que le périmètre est $2\pi r$.

```
perimetre p ; ;
-  : float = 6.65028153987

perimetre (Polygone [p1 ;p2 ;p3]) ; ;
Exception non rattrapée : Failure "ce n'est pas un polygone"

perimetre (Cercle (p1,1.)) ; ;
-  : float = 6.28318
```

Mélanger les types sommes et produits...

Exercice . Exercice TP 2 - Couleurs

Le but de cet exercice est de définir une couleur de diverse manière et d'écrire des fonctions de conversion et de traitement des couleurs. Cet exercice sera également l'occasion d'aborder (à peine) l'utilisation de la fenêtre graphique en Caml.

Une couleur peut être définie de plusieurs manières :

- On peut considérer une des trois couleurs de base Rouge, Vert ou Bleu
- Toute couleur contient une certaine quantité de Rouge, de Vert et de Bleu. Ces quantités sont des entiers compris entre 0 et 255. Ainsi, le "Blanc" est constitué de Rouge, Vert et Bleu en quantité maximum (255 chacun), alors que le "Noir" n'a ni Rouge, ni Vert, ni Bleu (0 chacun). On peut donc définir une couleur à l'aide de son code RVB.
- Enfin, on peut considérer la couleur obtenue en mélangeant deux autres couleurs.

Définition des types

1. Définir un type produit `CodeRVB` permettant de représenter une couleur par ses trois composantes R, V et B qui correspondent aux quantités respectives de Rouge, de Vert et de Bleu.
2. Définir un type somme `Couleur` permettant de représenter une couleur de base (Rouge, Vert ou Bleu), soit comme un mélange à parts égales de deux autres couleurs, soit par son code RVB représenté par un triplet d'entiers. Faites vérifier votre type par un professeur avant de vous lancer dans la suite de l'exercice !

Fonctions de conversion On souhaite convertir des objets du type `Couleur` dans le type `CodeRVB` et réciproquement. Par ailleurs, on aura parfois besoin de fournir les couleurs

sous forme de triplets (R,V,B). On définira donc les fonctions suivantes :

1. `rvbToCoul : CodeRVB -> Couleur = <fun>` qui convertit un objet de type `CodeRVB` en un objet de type `Couleur` représentant la même couleur.
2. `coulToRvb : Couleur -> CodeRVB = <fun>` qui convertit un objet de type `Couleur` en un objet de type `CodeRVB` représentant la même couleur.
3. `tripletToCoul : int * int * int -> CodeRVB = <fun>` permettant de créer une couleur (de type `Couleur`) à partir d'un triplet d'entiers correspondant à son code RVB.
4. `coulToTriplet : CodeRVB -> int * int * int = <fun>` qui convertit une couleur donnée par son code RVB en un triplet d'entiers.

Création de couleurs

1. Pour éclaircir une couleur (en gardant la même "teinte"), il suffit d'augmenter chaque composante RVB de la même quantité (on fait l'opération inverse pour l'assombrir).
   Écrire une fonction `eclaircir` et une fonction `assombrir` permettant d'éclaircir ou d'assombrir une couleur. *On pourra écrire des fonctions auxiliaires...*
2. Définir les fonctions `plusClaire` et `plusSombre` qui renvoient la couleur la plus claire possible (respectivement la plus sombre) correspondant à une couleur donnée. *On pourra écrire des fonctions auxiliaires...*
3. Écrire une fonction `nuancesRouge` qui, étant donnée une couleur, construit la liste des 256 couleurs que l'on peut obtenir en gardant les mêmes quantités de vert et de bleu et en faisant varier de 0 à 255 la quantité de rouge.

Exercice . Exercice TP 3 - Polynomes

Définissons les types suivants :

```
type monome= {coeff :float  ; deg :int}  ; ;

type polynome=Plein of int* float list
|Creux of monome list ; ;

let p = Plein(4,[1. ;0. ;3. ;0. ;5.]) ; ;
```

1. Ecrire une fonction `ajout : monome * polynome -> polynome` qui ajoute un monôme à un polynôme creux.
2. En déduire une fonction de conversion `pleinVersCreux : polynome -> polynome` qui transforme un polynome en représentation pleine vers un polynome en représentation creuse.