1 Type somme

Un type somme permet de représenter des catégories d'objets ayant plusieurs "cas". Il correspond à l'union disjointe mathématique, là où le produit de types correspond au produit cartésien d'ensembles.

Exemple 1. On veut implémenter un type pour les fournitures scolaires. On veut pouvoir représenter :

- Les stylos BIC, qui peuvent être de couleurs différentes
- Les règles, qui peuvent être de tailles différentes et peuvent être centrées ou pas (i.e. 0 est au centre ou au bord)
- Les gommes

Créez un fichier "fourniture.ml". Vous pourrez l'exécuter dans l'interpréteur en tapant :

```
#use "fourniture.ml";;
```

On définit le type somme avec :

```
type fourniture =
  | Stylo of string  (* couleur *)
  | Regle of int * bool (* taille en cm, centrée ou non  *)
  | Gomme
```

Ce qui se lit : "Il y a trois types de fournitures : Les stylos, qui sont paramétrés par une chaîne de caractères, les règles, paramétrées par un entier et un booléen, et les gommes". Les commentaires précisent à quoi servent les paramètres.
Les mots `Stylo`, `Regle` et `Gomme` sont appelés les **constructeurs** du type `fourniture`.

On peut ensuite créer des éléments de ce type :

```
let x = Stylo "rouge"
let r1 = Regle (30, true)
let r2 = Regle (20, false)
let g = Gomme
```

Notons qu'OCaml peut automatiquement tester l'égalité entre deux objets d'un même type :

```
let x = Stylo "rouge"
let y = Stylo "rouge";;
x = y;; (* Vaut true *)
```

La syntaxe utilisée pour définir ce nouveau type ressemble un peu au match with, et ce n'est pas un hasard : la manière principale de manipuler les types définis ainsi est précisément le ***match with***. Par exemple, on suppose que le prix des fournitures est comme suit :

- Les gommes coûtent 1,50€ ;
- Les stylos bleus coûtent 1,20€, les autres coûtent 1€ ;
- Une règle de l centimètres coûte $1 + \frac{l}{15}$ euros.

Voici comment on implémenterait une fonction calculant le prix d'une fourniture en OCaml :

```
(* prix de f en euros *)
let prix (f: fourniture) : float =
  match f with
  | Gomme -> 1.5
  | Stylo "bleu" ->  1.2
  | Stylo _ -> 1.0
  | Regle (longueur, _) -> 1.0 +. float_of_int longueur /. 15.0
```

On peut représenter une trousse comme une liste de fournitures. Écrivons une fonction qui calcule le nombre de gommes dans une trousse :

```
type trousse = fourniture list;;

let rec nombre_gommes (t:trousse) : int =
  match t with
  | [] -> 0
  | Gomme::q -> 1 + nombre_gommes q
  | _::q -> nombre_gommes q
```

Les différents cas donnés lors de la définition d'un type somme sont appelés les règles de constructions. Pour les fournitures, nous avons donc trois règles. Par exemple, l'une dit qu'à partir d'une chaîne de caractères s, on peut construire une fourniture **Stylo**(s).

Notons que nous avions déjà rencontré un type somme : les listes ! En effet, on peut définir l'ensemble des listes à partir de deux règles de constructions, la liste vide, et les listes non vides :

— La liste vide [] est une liste ;

— Pour x un élément et q une liste, on peut construire une nouvelle liste notée $x :: q$.

En OCaml, on pourrait donc définir un **nouveau** type pour les listes :

```
(* 'a est un paramètre de type. Il permettra d'avoir
   des int list, float list, etc... *)
type 'a liste =
  | LV (* Liste vide *)
  | LPV of 'a * 'a liste (* Liste pas vide *)

let liste_native = [1; 2; 3]
let liste_maison = LPV (1, LPV(2, LPV(3, LV)))
```

On pourrait alors réimplémenter toutes les fonctions classiques des listes avec ce nouveau type :

```
let rec taille (l: 'a liste) : int =
  match l with
  | LV -> 0
  | LPV (x, q) -> 1 + taille q
```

Ainsi, rien n'empêche un type somme de s'auto-référencer. On dit alors que c'est un type ***inductif***, ou récursif.

Syntaxe générale La syntaxe générale pour la création de type est donc :

```
type ('a, 'b, ...) nom_type =
  | Constructeur1 of type1 (* juste Constructeur1 s'il n'y a pas de paramètres *)
  | Constructeur2 of type2
  | ...
  | ConstructeurN of typeN
```

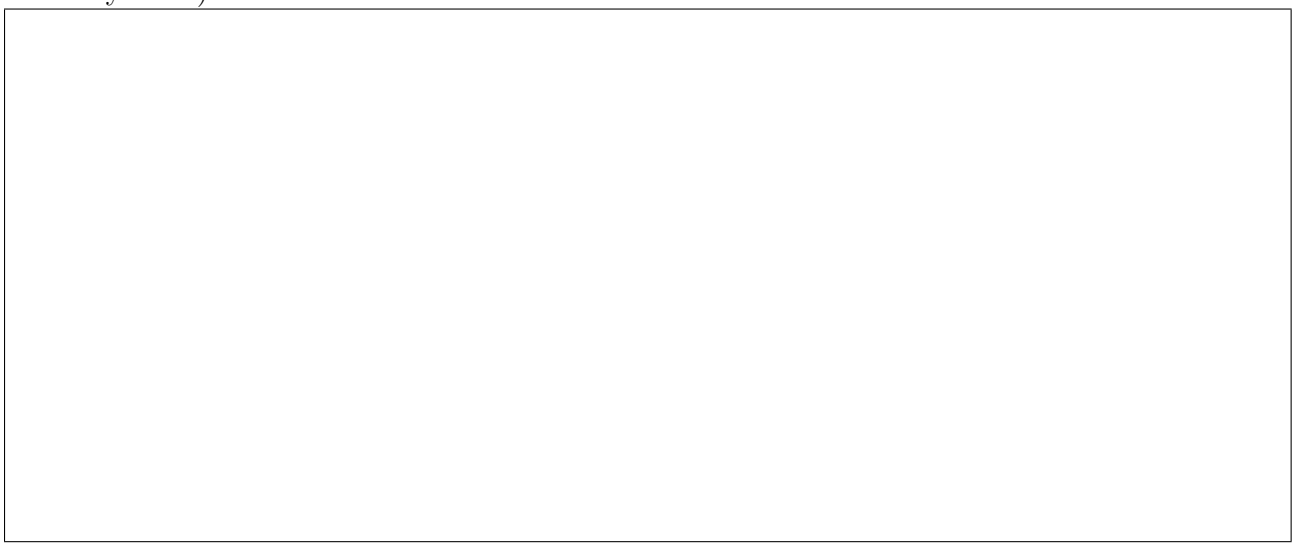
où les types `type1, ..., typeN` peuvent contenir le type `nom_type`. Les règles de constructions où c'est le cas sont dites **inductives**, et les autres règles sont dites **de base**. Les types peuvent aussi contenir `'a, 'b, ...`, qui sont appelés les **paramètres** de type.

2 Preuves par induction

Lorsque l'on manipule des types inductifs, on peut raisonner directement sur leur structure par disjonction de cas, en utilisant une généralisation des preuves par récurrence, appelé les preuves par **induction structurelle**.
Voyons le principe sur un exemple : on considère le type suivant permettant de représenter des couleurs :

```
1 type couleur =
2   | Rouge | Jaune | Bleu (* couleurs primaires) *)
3   | Melange of couleur * couleur (* 50% de chaque *)
```

Voici quelques éléments du type `couleur`, et la manière dont OCaml les voit (i.e. leurs arbres de syntaxe) :



On définit une fonction récursive calculant le pourcentage de rouge dans une couleur quelconque :

```
1 (* fraction de rouge dans la couleur *)
2 let rec fr (c: couleur) : float =
3   match c with
4   | Rouge -> 1.0
5   | Bleu  -> 0.0
6   | Jaune -> 0.0
7   | Melange (c1, c2) -> (fr c1 +. fr c2) /. 2.
```

On suppose avoir défini de même les fonctions `fj` (fraction de jaune) et `fb` (fraction de bleu).

Montrons par induction structurelle sur le type `couleur` que :

$$\text{Pour toute couleur } c,\ P(c) : \text{“}\mathbf{fr}(c) + \mathbf{fb}(c) + \mathbf{fj}(c) = 1$$

Il y a 4 cas, un par constructeur dans le type somme :

- Pour $c = \mathbf{Rouge}$: $\mathbf{fr}(c) = 1$ et $\mathbf{fj}(c) = \mathbf{fb}(c) = 0$, d'où $P(c)$ vraie : $1 + 0 + 0 = 1$.
- Pour $c = \mathbf{Jaune}$: analogue
- Pour $c = \mathbf{Bleu}$: analogue
- Pour $c = \mathbf{Melange}(c_1, c_2)$, par hypothèse d'induction, on a $P(c_1)$ et $P(c_2)$, i.e. :

$$\mathbf{fr}(c_1) + \mathbf{fb}(c_1) + \mathbf{fj}(c_1) = 1$$
$$\mathbf{fr}(c_2) + \mathbf{fb}(c_2) + \mathbf{fj}(c_2) = 1$$

Or, on a :

$$\begin{aligned} & \mathbf{fr}(c) + \mathbf{fb}(c) + \mathbf{fj}(c) \\ = \ & \tfrac{1}{2}(\mathbf{fr}(c_1) + \mathbf{fr}(c_2)) + \tfrac{1}{2}(\mathbf{fb}(c_1) + \mathbf{fb}(c_2)) + \tfrac{1}{2}(\mathbf{fj}(c_1) + \mathbf{fj}(c_2)) \\ = \ & \tfrac{1}{2}(\mathbf{fr}(c_1) + \mathbf{fb}(c_1) + \mathbf{fj}(c_1)) + \tfrac{1}{2}(\mathbf{fr}(c_2) + \mathbf{fb}(c_2) + \mathbf{fj}(c_2)) \\ = \ & \tfrac{1}{2} + \tfrac{1}{2} \\ = \ & 1 \end{aligned}$$

Nous avons montré la propriété énoncée par **induction structurelle** sur les couleurs. Notons que c'est presque comme si nous avions fait une preuve par récurrence sur la profondeur des couleurs (i.e. sur le nombre maximal de `Mélange` imbriqués).

Formellement, on peut énoncer un principe de preuve par induction pour **chaque** type inductif. Par exemple, pour le type couleur :

Proposition 1. Soit P une propriété portant sur les couleurs. On suppose que :

- $P(\mathbf{Rouge})$ est vraie
- $P(\mathbf{Bleu})$ est vraie
- $P(\mathbf{Jaune})$ est vraie
- Pour toutes couleurs c_1, c_2, si $P(c_1)$ et $P(c_2)$ sont vraies, alors $P(\mathbf{Melange}(c_1, c_2))$ est vraie.

Alors, $P(c)$ est vraie pour toute couleur c.

On ne s'intéressera pas dans ce cours à la démonstration de cette propriété, mais pourra mentionner qu'elle est fortement liée au principe d'ordre bien fondé :

Définition 1. Un ordre $(E, \leq)$ est bien fondé s'il n'admet aucune suite infinie strictement décroissante.

En effet, la propriété précédente permet de réduire la preuve de $P(\mathbf{Melange}(c_1, c_2)$ à la preuve de $P(c_1)$ et de $P(c_2)$, qui elles mêmes se réduisent à des preuves plus petites, et ainsi de suite en "remontant" jusqu'aux trois cas de base.

Exercice 1.

Question 1. Énoncer le principe d'induction sur les listes OCaml.

On considère les fonctions suivantes :

```
1  (* produit des éléments de l *)
2  let rec f (l: int list) : int =
3    match l with
4    | [] -> 1
5    | x :: q -> x * f q
6
7  let rec g (l: int list) (a: int) : int =
8    match l with
9    | [] -> a
10   | x :: q -> g q (x * a)
```

Question 2. Montrer par induction sur les listes que pour toute liste L, pour tout $a \in \mathbb{N}$, $g(L, a) = f(L) \times a$.