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 peut définir un type somme `fourniture` en écrivant :

```
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,

— les gommes, qui sont toutes identiques."

Les commentaires précisent le rôle des paramètres.
Les mots `Stylo`, `Regle` et `Gomme` sont appelés les **constructeurs** du type `fourniture`.

On peut ensuite créer des éléments de ce type, en mettant un constructeur suivi de l'éventuel tuple des paramètres :

```
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 méthode principale permettant d'utiliser les types sommes est le match-with. Par exemple, supposons que les fournitures ont les prix suivants :

— 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.

On peut implémenter cette fonction de prix comme suit :

```
(* 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 règles centrées dans une trousse :

```
let rec nb_regles_centree (t:fourniture list) : int =
  match t with
  | [] -> 0
  | Regle (_, true) :: q -> 1 + nb_regles_centree q
  | _::q -> nb_regles_centree q
```

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 liste, float liste, 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)))
```

Notons que ce type est récursif : ce n'est pas un problème en OCaml, on verra que c'est même très courant.
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.
La syntaxe générale pour la création de type somme est :

```
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` (c'est alors une définition inductive), ainsi que les types `'a, 'b, ...`, qui sont appelés les **paramètres** de type.
Les règles de constructions faisant apparaître `nom_type` sont dites **récursives** ou **inductives**, et les autres règles sont dites **de base**.

Cette définition de type ajoute alors aux expressions OCaml toutes les expressions de la forme `Constructeurk e`, où `e` est une expression de type `typek`. Elle ajoute aussi aux motifs `Constructeurk m`, où `m` est un motif compatible avec `typek`

Voyons un exemple plus poussé de type récursif :

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

On peut alors définir des couleurs comme suit :

```
1 let vert = Melange (Jaune, Bleu)
2 let orange = Melange (Jaune, Rouge)
3 let orange_clair = Melange(orange, Jaune)
```

Notons que ces définitions sont purement syntaxiques : OCaml n'a aucune idée du **sens** qu'a ce type. En particulier, les deux éléments suivants ne sont pas égaux :

```
1 let orangeA = Melange (Melange (Jaune, Rouge), Melange (Rouge, Jaune))
2 let orangeB = Melange (Rouge, Melange (Jaune, Jaune))
```

En effet, du point de vue d'OCaml, les valeurs stockées dans les deux variables sont des arbres de syntaxe. Ainsi, dans les deux cas, même si les proportions de jaune et de rouge sont égales, pour OCaml, on n'a pas `orangeA = orangeB`.

Cependant, on peut définir des fonctions calculant le pourcentage de rouge, jaune et bleu 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.
8 (* On définit de même fb et fj qui calculent la fraction de bleu et de jaune *)
```

Alors, on n'a pas `orangeA = orangeB`, mais on a bien `fr orangeA = fr orangeB`, `fb orangeA = fb orangeB` et `fj orangeA = fj orangeB` : on a donné une sémantique au type !

2 Preuves par induction

Rappelons le principe de récurrence faible sur $\mathbb{N}$:

Théorème 1

Soit P une propriété sur $\mathbb{N}$ telle que :

— $P(0)$

— $\forall n \in \mathbb{N}, P(n) \Rightarrow P(n+1)$

Alors, $\forall n \in \mathbb{N}, P(n)$

La raison pour laquelle ce théorème tient, intuitivement, est que pour montrer $P(n)$, il suffit de montrer $P(n-1)$, donc $P(n-2)$, etc..., mais cette chaîne n'est pas infinie : on tombe forcément sur $P(0)$, qui est vraie par hypothèse. Le principe de récurrence ne fonctionnerait pas sur $\mathbb{Z}$, car il n'y a pas de point de départ, et il ne fonctionnerait pas non plus sur $\mathbb{R}^+$, car il peut y avoir des suites strictement décroissantes infinies.

On peut énoncer un principe similaire au principe de récurrence sur les types inductifs. Reprenons le type `couleur` de la partie précédente :

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

Notons E l'ensemble mathématique des éléments de ce type. On ne détaillera pas sa construction, mais notons qu'il contient des éléments tels que :

$$\mathbf{Rouge}, \mathbf{Jaune}, \mathbf{Melange}(\mathbf{Rouge}, \mathbf{Bleu}), \mathbf{Melange}(\mathbf{Melange}(\mathbf{Rouge}, \mathbf{Bleu}), \mathbf{Jaune}), \dots$$

Alors, on a le **principe d'induction structurelle** sur le type `couleur` :

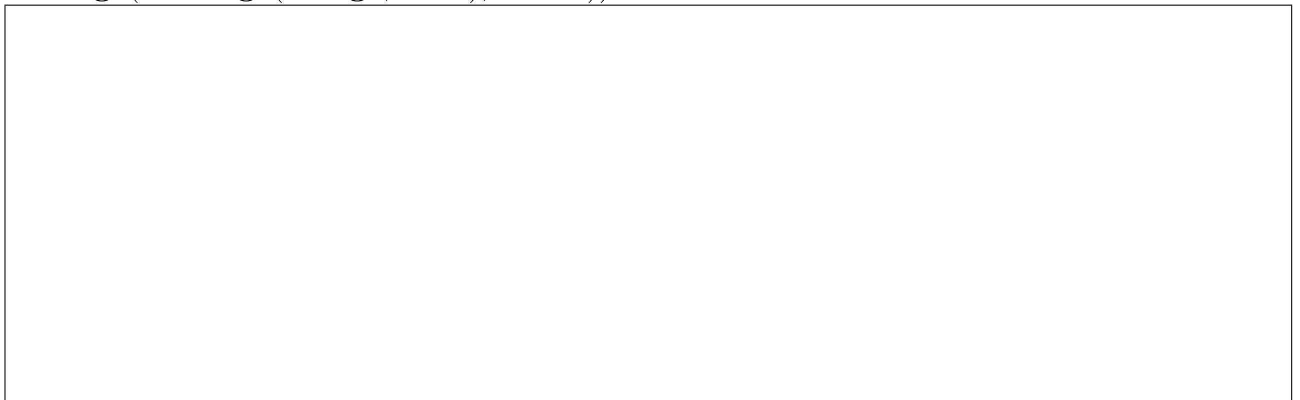
Théorème 2

Soit P une propriété sur E telle que :

- $P(\mathbf{Rouge})$
- $P(\mathbf{Bleu})$
- $P(\mathbf{Jaune})$
- Pour tout $c_1, c_2 \in E$, si $P(c_1)$ et $P(c_2)$ alors $P(\mathbf{Melange}(c_1, c_2))$

Alors, $\forall c \in E, P(c)$

A nouveau, cet énoncé tient car les objets de l'ensemble E sont de tailles finies, on peut donc **revenir aux éléments de base**. Par exemple, voici un schéma expliquant pourquoi $P(\mathbf{Melange}(\mathbf{Melange}(\mathbf{Rouge}, \mathbf{Bleu}), \mathbf{Jaune}))$ est vraie :



Utilisons maintenant le principe d'induction structurelle pour montrer une propriété simple : si l'on additionne les fractions de rouge, de jaune, et de bleu dans une couleur, on obtient toujours 1. Cela semble physiquement évident, mais la preuve permettra de montrer que les fonctions que l'on a écrit sont bien correctes.

La rédaction est similaire à celle d'une récurrence, mais on ne distingue pas les cas de base des cas inductifs (pas d'initialisation/hérédité). A la place, on fait un cas par constructeur du type somme : pour le type `couleur` il y en aura 4.

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

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

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

— Pour $c =$ **Rouge** : $\mathbf{fr}(c) = 1$ et $\mathbf{fj}(c) = \mathbf{fb}(c) = 0$, d'où $P(c)$ vraie : $1 + 0 + 0 = 1$.

— Pour $c =$ **Jaune** : analogue

— Pour $c =$ **Bleu** : analogue

— Pour $c =$ **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 :

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

D'où $P(c)$

Ainsi, par induction structurelle, $P(c)$ est vraie pour toute couleur c.

Ce nouveau formalisme permet en particulier d'écrire des preuves sur les listes plus élégantes que les preuves par récurrence sur la longueur que nous avons utilisé précédemment.

Exercice 1

Q1. Énoncer le principe d'induction sur les listes OCaml.

On considère les fonctions suivantes :

```
(* produit des éléments de l *)
let rec f (l: int list) : int =
  match l with
  | [] -> 1
  | x :: q -> x * f q

let rec g (l: int list) (a: int) : int =
  match l with
  | [] -> a
  | x :: q -> g q (x * a)
```

Q2. Montrer par induction sur les listes que pour toute liste L, pour tout $a \in \mathbb{N}$,

$g(L, a) = f(L) \times a$.

En particulier, cette preuve (simple) montre que `g` est une version récursive terminale valide de `f` !