TP n° 3 : Types sommes en CAML

Nous avons rencontré les types de base en CAML : `bool`, `int`, `float`, `char`, `string` et des opérations algébriques sur les types (`int -> float`, `(string * char * bool)`, etc). En CAML, on a également la possibilité de définir nos propres types personnalisés.

1 Types sommes

Imaginons par exemple que nous souhaitions ajouter aux booléens *vrai* et *faux* une valeur qui permette de représenter l'indécision. On peut alors définir un type :

OCAML

```
type trileen =
  | Vrai
  | Faux
  | PeutEtre
;;
```

La phrase `type nom_du_type = ...` permet de définir un nouveau type. Le caractère `|` se lit « ou[1] ». Les *constructeurs* `Vrai`, `Faux` et `PeutEtre` sont alors définis et sont les (seules) valeurs possibles du type `trileen`.

OCAML

```
Vrai;;
- : trileen = Vrai

PeutEtre;;
- : trileen = PeutEtre

Faux = PeutEtre;;
- : bool = false   (* Booléen standard ! *)
```

On appelle un type défini de cette manière un *type somme*. Intuitivement on réalise effectivement une « somme » ou une union de constructeurs simples.

Les noms de constructeurs commencent par une majuscule. Par convention, à respecter absolument, les noms de variables ou de fonctions ne doivent donc jamais commencer par une majuscule.

Pour utiliser les types sommes, et pour bien d'autres choses comme nous le verrons prochainement, on dispose en CAML d'un outil très puissant : `le filtrage de motifs`.

Définition 1.1

Le filtrage est une *expression* dont la syntaxe est :

1. À ne pas confondre avec le « ou logique » qui s'écrit `||`.

OCAML

```
match expr0 with
| motif1 -> expr1
| motif2 -> expr2
...
```

Toutes les expressions `expr1`, `expr2`, ... doivent être de même type, qui est alors le type de cette expression.

1. Comparaison de la valeur de `expr0` avec les *motifs* ;
2. Dans l'ordre et on prend le premier qui correspond ;
3. Le filtrage doit être *exhaustif* (un des motifs doit correspondre).

Par exemple, si `prop` est de type `trileen`, on réalise un filtrage pour agir selon les cas :

OCAML

```
(* Réalise le "non" en logique triléenne *)
let non_trileen prop =
  match prop with
  | Vrai -> Faux
  | PeutEtre -> PeutEtre
  | Faux -> Vrai
;;
non_trileen : trileen -> trileen = <fun>

let prop = Vrai;;
prop : trileen = Vrai

non_trileen prop;;
- : trileen = Faux
```

CAML détecte automatiquement que tous les cas ont bien été considérés et renvoie un message d'avertissement si ce n'est pas le cas :

OCAML

```
let hesiter prop =
  match prop with
  | Vrai -> PeutEtre
  | Faux -> PeutEtre
;;
Warning 8: this pattern-matching is not exhaustive.
```

C'est un trait très puissant de CAML. Ne pas avoir traité tous les cas est détecté *avant même d'exécuter le programme* !

Question 1

Imaginer ce que peut être un « et » en logique triléenne et écrire une fonction `et_trileen : trileen -> trileen -> trileen` qui le réalise.

2 Constructeurs avec paramètres

Jusqu'ici les trois constructeurs étaient des *constantes*. On peut également définir des constructeurs paramétrés qui dépendent d'un autre type avec la syntaxe `NomConstructeur of un_type`. Par exemple,

on peut définir un jeu de cartes avec :

OCAML

```
type couleur =
  | Pique
  | Coeur
  | Carreau
  | Trefle
;;
```

OCAML

```
type carte =
  | Joker
  | As of couleur
  | Roi of couleur
  | Dame of couleur
  | Valet of couleur
  | Nombre of int * couleur
;;
```

On peut alors définir des cartes avec la syntaxe `NomConstructeur (expr : un_type)` :

OCAML

```
let carte1 = As Pique;;
carte1 : carte = As Pique

let carte2 = Nombre (3, Carreau);;
carte2 : carte = Nombre (3, Carreau)
```

Question 2

Définir le roi de cœur, la dame de pique et le neuf de trèfle.

Question 3

Prévoir et expliquer ce que renvoient les expressions suivantes :

- `carte1 = carte2;;`
- `carte2 = (3, Carreau);;`
- `Pique;;`
- `Joker;;`
- `As;;`
- `Nombre;;`

Un constructeur est plus qu'une simple fonction : on peut faire un filtrage de motif :

OCAML

```
let est_carreau carte =
  match carte with
  | Joker -> PeutEtre
  | As Carreau -> Vrai
  | Roi Carreau -> Vrai
  | Dame Carreau -> Vrai
  | Valet Carreau -> Vrai
  | Nombre (_, Carreau) -> Vrai
  | _ -> Faux
;;
est_carreau : carte -> trileen = <fun>
```

☞ **Remarque 1**

On utilise la variable spéciale _ pour filtrer sur n'importe quel motif : « dans tous les autres cas ». C'est souvent pratique, mais attention à ne surtout pas prendre la mauvaise habitude de systématiquement ajouter une clause `| _ -> ...` en fin de filtrage pour éviter les messages d'avertissement lorsqu'un filtrage n'est pas exhaustif !

Question 4

Écrire une fonction `est_figure : carte -> trileen` qui indique si la carte reçue en entrée est une figure ou non.

On pourrait, sans difficultés mais avec beaucoup de temps, programmer le jeu de belote. Par exemple pour compter les points à la fin de la partie on peut écrire :

OCAML

```
let valeur carte atout =
  match carte with
  | Joker -> failwith "Carte impossible"
  | Nombre (n, _) when n < 7 -> failwith "Carte impossible"
  | As _ -> 11
  | Nombre (10, _) -> 10
  | Roi _ ->  4
  | Dame _ -> 3
  | Valet color when color = atout -> 20
  | Valet _ -> 2
  | Nombre (9, color) when color = atout -> 14
  | Nombre _ -> 0
;;
```

Question 5

À quoi peut servir le type suivant ?

OCAML

```
type mixte =
  | Reel of float
  | Int of int
;;
```

Question 6

Définir un type `reel_etendu` permettant de représenter la droite numérique achevée $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$ et une fonction `float -> reel_etendu` qui permet de plonger $\mathbb{R}$ dans $\overline{\mathbb{R}}$.

3 Types récursifs

Il est possible de définir des types *récursifs*, c'est-à-dire dont leur définition fait appel à eux-même. Par exemple, on peut définir les couleurs par synthèse soustractive :

OCAML
```
type coloration =
  | Cyan
  | Magenta
  | Jaune
  | Melange of coloration * coloration
;;
```

Question 7

Définir en CAML le rouge (mélange de magenta et de jaune) puis l'orange (mélange de rouge et de jaune).

Question 8

Que fait la fonction suivante ?

OCAML
```
let rec cmj color =
  match color with
  | Cyan -> (1., 0., 0.)
  | Magenta -> (0., 1., 0.)
  | Jaune -> (0., 0., 1.)
  | Melange (color1, color2) ->
     let c1, m1, j1 = cmj color1 in
     let c2, m2, j2 = cmj color2 in
     ((c1 +. c2) /. 2., (m1 +. m2) /. 2., (j1 +. j2) /. 2.)
;;
```

4 Listes chaînées

On peut maintenant définir notre propre type CAML pour créer des listes chaînées. Une liste chaînée est soit la liste vide (`Nil`), soit une tête et une queue.

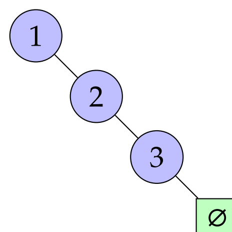

FIGURE 1 – Une représentation possible de la liste `[1; 2; 3]`. La tête de la liste est `1` et la queue `[2; 3]`.

On peut par exemple choisir le type CAML suivant :

OCAML
```
type 'a liste =
  | Nil
  | TeteEtQueue of 'a * 'a liste
;;
```

On a construit ici un type *paramétré* qui dépend d'un type `'a`.

Question 9

Définir avec ce type la liste `[1; 2; 3]`, représentée sur la figure 1.

On peut alors écrire une fonction `cons : 'a -> 'a liste -> 'a liste` qui ajoute un élément en tête de liste :

OCAML
```
let cons tete queue =
  TeteEtQueue (tete, queue)
;;
```

Question 10

Ajouter 0 en tête de la liste précédente.

Question 11

Écrire les fonctions `tete : 'a liste -> 'a` et `queue : 'a liste -> 'a liste` qui renvoient la tête et la queue d'une liste si possible et un message d'erreur sinon (ce que l'on peut obtenir à l'aide de `failwith "Message"`).

Question 12

Écrire la fonction `taille : 'a liste -> int` qui renvoie la longueur d'une liste.

Question 13

Écrire une fonction `appartient : 'a -> 'a liste -> bool` qui vérifie si un élément appartient à une liste.

Question 14

Écrire une fonction `supprime : 'a -> 'a liste -> 'a liste` qui renvoie une liste identique à celle passée en argument à ceci près que toutes les occurrences éventuelles de l'élément passé en argument ont été supprimées.