TP n° 7 : Types récursifs et arbres

Pour sauver un arbre, mangez un castor — Henri Prades

1 Types sommes en CAML

Nous avons rencontré les types de base en CAML (`unit`, `bool`, `int`, `float`, `char`, `string`) ainsi que des types prédéfinis pour des structures de données (`'a list`, `'a ref`, `'a vect`). En CAML, on a également la possibilité de définir nos propres types personnalisés.

1.1 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 :

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

CAML nous répond `Type trileen defined.` La *phrase* `type nom_du_type = ...;;` permet de définir un nouveau type. Le caractère `|` se lit « ou » comme pour le filtrage ou pour les définitions par cas. Les *constructeurs* `Vrai`, `Faux` et `PeutEtre` sont alors définis et sont les (seules) valeurs possibles du type `trileen`.

```
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.

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

```
(* 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 :

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

C'est un trait très puissant de CAML. Ne pas avoir traité tous les cas est détecté [1] *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.

1.2 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 :

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

1. À condition de ne pas prendre la mauvaise habitude de systématiquement ajouter une clause `| _ -> ...` en fin de filtrage.

```
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)` :

```
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;;`

Une fois que l'on a bien compris cela, la syntaxe pour créer une valeur avec un constructeur devient limpide ! Cependant, un constructeur est plus qu'une simple fonction : on peut faire un filtrage de motif.

```
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>
```

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 :

```
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 ?

```
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 $\overline{\mathbb{R}}$.

1.3 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 :

```
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 ?

```
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.)
;;
```

2 Retour sur les listes

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.

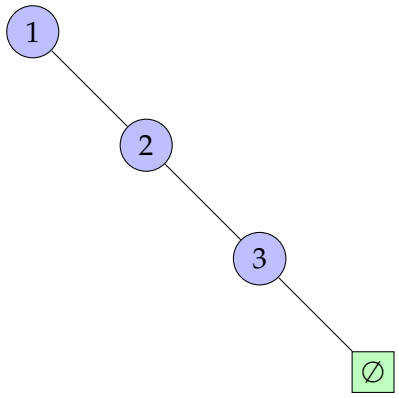

FIGURE 1 – Une représentation possible de la liste `[1; 2; 3]`.

On peut par exemple choisir le type CAML suivant :

```
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` et qui est l'analogue de `::` en notation postfixe :

```
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.

QUESTION 12

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

On pourra, pour s'entraîner, essayer de réécrire toutes les fonctions que nous avons vues jusqu'ici sur les listes, avec ce type personnalisé. Bien entendu, nous continuerons principalement à utiliser les listes CAML, puisqu'elles sont faites pour cela.

3 Arbres

On peut voir les arbres comme une généralisation de la structure de liste. À une tête (que l'on appelle plutôt *nœud*) on associe non pas une queue, mais deux (que l'on appelle sous-arbres) :

```
type 'a arbre =
  | Vide
  | Noeud of 'a * 'a arbre * 'a arbre
;;
```

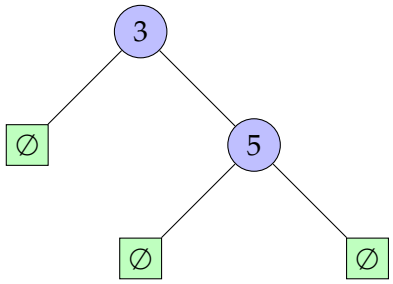

FIGURE 2 – Un exemple d'arbre avec deux nœuds. La racine d'étiquette 3 possède deux sous-arbres : son fils gauche qui est un sous-arbre vide et son fils droit qui est un sous-arbre de taille 1 (une feuille), dont l'étiquette est 5.

Un arbre non vide est de la forme `Noeud (etiquette, fils_gauche, fils_droit)`. Le nœud est appelée `racine` de l'arbre (l'analogue de la tête d'une liste) et on dit alors que `etiquette` est l'*étiquette* de ce nœud. Les sous-arbres `fils_gauche` et `fils_droits` sont respectivement le *fils gauche* et le *fils droit* de l'arbre (l'analogue de la queue de la liste, mais il y en a deux). On appelle *feuille* un nœud dont les deux fils sont des arbres vides.

QUESTION 13

Définir en CAML l'arbre représenté sur la figure 2.

QUESTION 14

Écrire une fonction `nb_noeuds : 'a arbre -> int` qui renvoie le nombre de nœuds d'un arbre (c'est l'analogue de la fonction `list_length`).

QUESTION 15

Écrire une fonction `nb_feuilles : 'a arbre -> int` qui renvoie le nombre de feuilles d'un arbre.

QUESTION 16

Écrire une fonction `imprime : int arbre -> unit` qui imprime les étiquettes d'un arbre, dans l'ordre que vous voulez, mais une et une seule fois chacune.

QUESTION 17

Écrire une fonction `somme_etiquettes : int arbre -> int` qui renvoie la somme des étiquettes des nœuds de l'arbre.

QUESTION 18

Écrire une fonction `max_etiquettes : 'a arbre -> 'a` qui renvoie la somme des étiquettes des nœuds de l'arbre.

QUESTION 19

Écrire une fonction `somme_feuilles : int arbre -> int` qui renvoie la somme des étiquettes de toutes les feuilles.

QUESTION 20

Écrire une fonction `max_min_arbre : int arbre -> int * int` qui renvoie le minimum et le maximum des étiquettes d'un arbre.

QUESTION 21

Une branche est un chemin de la racine vers une feuille. Le poids d'une branche est la somme des étiquettes des nœuds qui la composent. Écrire une fonction `max_somme_branche : int arbre -> int` qui renvoie le poids de la branche de poids maximal.