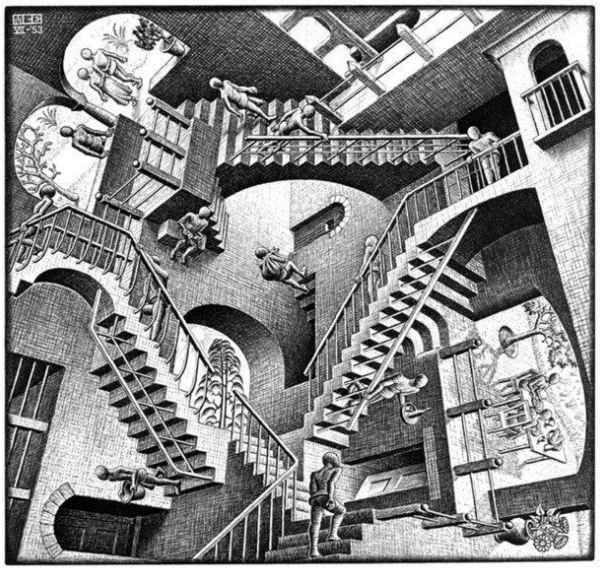
TP n° 8 : Types et structures de données



1 Types personnalisés

1.1 Alias de types

Comme vous l'avez vu dans le DS, en CAML on peut donner un nouveau nom à un type existant :

```
type complexe == float * float;;
```

Il s'agit d'une abréviation — d'un surnom — pour un type déjà existant et non de la définition d'un *nouveau* type, contrairement aux types sommes. On comprend alors l'utilisation en CAML [1] du double symbole égal et non d'un simple égal.

Cependant, CAML continue d'interpréter les couples de flottants avec le type de base et non avec notre nouveau type pour les complexes :

```
(3., 4.);;
- : float * float = 3.0, 4.0
```

Il est possible de forcer le type à l'aide de la syntaxe suivante :

```
((3, 4) : complexe);;
- : complexe = 3.0, 4.0

let somme (z1 : complexe) (z2 : complexe) =
  let (re1, im1), (re2, im2) = z1, z2 in
  ((re1 +. re2, im1 +. im2) : complexe)
;;
```

EXERCICE 1 Définir la fonction `produit : complexe -> complexe -> complexe`.

1. Mais pas en OCAML.

Néanmoins forcer les types n'offre pour nous que peu d'intérêt[2] et alourdi l'écriture. Après tout il s'agit de synonymes exacts, alors oublions l'esthétique et laissons l'inférence de type de CAML faire son travail ! On ne cherchera donc pas à respecter au mot près la signature des fonctions (mais les types doivent bien sûr être compatibles, c'est-à-dire équivalent ou plus généraux).

1.2 Types enregistrement

Lorsque l'on souhaite représenter une structure de données comportant plusieurs champs, on peut utiliser en CAML un type produit.

Imaginons par exemple que l'on souhaite représenter un élève ainsi que les notes qu'il a obtenu dans différentes matières. On pourrait définir un type :

```
type bulletin == string * int * int * int * int * int;;
```

Mais il faut alors se souvenir exactement quelle position correspond à quelle matière ! Ce n'est pas pratique du tout. Il existe en CAML une structure de données appelée *enregistrement* ou *type produit nommé*.

En revenant sur l'exemple précédent, on pourrait alors définir :

```
type bulletin = {
  etudiant : string;
  note_maths : int;
  note_physique : int;
  note_philo : int;
  note_info : int;
  note_langues : int
}
```

Pour définir les nombres complexes[3] on peut prendre un enregistrement à deux *champs* :

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

Pour « créer » le nombre complexe $z = 2 + i$ on écrira :

```
let z = {re = 2.0; im = 1.0};;
z : complexe = {re = 2.0; im = 1.0}
```

QUESTION 1

Pourquoi CAML a-t-il reconnu qu'il s'agissait d'une valeur de type `complexe` ici ?

z est une structure contenant deux « cases » appelées `re` et `im` dont le contenu est accessible, comme pour les vecteurs à l'aide du symbole « . » :

```
z.re;;
- : float = 2.0
z.im;;
- : float = 1.0
```

On a, en particulier, un accès direct à tous les champs d'un élément.

2. Sauf éventuellement pour des TIPE, mais il existe alors d'autres moyens plus intéressants.
3. Pas exactement les nombres complexes bien sûr. Pourquoi ?

On peut par exemple définir la fonction de conjugaison :

```
let conjuge z = {re = z.re; im = -.z.im};;
conjuge : complexe -> complexe = <fun>
```

ou encore :

```
let conjuge {re = a; im = b} = {re = a; im = -.b};;
```

EXERCICE 2

Écrire les fonctions `somme : complexe -> complexe -> complexe` et `produit : complexe -> complexe -> complexe` avec ce nouveau type.

Par défaut, les champs ne sont pas modifiables. On peut cependant ajouter le mot-clé **`mutable`** lors de la définition du type :

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

On utilise alors la même syntaxe que pour les tableaux pour modifier la valeur d'un champs :

```
let z = {re = 2.0; im = 1.0};;
z : complexe = {re = 2.0; im = 1.0}
z.re <- z.im -. 1.;;
- : unit = ()
z;;
- : complexe = {re = 0.0; im = 1.0}
```

EXERCICE 3

Réécrire la fonction `conjuge` mais cette fois-ci de type `complexe -> unit` et non `complexe -> complexe`.

1.3 Polymorphie

Qu'il s'agisse de type somme ou de type produit, il est possible de paramétrer les types comme dans les exemples suivants :

```
type 'a boite = {taille : int; contenu : 'a list};;
```

```
type 'a option =
  | None
  | Some of 'a
;;
```

Remarque 1. *Ce dernier type est prédéfini en* CAML *et est très utile lorsque dans certains cas on ne veut pas renvoyer de valeur dans une fonction, par exemple lors de la recherche d'un élément si l'élément n'est pas présent. La sortie doit toujours être de même type et le type `'a option` permet donc d'encapsuler les deux cas.*

EXERCICE 4

Écrire une fonction `recherche 'a -> 'a vect -> int option` qui recherche le premier indice d'un élément dans un tableau.

QUESTION 2

Que pensez-vous du type :

```
type 'a my_ref = {mutable content : 'a};;
```

2 Arbres n-aires et forêts

Dans cette section on s'intéresse à une généralisation possible des arbres binaires vus en cours. Chaque nœud peut avoir un nombre illimité (mais fini !) de fils. Autrement dit, le degré d'un arbre n-aires n'est plus nécessairement 2, mais peut-être quelconque.

Définition 1. *Un arbre n-aire sur un ensemble de valeurs de nœuds $\mathcal{X}$ est soit l'arbre vide, soit un couple :*

$$(x, (a_1, \ldots, a_k))$$

avec

- $x \in \mathcal{X}$ *l'étiquette du nœud (racine)*
- $k \in \mathbb{N}$ *l'*arité *du nœud*
- $a_1, \ldots, a_k$ *des arbres n-aires non-vides*

Remarque 2. *L'équivalent d'une feuille est donc un nœud d'arité* 0.

Définition 2. *Une* forêt *est un ensemble fini (éventuellement vide) d'arbres non vides sans nœuds communs.*

Remarque 3. *Les fils d'un nœuds forment donc une forêt.*

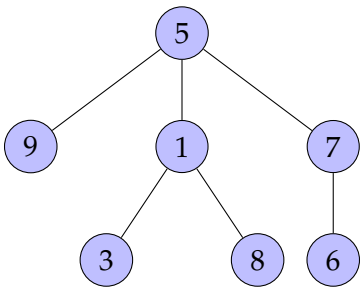
En CAML on peut définir les arbres n-aires ainsi :

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

On peut vouloir donner un nom plus explicite à une liste d'arbres qui est une forêt ! Mais alors pour définir une forêt il faut savoir définir un arbre et réciproquement... De la même manière que l'on peut définir des fonctions mutuellement récursives, on peut définir des types mutuellement récursifs :

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

```
let exemple = Noeud (5,
  [Noeud (9, []);
   Noeud (1, [Noeud (3, []); Noeud (8, [])]);
   Noeud (7, [Noeud (6, [])])
  ]);;
```



Les notions de taille et profondeur existent toujours sur les arbres n-aires. On peut les programmer en Caml à l'aide de deux fonctions mutuellement récursives :

```
let rec taille arbre =
  match arbre with
  | Vide -> 0
  | Noeud (_, fils) -> 1 + taille_foret fils
and taille_foret foret =
  match foret with
  | [] -> 0
  | arbre :: autres -> (taille arbre) + (taille_foret autres)
;;
```

EXERCICE 5

Écrire la fonction `hauteur : 'a arbre -> int`. On pourra définir la fonction mutuellement récursive `max_hauteur_foret : 'a arbre list -> int`.

De même, les parcours préfixe et suffixe (mais pas infixe) peuvent être définis sur les arbres n-aires. Par exemple :

```
let rec parcours_prefixe arbre =
  match arbre with
  | Vide -> ()
  | Noeud (etiquette, fils) ->
     print_int etiquette;
     parcours_foret fils
and parcours_foret foret =
  match foret with
  | [] -> ()
  | arbre :: autres ->
     parcours_prefixe arbre;
     parcours_foret autres
;;
```

EXERCICE 6

Écrire de même le parcours suffixe d'un arbre `parcours_suffixe : int arbre -> unit`.

EXERCICE 7

Écrire les fonctions `parcours_prefixe : int arbre -> int list` et `parcours_suffixe : int arbre -> int list` qui renvoient les listes des nœuds parcours.

EXERCICE 8

Écrire une fonction `degre : 'a arbre -> int` qui donne le degré d'un arbre, c'est-à-dire le plus grand degré de ses nœuds.

Si on suppose que l'on a pas besoin de l'arbre vide (ce qui est assez naturel dans certaines études) on peut se passer du constructeur de type. Pour bien insister sur le fait que l'on manipule un arbre et non un couple quelconque, on propose le type suivant :

```
type 'a arbre = {mutable etiquette : 'a; fils : 'a arbre list};;
```

On a de plus autorisé les étiquettes des nœuds à être modifiées.

EXERCICE 9

Réécrire les fonctions `taille` et `hauteur` avec ce nouveau type.

EXERCICE 10

Écrire une fonction `reinitialiser : int arbre -> unit` qui met à 0 toutes les étiquettes d'un arbre.

EXERCICE 11

Peut-on écrire une fonction qui renverse l'ordre des fils de chaque nœud d'un arbre (du *même* arbre, sans en recréer un nouveau) ? Pourquoi ? Modifier ce qui doit l'être et implémentez cette fonction.

3 Piles et files

Les piles (*stack* en anglais) et les files (*queue* en anglais) sont deux autres exemples de structures de données fondamentales en informatique. Il faut bien les connaître. Vous reverrez les piles en cours de tronc commun d'informatique l'année prochaine.

Ces deux ces deux structures de données diffèrent par l'ordre d'accès aux éléments :

- Dans une *pile*, la valeur extraite est la dernière valeur à avoir été placée dans la pile. On parle de structure LIFO (*Last In First Out*).
- Dans une *file*, l'élément que l'on retire de la file est le premier à y être entré. On parle de structure structure FIFO (*First In First Out*).

QUESTION 3

Faire un shéma et donner deux ou trois exemples[4] d'utilisation de ces structures dans la vie courante.

Avant d'implémenter de plusieurs manières ces structures, nous allons les utiliser. Remarquez qu'il n'y a même pas besoin d'avoir implémenté les structures pour pouvoir définir les fonctions qui les utilisent, à condition d'avoir bien spécifié leur signature (type et opérations valables). Dans ce TP, nous allons utiliser les modules[5] `stack` et `queue` implémentés en CAML.

La signature et les spécifications de ces structures sont disponibles aux adresses suivantes :

- Piles : `http://caml.inria.fr/pub/docs/manual-caml-light/node15.18.html`
- Files : `http://caml.inria.fr/pub/docs/manual-caml-light/node15.14.html`

Pour utiliser une fonction d'un module, il suffit de préfixer son nom par le nom du module suivi de deux tirets-bas _ _ :

```
let pile = stack__new ();;
for i = 0 to 9 do stack__push i pile done;;
for i = 0 to 9 do print_int (stack__pop pile) done;;
(* Prévoir le résultat *)

let file = queue__new ();;
for 0 = 1 to 9 do queue__add i file done;;
for 0 = 1 to 9 do print_int (queue__take file) done;;
(* Prévoir le résultat *)
let _ = queue__take file;;
(* Prévoir le résultat *)
```

4. Originaux et drôles si possible — et dans ce cas appelez nous pour les partager.

5. On ne peut pas utiliser directement ces modules dans un écrit de concours — sauf indication contraire, mais bien les connaître peut se révéler très utile pour l'épreuve de programmation des ENS.

Remarque 4. *On peut tester si une pile/file est vide en vérifiant que sa longueur est nulle. Ici, cela n'est pas en $O(1)$, en* CAML *on utiliserait plutôt un système de rattrapage d'exception qui est hors-programme. On utilsera donc les fonctions :*

```
let stack_empty pile = stack__length pile = 0;;
let queue_empty file = queue__length file = 0;;
```

comme si elles étaient en $O(1)$ et nous aurons prochainement nos propres implémentations de piles et files.

EXERCICE 12 *À vous de jouer*

Bien prendre connaissance de ces deux modules et faire quelques test.

EXERCICE 13 *Copies*

On dispose d'une pile de copies de deuxième année, triées par noms d'élèves, qui proviennent de MP ou de MP*. Comment procéder pour transformer cette pile en une pile dans laquel toutes les copies de MP sont situées au dessus de celles de MP*, en conservant l'ordre relatif des copies, c'est-à-dire que les copies de MP et de MP* sont toujours bien triées (sans les trier à nouveau, évidemment) ?

On propose les types :

```
type classe = MP | MPstar;;
type copie = {nom : string; note : int; classe : classe};;
```

Écrire une fonction `separe : copie stack__t -> unit` qui réordonne une pile de copies suivant ce principe.

EXERCICE 14 *Rotations*

Écrire une fonction de type `'a stack__t` qui réalise la rotation d'une pile, c'est-à-dire que l'élément au sommet se retrouve tout en bas de la pile.

$$\begin{vmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{vmatrix} \longrightarrow \begin{vmatrix} a_1 \\ \vdots \\ a_n \\ a_0 \end{vmatrix}$$

On pourra utiliser une pile auxiliaire.

Quelle est sa complexité ? Écrire la fonction qui réalise la rotation inverse. Faire de même avec une file au lieu d'une pile, sans utiliser la fonction `queue__length` en dehors de `queue_empty`.

EXERCICE 15 *Mots bien parenthésés*

On considère des mots sur l'alphabet $\{(,[,),]\}$. Par exemple $(||((]($ et $)[()]$ sont deux mots. On représente les mots par une liste de caractères (`type mot == char list;;`). Le mot vide ne comportant aucune lettre est notée ε. Il est donc représenté par la liste vide. On dit qu'un mot est *bien parenthésé* s'il peut être réduit au mot vide en supprimant successivement des paires adjacentes de parenthèses de même type. Par exemple le mot $([()[]])()$ est bien parenthésé car :

$$([()[]])() \to ([[]])() \to ([])() \to ()() \to () \to \varepsilon$$

Écrire une fonction `est_bien_parenthese : mot -> bool` qui vérifie si un mot est bien parenthésé.