Chapitre 7. Création de nouveaux types en OCaml

I Introduction

On sait que OCaml est un langage fortement typé, ce qui est certes contraignant dans l'écriture des programmes (le compilateur ou l'interpréteur est intransigeant...), mais ce qui surtout apporte une plus grande certitude quand à la validité des programmes écrits.

OCaml présente un certain nombre de types natifs : entier, flottant, booléen, caractère, chaîne de caractères. Il autorise surtout la création de types supplémentaires. Certains sont déjà présents dans des modules : List, Array. Mais on est libre de définir en plus soit même de nouveaux types qui nous sont propres (quitte à en faire des modules plus tard).

Comme on le verra en cours, l'intérêt est d'alors de proposer un ensemble de fonctionnalités relatifs à un nouveau type, ce qui permet là aussi une programmation en général plus propre et sure. On dit alors qu'on programme une **structure de données**.

À notre niveau on va distinguer essentiellement les types **synonymes**, les types **sommes** et les types **produits** (ou **enregistrements**). Toutes ces définitions de type utiliseront le mot clé `type`

Code Caml 1

```
type nouveau_type = ...
```

Nous allons maintenant présenter la syntaxe associée à ces trois possibilités de définition de nouveaux types. En cours nous étudierons de nouvelles structures de données que l'on implémentera de différentes façons en OCaml en utilisant ce mécanisme de création de nouveaux types.

II Type synonyme

On utilise une déclaration de la forme

```
type nouveau_type = type_deja_defini ;;
```

Par exemple, si on veut franciser OCaml ... :

```
type entier = int ;;
type entier = int
```

Testons voir les conséquences sur une fonction calculant sur les entiers :

```
let succ n = n + 1 ;;
val succ : int -> int = <fun>
```

OCaml ne reconnait pas le nouveau type automatiquement ! Comment pourrait-il choisir d'ailleurs entre les différents types...

On peut alors forcer le type d'un paramètre en le faisant suivre de deux points et du nom du type, le tout entre parenthèses. Ce mécanisme est général et on peut s'en servir partout, même d'ailleurs quand cela n'est pas nécessaire, plutôt à fins documentaires dans ce cas...

```
let succ (n : entier) = (n + 1 : entier) ;;
succ : entier -> entier = <fun>
```

Noter dans l'exemple précédent que l'on précise également le type du résultat, de la même façon.

III Type somme

Dans un type somme on décrit l'ensemble des "valeurs" possibles à l'aide de **Constructeurs**, qui commencent par une majuscule, i.e. on énumère les différentes valeurs possibles pour le type. On distinguera le cas des énumérations finies, et des énumérations infinies.

- énumération finie : constructeur avec une Majuscule ; on parle ici de **constructeur constant**.

```
type carte = Roi | Dame | Valet ;;
type carte = Roi | Dame | Valet ;;
```

- énumération "infinie" : constructeur avec Majuscule avec un paramètre de type

```
type carte = Roi | Dame | Valet | Nombre of int;;
type carte = Roi | Dame | Valet | Nombre of int;;
```

On peut alors facilement créer une instance du type somme, en précisant sa valeur. Noter la réponse de l'interpréteur qui précise bien le type et la valeur,

```
let bonne_carte = Roi ;;
val bonne_carte : carte = Roi
let carte_pourrie = Nombre 2 ;;
val carte_pourrie : carte = Nombre 2
```

Le fait qu'il s'agisse de **Constructeurs** est important car comme on l'a vu en cours en début d'année, on pourra les utiliser pour faire du filtrage **(je rappelle qu'on ne peut pas faire de calculs dans un motif de filtrage, mais on peut utiliser des constructeurs) !**

```
let evaluation c = match c with
  | Roi | Dame | Valet -> print_string "Bonne carte"
  | Nombre (v) -> print_int v; print_string ", c'est pourri" ;;
val evaluation : carte -> unit = <fun>
```

Dont l'exécution donne :

```
evaluation bonne_carte ;;
Bonne carte- : unit = ()
evaluation carte_pourrie ;;
2, c'est pourri- : unit = ()
```

IV Type produit (ou enregistrement)

Cette notion correspond plus à la notion de produit d'ensembles (d'où son nom...). On peut voir cela comme des n-uplets auquels on donnerait un nom pour chaque composante pour faciliter les manipulations. La syntaxe est la suivante :

```
type nouveau_type = { nom_1 : type_1; nom_2 : type_2; ... ; nom_n :
    type_n}
type etudiant = {nom : string; id : int }
type etudiant = {nom : string; id : int }
```

À la création on donne les valeurs des champs (**TOUS !**) avec des =

```
let bill = {nom = "Gates" ; id = 1 }
val bill : etudiant = {nom = "Gates" ; id = 1 }
```

On accède à la valeur d'un champ par `.nom_du_champ` :

```
bill.nom ;;
- : string = "Gates"
```

Pour modifier la valeur on utilise `<-`

```
bill.nom <- "Clinton" ;;
Error: The record field nom is not mutable
```

Sauf que par défaut les champs sont non mutables (pas modifiables !). Si on veut modifier un champ il faut le spécifier `mutable` dans la définition du type :

```
type etudiant = {mutable nom : string; id : int }
type etudiant = {mutable nom : string; id : int }
let bill = { nom = "Gates"; id = 1 } ;;
bill.nom ;;
- : string = "Gates"
bill.nom <- "Clinton" ;;
- : unit = ()"
bill.nom ;;
- : string = "Clinton"
```

V Quelques précisions

V.1 Polymorphisme

Le polymophisme est bien sûr supporté ! Par exemple si on veut implémenter concrètement une pile (qui sera vue en cours) à l'aide d'un tableau, on peut définir le type pile contenant des objets de type 'a par :

```
type 'a ma_pile = { tab : 'a array } ;;
type 'a ma_pile = { tab : 'a array }
```

Ici on a créé un type enregistrement contenant un unique champ `tab` , de type `'a array'` , ce qui permet de définir un type polymorphe `'a ma_pile` .

V.2 Types récursifs

La définition générale du concept de récursivité s'applique également aux types (et vous en ferez un grand usage l'année prochaine). Un type peut être récursif, en faisant référence à lui-même.

Il n'y a pas besoin du mot clé `rec` car par défaut les types sont récursifs Par exemple on a vu qu'une liste était soit la liste vide, soit un couple avec une valeur (tête) et une liste (queue) (d'où la définition récursive). On peut traduire cela en OCaml par

```
type 'a ma_liste = Vide | Cons of 'a * 'a ma_liste ;;
type 'a ma_liste = Vide | Cons of 'a * 'a ma_liste
```

On a ici un type somme, avec le constructeur constant Vide, et le constructeur Cons qui prend des objets de type `'a * 'a ma_liste` (le type est bien défini récursivement).

Voici un exemple de programmation utilisant ce type : conversion d'une liste OCaml (`'a list`) en son équivalent `'a ma_list` et un exemple de conversion.

```
let rec conversion_liste_maliste l =
  match l with
  | [] -> Vide
  | t :: q -> Cons (t, conversion_liste_maliste q) ;;

val conversion_liste_maliste : 'a list -> 'a ma_liste = <fun>

conversion_liste_maliste [1; 2; 3; 4]
- : int ma_liste = Cons (1, Cons (2, Cons (3, Cons (4, Vide))))
```

Assurez-vous d'avoir bien compris la forme de la conversion.