Création de types en Caml

Cours – TP

I Types somme

I.1 Exemples

Le mot "somme" dans ce contexte désigne une réunion disjointe, et se note |. Chaque terme de la réunion disjointe est introduit par un *constructeur* (qui peut avoir 0, 1 ou plusieurs arguments) dont le nom commence par une majuscule. Le *pattern-matching* s'applique précisément aux types somme (on filtre suivant le constructeur utilisé pour définir l'élément ayant le type somme).

Exemple 1 : Le type `couleur` suivant a trois constructeurs qui n'ont aucun argument.

```
type couleur = Bleu | Rouge | Vert;;
```

Exemple 2 : Le type `nombre` suivant a deux constructeurs qui ont chacun un argument.

```
type nombre = Entier of int | Flottant of float;;
```

Exemple 3 : Le type `'a option` existe déjà en Caml. Il consiste à réunir le type `'a` avec un avatar de `unit`.

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

I.2 Travaux pratiques

Exercice 1. Écrire une fonction `somme : nombre → nombre→ nombre` pertinente .

II Types inductifs

Dans le contexte des types, le mot inductif signifie récursif.

II.1 Trois exemples

Le entiers naturels sont l'exemple-type d'ensemble inductif.

On peut en faire un type, qu'on appellera le type des naturels abstraits :

Un dessin.

Exemple 4 : Le type `nat` :

```
type nat = O | S of nat;;
```

On connaît un autre type inductif : le type (polymorphe) `'a list`. Redéfinissons-le !

Exemple 5 : Le type `'a liste` :

Un dessin.

```
type 'a liste = Vide | Cons of 'a*('a liste);;
```

On va prochainement travailler sur la version bidimentionnelle des `'a list` : les `'a arbre`.

Exemple 6 : Le type `'a arbre` :

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

Un dessin :

Un terme d'un type inductif a naturellement une représentation arborescente ; on les a fait figurer à côté des définitions.

II.2 Travaux pratiques

Exercice 2. Écrire une fonction qui convertit un naturel abstrait en entier Caml.
Écrire aussi sa réciproque, en provoquant une exception si l'argument est négatif.

Exercice 3. Écrire une fonction qui affiche joliement les éléments d'une `int liste`.

II.3 Induction structurelle

Théorème 1 : Induction structurelle

Soit $P(x)$ une propriété portant sur un terme x dont le type est un type inductif. Si

1. la propriété $P(x)$ est vraie pour tout terme de base du type (un terme obtenu à l'aide d'un constructeur sans argument) ;
2. la propriété $P(x)$ est vraie pour tout terme x obtenu à l'aide d'un constructeur ayant plusieurs arguments **si l'on suppose qu'elle est vraie pour ses arguments** ;

alors la propriété $P(x)$ est vraie pour tout terme x du type inductif.

Remarque 1 : Le point 1. est en fait un cas particulier du point 2.

Exercice 4. Dans le cas des naturels abstraits, on retrouve .....

Remarque 2 : On peut de même écrire des **définitions inductives**, c'est ce qu'on fait en Caml en combinant pattern-matching et récursivité.

III Types alias.

III.1 Exemples

C'est un type existant mais auquel on attribue un nouveau nom avec la syntaxe `type ... = ...`

Exemple 7 : On peut définir le type `mot` des listes de caractères :

```
type mot = char list ;;
```

On peut imposer le type d'une fonction définie sur ou à valeurs dans un type alias.
Un exemple avec la fonction qui calcule le miroir d'un mot :

```
let miroir (m : mot) : mot = List.rev m;;
```

Un type peut être polymorphe, c'est-à-dire être paramétré par un ou plusieurs autres types.

Exemple 8 : On peut définir le type `'a biglist` des listes de listes d'éléments de type 'a :

```
type 'a biglist = 'a list list ;;
```

III.2 Travaux pratiques

Exercice 5. Écrire les fonctions `string_of_mot : mot` $\to$ `string` et `mot_of_string : string` $\to$ `mot` réalisant les conversions d'un mot en chaine de caractère et inversement, en forcant leur types.

IV Types produits et enregistrements

IV.1 Exemples

Types produits :
Si `'a` et `'b` sont deux types alors `'a*'b` est le type des couples `(x,y)` avec `x` de type `'a` et `y` de type `'b`.

Remarque 3 : On peut définir de même de types produits avec $n \geqslant 3$ facteurs plutôt que 2.

Exemple 9 : On peut définir le type `complexe` comme un alias du type `float*float`

Types enregistements :
C'est le même principe mais en permettant de *nommer* les différents éléments du couple (ou du n-uplet). Ainsi pour les complexes, il est plus agéable de pouvoir clairement spécifier ce qu'est la partie réelle et ce qu'est la partie imaginaire.

Exemple 10 :
```
type complexe = {re : float ; im : float};;
```

IV.2 Travaux pratiques

Exercice 6. Définir le nombre complexe **i**, et les fonctions somme et produit sur les complexes.

IV.3 Mutabilité

Par défaut, un type enregistement est un type *immuable*. On ne peut pas modifier les champs des enregistements d'un même élément après l'avoir défini, on peut seulement créer un nouvel élément avec des champs différents (de ce point de vue, les types enregistrements se comportent comme les listes et pas comme les tableaux).

Il est néanmoins possible d'y remédier en spécifiant que les champs concernés peuvent être mutables.

Exemple 11 :
```
type cplx = {mutable re : float ; mutable im : float};;
```

Exercice 7. À l'aide d'une boucle, écrire une fonction qui prend un argument un complexe z_0 et un entier n, et retourne le $n^{\text{ième}}$ terme de la suite $(z_n)_{n\in\mathbb{N}}$ vérifiant $\forall n \in \mathbb{N},\ z_{n+1} = \dfrac{1}{2}(z_n + |z_n|)$.