TYPES CONSTRUITS EN CAML

Le langage CAML permet de construire des types à partir des types de bases (entiers, flottants, listes, tableaux, etc). Deux catégories de types sont distinguées : les types produits et les types sommes.

1 Types produits

Les types produits sont ceux qui correspondent aux produits cartésiens d'ensembles. Le plus simple des types produits est le produit cartésien de types, c'est-à-dire les couples, triplets, etc.

1.1 n-uplets

On construit un n-uplet par le constructeur ,, les parenthèses étant facultatives dans la plupart des cas.

```
let tpl = 1, "mot", [-2; 3];;
let x =
  if tpl = (1, "mot", []) then 0
  else 1;;
```

construit un triplet de type `int * string * int list`

On peut filtrer un n-uplet pour récupérer ses composantes :

```
let a, b, c = tpl;;
let _, d, _ = tpl;;
let e = match tpl with | _, _, x -> x;;
```

Pour les couples, on dispose de deux fonctions prédéfinies fst et snd dont la définition est :

```
let fst (a,b) = a;;
let snd (a,b) = b;;
```

1.2 Enregistrements

Un enregistrement est une sorte de n-uplet, dont les composantes portent un nom et qui ont un type spécifié lors de la déclaration du type.

```
(* définition du type fiche : enregistrement à 2 champs *)
type fiche = {nom : string; ine : int};;

(* creation d'une fiche qui représente un étudiant *)
let f = {nom = "toto"; ine = 237 };;

(* une fiche étant connue, on peut récupérer ses champs *)
(* par la notation pointée *)
let name = f.nom and id = f.ine;;
(* par filtrage *)
let nom f =
  match f with
  | { nom = name; ine = _ } -> name;;
```

2 Types sommes

Les types sommes sont ceux qui correspondent aux réunions d'ensembles. On les définit par des constructeurs, dont le nom commencent toujours par une majuscule.

2.1 types énumérés simples

Les types énumérés définissent de nouvelles constantes symboliques en nombre fini : ils correspondent mathématiquement aux ensembles finis.

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

Une fois définies, ces constantes symboliques sont utilisables comme toutes les autres.

```
let couleur s =
  match s with
  | Trefle | Pique -> "noir"
  | Carreau | Coeur -> "rouge";;
```

2.2 types sommes généraux

Les types sommes peuvent accepter des ensembles infinis. Dans l'exemple qui suit, on définit un type nombre qui permet de réunir les entiers et les flottants.

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

let n = Entier(12);; let r = Reel(1.5);;
let liste = [n; r];;

let add m n =
  match m, n with
  | Entier(a), Entier(b) -> Entier(a + b)
  | Entier(a), Reel(r) -> Reel(r +. float_of_int a)
  | Reel(r), Entier(a) -> Reel(r +. float_of_int a)
  | Reel(r), Reel(s) -> Reel(r +. s);;
```

Dans l'exemple qui suit, on ajoute les deux infinis aux réels et un symbole spécial.

```
type limites = Reel of float | Pinf | Minf | Indet;;

let somme l l' =
  match l, l' with
  | Reel(r), Reel(s) -> Reel(r +. s)
  | Pinf, Minf | Minf, Pinf -> Indet
  | Pinf, _ | _, Pinf -> Pinf
  | Minf, _ | _, Minf -> Minf
  | _ -> Indet;;
```

3 Types polymorphes

Le polymorphisme d'un type construit est sa faculté de pouvoir être utilisable avec des types internes différents. Les listes, les tableaux sont des types polymorphes : on peut mettre ce qu'on veut dans une liste, à condition que les types des objets soient les mêmes. Lors de la compilation, CAML montre le polymorphisme avec les pseudo-types 'a, 'b, ...

Les types précédents ne sont pas polymorphes : leur signature est fixée une fois pour toute lors de la déclaration du type. Pour permettre plus de souplesse, on peut introduire des variables de types, qui ajoutent du polymorphisme : il suffit de les ajouter dans la déclaration du type.

```
(* type fiche générique *)
type 'a fiche = {nom : string; donnee : 'a};;

(* type  int fiche *)
let f = {nom = "toto"; donnee = 237};;
(* type  string fiche *)
let g = {nom = "titi"; donnee = "oiseau"};;
```

4 Types récursifs

Les types construits en CAML sont par défaut récursifs. On peut donc déclarer des ensembles définis par induction structurelle.