II Types produits et sommes

26 janvier 2024

Nous n'avons jusque là manipulé que des types déjà existant. Mais se restreindre à des types prédéfinis peut être contraignant. Si vous voulez coder une messagerie en OCaml, vous pouvez stocker les messages dans une chaîne de caractères. Mais comment y adjoindre les noms de l'expéditeur et des destinataires, l'heure et la date d'envoi, des étiquettes "important" ou "indésirable" etc ? Le tout de manière efficace sans utiliser de n-uplets à 15000 coordonnées ? Heureusement OCaml permet de construire de nouveaux types, sur mesure. Nous allons en voir trois catégories : les types synonymes, les types produits et les types sommes.

1 Types produits

1.1 Types synonymes

Le premier type de types que nous voyons est celui des *types synonymes*. L'idée est très simple : on donne un nouveau nom à un type déjà existant. L'intérêt est juste de bien "ranger" les différentzs objets d'un programme afin de le clarifier.

Par exemple, en géométrie il est souvent important de distinguer les points et les vecteurs. Mais les deux se représentent comme des couples de flottants. Il est peut donc être intéressant de créer un type `point` et un type `vecteur`.

[1] :
```
type point = float * float  ; ;
type vecteur = float * float  ; ;
```

[1] :
```
type point = float * float
type vecteur = float * float
```

Par exemple, nous pouvons utiliser ces types pour définir une fonction qui à un vecteur `v` et un point `p` associe le translaté de `p` par `v`.

[2] :
```
let translation (v :vecteur) (p :point)  : point = (fst v +. fst p,snd v+.␣
↪snd p) ; ;
translation (2.,3.) (1.,1.) ; ;
```

[2] :
```
val translation  : vecteur -> point -> point = <fun>
-  : point = (3., 4.)
```

ou encore

[3] :
```
let translation (v :vecteur) (p :point)  : point =
    match (v,p) with
    |(a,b),(x,y) -> (x+.a,y+.b)
 ; ;
```

```
[3] : val translation  : vecteur -> point -> point = <fun>
```

Nous avons ici créé deux types qui représentent des produits cartésiens. Mais nous voyons que l'accès aux différentes coordonnées est pénible. Nous avons ici utilisé les fonctions `fst`et `snd` (mais elles ne s'appliquent qu'à des couples), ou un match, mais qui fait perdre la lisibilité gagnée par ailleurs. Si nous voulions manipuler des n-uplets plus grands, la difficulté serait encore plus visible. Les types synonymes sont donc plutôt limités. Nous allons avoir besoin d'autres types.

1.2 Enregistrements

Il existe une autre catégorie de types produits en OCaml : les *enregistrements*. Alors que dans un n-uplets, les différentes informations ne sont repérées que par leur position, dans un enregistrement, chaque composante possède un nom.

Nous souhaitons par exemple représenter un étudiant par son nom, son prénom et sa date de naissance. On peut utiliser un quintuplet constitué de deux chaînes de caractères (nom et prénom) et de trois entiers (année, mois et jour de naissance).

Il est possible de définir des nouveaux *types enregistrement*, par exemple un type `date` contenant trois *champs* (ses composantes) de type `int` : un jour, un mois et une année.

```
[4] : type date = {
        jour  : int ;
        mois  : int ;
        annee  : int ;
      } ; ;
```

```
[4] : type date = { jour  : int ; mois  : int ; annee  : int ; }
```

Pour construire une date, par exemple le 14 mars 2022 :

```
[5] : let d = {annee = 2022 ; mois = 3 ; jour = 14 ;} ; ;
```

```
[5] : val d  : date = {jour = 14 ; mois = 3 ; annee = 2022}
```

On remarque qu'il n'est pas nécessaire de donner les champs dans le même ordre que dans la définition.

Ce type `date` peut être utilisé pour construire le type `etudiant` :

```
[6] : type etudiant = {
        nom  : string ;
        prenom  : string ;
        naissance  : date ;
      } ; ;
```

```
[6] : type etudiant = { nom  : string ; prenom  : string ; naissance  : date ; }
```

```
[7] : let e = {
        nom = "Tournesol" ;
        prenom = "Tryphon" ;
        naissance = {
            jour = 28  ;
```

```
        mois = 1 ;
        annee = 2006 ;
      }
  } ; ;
```

```
[7] : val e  : etudiant =
        {nom = "Tournesol" ; prenom = "Tryphon" ;
         naissance = {jour = 28 ; mois = 1 ; annee = 2006}}
```

Pour accéder à l'un des champs d'un objet de type enregistrement, il suffit de faire suivre le nom de l'objet d'un point puis du nom du champ.

```
[8] : e.naissance ; ;
```

```
[8] : -  : date = {jour = 28 ; mois = 1 ; annee = 2006}
```

```
[9] : let est_majeur e date =
        let n = e.naissance in
        let j = n.jour and m = n.mois and a = n.annee in
        let age = date.annee - a - (if m > date.mois || (m = date.mois && j > date.
      ↪jour) then 1 else 0) in
        age >= 18
      ; ;
```

```
[9] : val est_majeur  : etudiant -> date -> bool = <fun>
```

```
[10] : let aujourdhui = {jour = 29  ; mois = 1 ; annee = 2024} in
       est_majeur e aujourdhui ; ;
```

```
[10] : -  : bool = true
```

Le filtrage par cas est toujours aussi pertinent :

```
[11] : let nom_commence_par_T e =
         match e.nom with
         | s -> s.[0] = 'T'
       ; ;
       nom_commence_par_T e ; ;
```

```
[11] : val nom_commence_par_T  : etudiant -> bool = <fun>
       -  : bool = true
```

2 Types sommes

2.1 Types énumérations

Jusqu'à présent, toutes les données que nous pouvons représenter reposent sur les types standards. Comment représenter alors les jours de la semaine ?

— Par un entier : quelle convention choisir ? Numérote-t-on de 0 à 6 ou de 1 à 7 ? En commençant par quel jour ?

— Ou par une chaîne de caractères.

Dans les deux cas, si on utilise une donnée qui ne correspond à aucun jour (par exemple en faisant une faute de frappe dans une chaîne de caractères), cela ne sera détecté qu'à l'exécution.

On peut alors utiliser un type *énumération*, en donnant toutes les valeurs possibles :

[12] :
```
type jour_semaine =
  | Lundi
  | Mardi
  | Mercredi
  | Jeudi
  | Vendredi
  | Samedi
  | Dimanche
;;
```

[12] :
```
type jour_semaine =
    Lundi
  | Mardi
  | Mercredi
  | Jeudi
  | Vendredi
  | Samedi
  | Dimanche
```

Les valeurs `Lundi` ,... sont appelées les *constructeurs* du type `jour_semaine` . Ce sont en effet les seules opérations permettant de construire des objets du type `jour_semaine` . Les constructeurs doivent commencer par une majuscule, afin de ne pas les confondre avec des variables (qui elles commenceront toujours par des minuscules).

[13] :
```
Lundi ;;
```

[13] :
```
-  : jour_semaine = Lundi
```

[14] :
```
Mardi ;;
```

[14] :
```
-  : jour_semaine = Mardi
```

On peut réaliser un filtrage sur les valeurs du type `jour_semaine`

[15] :
```
let week_end j =
  match j with
  | Samedi -> true
  | Dimanche -> true
  | _ -> false
;;
```

[15] :
```
val week_end  : jour_semaine -> bool = <fun>
```

[16] :
```
week_end Mardi ;;
```

[16] :
```
-  : bool = false
```

Utiliser un type énumération demande néanmoins que le nombre de valeurs possibles soit fini, comme pour les jours de la semaine ou les couleurs d'une carte à la belote.

[17] :
```
type couleur =
  | Trefle
  | Carreau
  | Coeur
  | Pique
 ; ;
```

[17] :
```
type couleur = Trefle | Carreau | Coeur | Pique
```

2.2 Types sommes

Il s'agit d'une généralisation des types énumérations. Supposons que nous souhaitions représenter les cartes d'un jeu de 32 cartes. Chaque carte est alors identifiée par sa couleur (Trèfle, Carreau, Cœur, Pique) et sa valeur (As, Roi, Dame, valet, 10, 9, 8, 7).

[18] :
```
type carte =
  | As of couleur
  | Roi of couleur
  | Dame of couleur
  | Valet of couleur
  | Petite_carte of int * couleur
 ; ;
```

[18] :
```
type carte =
    As of couleur
  | Roi of couleur
  | Dame of couleur
  | Valet of couleur
  | Petite_carte of int * couleur
```

Comme précédemment, `As` , `Petite_carte` sont des constructeurs :

[19] :
```
let ma_carte = Valet Trefle ; ;
```

[19] :
```
val ma_carte  : carte = Valet Trefle
```

[20] :
```
Petite_carte (9, Pique) ; ;
```

[20] :
```
-  : carte = Petite_carte (9, Pique)
```

Il est possible d'effectuer un filtrage sur ces valeurs, par exemple pour déterminer ici les valeurs des cartes à la belote, qui dépend de la couleur de l'atout :

[21] :
```
let valeur_carte atout carte =
  match carte with
  | As _ -> 11
  | Roi _ -> 4
  | Dame _ -> 3
  | Valet c -> if c = atout then 20 else 2
```

```
  | Petite_carte (10, _) -> 10
  | Petite_carte (9, c) -> if c = atout then 14 else 0
  | _ -> 0
;;
```

```
[21] : val valeur_carte  : couleur -> carte -> int = <fun>
```

```
[22] : let toto = Valet Coeur in
valeur_carte Coeur toto ;;
```

```
[22] : -  : int = 20
```

Un type somme peut être *récursif*, c'est à dire intervenir dans sa propre définition. Définissons par exemple les couleurs par synthèse soustractive, en considérant qu'une couleur est soit une couleur primaire, soit un mélange de deux couleurs :

```
[23] : type color =
  | Cyan
  | Magenta
  | Jaune
  | Melange of color * color ;;
```

```
[23] : type color = Cyan | Magenta | Jaune | Melange of color * color
```

```
[24] : let rouge = Melange (Magenta, Jaune) ;;
```

```
[24] : val rouge  : color = Melange (Magenta, Jaune)
```

```
[25] : let orange = Melange (rouge, Jaune) ;;
```

```
[25] : val orange  : color = Melange (Melange (Magenta, Jaune), Jaune)
```

3 Exercices divers

3.1 Exercice 1

— Définir un type enregistrement `complexe` associé aux nombres complexes.

— Écrire une fonction `module_complexe  : complexe -> float` calculant le module d'un nombre complexe.

— Écrire une fonction `produit  : complexe -> complexe -> complexe` calculant le produit de deux nombres complexes.

3.2 Exercice 2

— Définir un type somme `reel_etendu` permettant de représenter la droite numérique achevée.

— Écrire une fonction `etendu_of_float` permettant de convertir un nombre de type `float` en un nombre de type `reel_etendu` .

— Écrire une fonction `somme : reel_etendu -> reel_etendu -> reel_etendu` calculant (si c'est possible) la somme de deux réels étendus. Dans le cas d'une forme indéterminée, on pourra utiliser la syntaxe `failwith "forme indéterminée"`.