Types produits et types sommes

1 Types produits

1.1 Produits cartésiens

On peut, si on le souhaite, définir un type correspondant à un produit cartésien :

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

Cela pourra être utile lorsque nous définirons d'autres types (voir ci-dessous), mais OCaml n'utilisera pas ce type (synonyme de `float*float`) par défaut :

```
# let module_complexe z =
  match z with
  | (a, b) -> sqrt(a**2. +. b**2.);;
    val module_complexe : float * float -> float = <fun>
```

1.2 Enregistrements

Lorsqu'on souhaite représenter une entité décrite par un ensemble d'informations, par exemple représenter un étudiant par son nom, son prénom et sa date de naissance, on peut utiliser un n-uplet, dans notre exemple un quintuplet constitué de deux chaînes de caractères (nom et prénom) et de trois entiers (année, mois et jour de naissance).

Manipuler les informations liées à ces objets est alors assez délicat : il faut gérer l'ordre dans lequel les informations sont données.

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.

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

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

Pour construire une date, par exemple le 14 mars 2019 :

```
# let d = {annee = 2019; mois = 3; jour = 14;};;
val d : date = {jour = 14; mois = 3; annee = 2019}
```

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` :

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


let e = {
    nom = "Tournesol";
    prenom = "Tryphon";
    naissance = {
        jour = 28 ;
        mois = 1;
        annee = 2000;
      }
  };;
```

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.

```
let est_majeur e =
  let n = e.naissance in
  let j = n.jour and m = n.mois and a = n.annee in
  let age = 2019 - a -
              (if m > 2 || (m = 2 && j > 5) then 1 else 0) in
  age >= 18
;;
```

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 :

```
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.

```
# Lundi;;
- : jour_semaine = Lundi
# Mardi;;
- : jour_semaine = Mardi
```

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

```
let week_end j =
  match j with
  | Samedi -> true
  | Dimanche -> true
  | _ -> 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.

```
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).

```
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 :

```
# As Trefle;;
- : carte = As Trefle
# Petite_carte (9, Pique);;
- : carte = Petite_carte (9, Pique)
```

Il est possible d'effectuer un filtrage sur ces valeurs :

```
let valeur_carte atout = fun 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
;;
```

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 :

```
# type color =
  | Cyan
  | Magenta
  | Jaune
  | Melange of color * color;;
       type color = Cyan | Magenta | Jaune | Melange of color *
color
# let rouge = Melange (Magenta, Jaune);;
val rouge : color = Melange (Magenta, Jaune)
# let orange = Melange (rouge, Jaune);;
val orange : color = Melange (Melange (Magenta, Jaune), Jaune)
```

3 Exercices divers

Exercice 1

1. *Définir un type enregistrement* `complexe` *associé aux nombres complexes.*
2. *Écrire une fonction* `module : complexe -> float` *calculant le module d'un nombre complexe.*
3. *Écrire une fonction* `produit : complexe -> complexe -> complexe` *calculant le produit de deux nombres complexes.*

Exercice 2

1. *Définir un type somme* `reel_etendu` *permettant de représenter la droite numérique achevée.*
2. *Écrire une fonction* `etendu_of_float` *permettant de convertir un nombre de type* `float` *en un nombre de type* `reel_etendu`.
3. *Écrire une fonction* `somme : reel_etendu -> reel_etendu -> reel_etendu` *calculant (si c'est possible) la somme de deux réels étendus.*