Cours numéro 6 : Les types somme

LI213 – Types et Structures de données

Christophe Gonzales – Pierre-Henri Wuillemin

Licence d'Informatique – Université Paris 6

Introduction aux types somme

On sait représenter toute sorte d'agrégations de données. Mais toutes ces agrégations n'utilisent, à la base, que des types standards (`int, float, char, string, ...`).

exemple : Comment représenter correctement le type "jour de la semaine" ? **par un entier ?**

```
# let string_of_jour j = match j with
  | 1 -> "Lundi"
  | 2 -> "Mardi"
  | 3 -> "Mercredi"
  | 4 -> "Jeudi"
  | 5 -> "Vendredi"
  | 6 -> "Samedi"
  | 0 -> "Dimanche" ;;
```

```
Warning : this pattern-matching is not exhaustive.
Here is an example of a value that is not matched : 7
val string_of_jour : int -> string = <fun>
```

Définir un type somme (1/2)

Type somme : Constructeur constant

`type` *nom* `= Cons1 | Cons2 | Cons3 ...;;`

Les `Cons`$_i$ sont nommés «constructeur» et **commence par une majuscule**. Les `Cons`$_i$ sont utilisables comme constantes.

```
# type jour = Lundi | Mardi | Mercredi | Jeudi |
          Vendredi | Samedi | Dimanche;;
type jour = ...
# let j=Lundi;;
val j : jour = Lundi
#  let string_of_jour j = match j with
   | Lundi -> "Lundi"
   | Mardi -> "Mardi"
   ...
   | Dimanche -> "Dimanche";;
val string_of_jour : jour -> string = <fun>
```

Définir un type somme (2/2)

Problème : Représentation d'un jeu de carte.

- Couleurs :

```
type couleur = Coeur | Pique | Carreau | Trèfle;;
```

- Valeurs :

```
type valeur = Roi  |  Reine  |  Valet  |  As  |
              Deux  | Trois  |... | Neuf | Dix;;
```

Peut on faire mieux pour `valeur` qu'énumérer de 1 à 10 ?

Type somme : Constructeur non constant

```
type nom = Cons of type;;
```

- Valeurs :

```
type valeur = Roi  |  Reine  |  Valet  |  Val of int;;
```

```
# let y=Val 5;;
val y : valeur = Val 5
```

Pourquoi type somme ?

On sait que, pour le type produit (pour les n-uplets) :

- Si le type `Tx` a x valeurs possibles
- Et le type `Ty` a y valeurs possibles
- Alors **le type `Tx*Ty` a $x * y$ valeurs possibles**.

Type Somme

- Si le type `Tx` a x valeurs possibles
- Et le type `Ty` a y valeurs possibles
- Alors le type `ConsX of Tx | ConsY of Ty` a $x + y$ valeurs possibles.

```
# type piece = Pile | Face | Tranche;;
```

```
# type bizarre = ConsPiece of piece | ConsBool of bool;;
```

Les valeurs du type possibles `bizarre` sont :

ConsPiece Pile	ConsPiece Face	ConsPiece Tranche
ConsBool true	ConsBool false	

Matching sur un type somme

Le filtrage se présente exactement comme la construction de valeur d'un type somme.

```
# type bizarre = Reel of float | Entier of int;;
```

```
# let x= Entier 4;;
```

```
val x : bizarre = Entier 4
```

```
# let y= Reel 4.3;;
```

```
val y : bizarre = Reel 4.3
```

```
# let ecritBizarre x = match x with
   | Entier i -> print_int i
   | Reel f -> print_float f ;;
```

```
val ecritBizarre : bizarre -> unit = <fun>
```

```
# let addBizarre x y = match (x,y) with
   | (Entier i,Entier j) -> Entier (i+j)
   | (Reel u,Reel v) -> Reel (u +. v)
   | (Reel u,Entier j) -> Reel (u +. (float_of_int j))
   | (Entier i,Reel v) -> Reel ((float_of_int i) +. v) ;;
```

```
val addBizarre : bizarre -> bizarre -> bizarre = <fun>
```

Avantages des types somme

- Représentation de valeur “symbolique”. (`enum` de C).
- Manipulation dans un seul objet de types différents (`union` de C).
- Vérification du type à la compilation (et non à l'exécution comme en C).
- Application à des types complexes nécessaires à OCaml :
  - Types récursifs,
  - Types fonctionnels,
  - Types somme paramétrés.

Types récursifs

Les types récursifs sont nécessaires en OCaml pour décrire des types de données standards : liste, pile, arbre, graphe, ...
Contrairement à la définition de valeur (`let`), la définition de type (`type`) est récursive par défaut :

type produit **type** `monType = type1 * type2;;`

type somme **type** `monType = E1 of type1 | E2 of type2;;`

`type1` ou `type2` peuvent être le même type que `monType`.

```
# type toto = toto * toto;;
```

```
The type abbreviation toto is cyclic
```

Il faut une alternative non récursive (condition d'arrêt) : type somme

```
# type toto = Nil | E of toto * toto;;
type toto = Nil | E of toto * toto
```

Type récursifs et valeurs de types récursifs

Un type récursif permet de définir un type dont l'une des composantes est une valeur du même type.

Liste Linéaire Chaînée

```
type llc = Nil | Cell of int * llc
```

La liste linéaire 3 $\longrightarrow$ 4 $\longrightarrow$ 5 a comme première cellule le couple dont le premier élément est 3 et le second est la liste linéaire 4 $\longrightarrow$ 5.

```
# let l=Cell(3, Cell(4, Cell(5,Nil)));;
```

 On n'est pas obligé d'utiliser l'alternative non récursive dans la définition d'une valeur de type récursif.

```
# let rec ll=Cell(1,Cell(2,(Cell 3,ll)));;
val ll : llc = Cell (1, Cell( 2,Cell(3,...)))
```

Types somme paramétré

Dans la déclaration d'un type, tous les types peuvent être considérés comme des paramètres.

Type paramétré

```
type 'a truc = Rien | Ea of 'a * bool;;
```

Type paramétré récursif

Liste linéaire chaînée contenant 2 types possibles de contenu :
$3 \longrightarrow a \longrightarrow b \longrightarrow 5 \longrightarrow 6$

```
type ('a , 'b) llc_ab = Nil
                      | Ea of 'a * ('a,'b) llc_ab
                      | Eb of 'b * ('a,'b) llc_ab;;
```

```
# let l=Ea(3,Eb('a',Eb('b',Ea(5,Ea(6,Nil)))));;
- : (int, char) llc_ab = ...
```

Types fonctionnels

Dans la déclaration d'un type, tous les types sont permis. Donc aussi les types de fonctions.

Type somme récursif, paramétré et fonctionnel

```
type 'a listf = Val of 'a
              | Fun of ('a -> 'a) * 'a listf
```

Ce type construit des listes dont tous les éléments sauf le dernier sont des valeurs fonctionnelles.

```
# let huit_div = (/) 8;;
val huit_div : int -> int = <fun>
# let gl = Fun (succ , (Fun (huit_div,Val 4)));;
val gl : int listf = Fun (<fun>, Fun (<fun>, Val 4))
```

Types fonctionnels - 2

```
type 'a listf = Val of 'a
              | Fun of ('a -> 'a) * 'a listf;;
let gl = Fun (succ , (Fun (huit_div,Val 4)));;
```

Écrire la fonction `calcule` de signature `'a listf -> 'a`

```
# calcule gl;;
- : int = 3
```

Fonction `calcule`

```
let rec calcule l = match l with
    Val v -> v
  | Fun (f, x) -> f (calcule x);;
```

En utilisant le filtrage explicit des paramètres :

```
let rec compute = function
    Val v -> v
  | Fun(f, x) -> f (compute x);;
```

Exemple d'application : Expressions logiques

Des expressions logiques sont formées à partir de :

1. Deux constantes 0 et 1
2. Des variables logiques $x_0, x_1, ..., x_n$
3. Un opérateur unaire : `Non`
4. Deux opérateurs binaire : `Et` et `Ou`

Représentation par un type somme récursif

```
type exprLogique =
     Vrai | Faux
    | X of int
    | Non of exprLogique
    | Et of exprLogique*exprLogique
    | Ou of exprLogique*exprLogique
```

Exemple d'application : Expressions logiques (2)

Ou exclusif

But : Construire un `ou exclusif` entre 2 propositions.

```
let xor exprA exprB = Ou (
          Et (exprA , Non exprB) ,
          Et (Non exprA , exprB) );;
```

Liste des monômes

But : Construire à partir d'une expression logique, la liste des monômes (x_i ou `Non` x_i) de cette expression.

```
let rec monomes = function
  | Vrai | Faux -> []
  | X i -> [X i]
  | Non(X i) -> [Non(X i)]
  | Non e -> monomes e
  | Et (e1,e2)
  | Ou (e1,e2) ->(monomes e1) @ (monomes e2); ;
```

Exemple d'application : Liste comme OCaml

Les listes en OCaml sont des listes linéaires chaînées simples et homogènes (toutes les cellules sont de même type).

Liste Linéaire Homogène

```
type 'a maListe = Nil
                | Cell of ('a * 'a maListe);;
```

```
# let l = [ 'a'; 'b'; 'c'];;
val l : char list = ['a'; 'b'; 'c']
# let l = Cell('a',Cell('b',Cell('c',Nil)));;
val l : char maListe = Cell ('a', Cell
('b', Cell ('c', Nil)))
```

Spécification 'officielle' des listes OCaml

```
let 'a list = [] | :: of ('a * 'a list)
```

Exemple d'application : Liste comme OCaml (2)

But : compter le nombre d'élément d'une liste
(`List.length l`)

Version liste OCaml

```
let rec maLength = function
    [] -> 0
  | tete :: queue -> 1+(maLength queue)
;;
```

Version maListe

```
let rec maLength = function
    Nil -> 0
  | Cell(tete,queue) -> 1 + (maLength queue)
;;
```