Programmation Fonctionnelle
Cours 4
Plus de types structurés

October 10, 2012

Table de matières

Le type des n-uplets

- Type polymorphe : pour tous types $t_1, \ldots, t_n$, il y a un type de n-uplet $t_1 * \ldots * t_n$.
- Les t_i peuvent être les mêmes ou pas.
- Les n-uplets de types composant différent sont disjoints.
- On parle aussi d'un produit cartésien de types.
- Pas confondre les n-uplets avec les listes !!

Ca sert à quoi les les types produits ?

- Écrire des fonctions qui prennent un n-uplet de valeurs en argument (pas à confondre avec les fonctions à plusieurs arguments !)
- Écrire des fonctions qui "envoient plusieurs résultats" (groupés sous forme d'un n-uplet)
- Utilisé dans les types algébriques (voir plus tard)

Construire est déconstruire des n-uplets

- Construire : combiner les éléments par la virgule (souvent écrit entre parenthèses, mais ce n'est pas strictement nécessaire) : `(1,2.0,"hello")`
- Déconstruire : Il y a des fonctions `fst` et `snd` pour des paires, sinon filtrage par motif ...
- ... ou avec un let. Plus simple que le filtrage par motif car il y a un seul cas (contrairement aux listes, par exemple).

Exemples (tuples1.ml)

```
(2, 3, 4);;
(1, 3.0, "hello");;
let x=(2,3.5);;
fst x;;
snd x;;
match x with
  | (l,r) -> l
;;
let (l,r) = x in l;;
```


Filtrage implicite

- Raccourci pour le cas où il y a un seul cas dans un filtrage par motif.
- Peut être utilisé dans des let, ou des définitions de fonction.

Exemples (tuples2.ml)

```
let z = (3,4);;

match z with (x,y) -> x+y;;
let (x,y) = z in x+y;; (* raccourci *)

let f z = match z with (x,y) -> x+y;;
let f(x,y) = x+y;; (* raccourci *)
```

Exemples (tuples3.ml)

```
(* Une fonction a UN SEUL argument (qui est une pai
let f(x,y) = x+y;;
f(2,3);;
f 2 3;; (* erreur*)

(* Une fonction a deux arguments *)
let g x y = x + y;;
g 2 3;;
g(2,3);; (* erreur *)
```


Exemples (tuples4.ml)

```
(* Fonction qui rend "deux resultats" *)
let rec split pivot liste = match liste with
  | [] -> ([], [])
  | h::r ->
      let (r1,r2) = split pivot r
      in if h<pivot then (h::r1,r2) else (r1,h::r2)

split 17 [4;67;1;13;25];;
```


Exemples (tuples5.ml)

```
(1,2,3);;
(1,(2,3));;
(1,2,3) < (1,(2,3));; (* types differents *)
(1,2,3) < [1;2;3];; (* pas confondre n-uplets et lis
[1,2,3];; (* quel est le type ? *)
1,2,3;;
```


Listes d'association

- Liste de paires
- Très souvent utilisées pour représenter des fonctions à domaine fini.
- On parle aussi de clefs, auxquelles on associe des valeurs (bien que, du point de vue de OCaml, les deux sont des valeurs dans le sens de OCaml).
- Il y a des fonctions utiles dans la bibliothèque `List`.

Exemples (assoclist1.ml)

```
let bureaux = [ ("pierre", 101); ("julie", 103); ("
let rec trouver clef liste = match liste with
  | [] -> failwith ("Clef pas trouvee :"^clef)
  | (c,v)::r when c=clef -> v
  | _::r -> trouver clef r
;;

trouver "julie" bureaux;;
trouver "marie" bureaux;;
```


Exemples (assoclist2.ml)

```
let x = [ ("a",1); ("b",2); ("c",3)] ;;
List.assoc "b" x;;
List.assoc "toto" x;;
List.mem_assoc "a" x;;
List.mem_assoc "titi" x;;
```


Enregistrements

- Anglais : record
- Définition de type obligatoire : associe à un nom de type, dans ce cas, les champs et leur type respectif.
- On ne peut pas utiliser le même nom de champs dans deux types d'enregistrement différents.
- Cela est nécessaire pour avoir une inférence de type efficace.

Exemples (records1.ml)

```
type date = {
  day: int;
  month: string;
  year: int
};;

let today = {
  day = 7;
  month = "october";
  year = 2010;
};;

today.year;;
today.annee;;
let next_week = {today with day=14};;
next_week.day;;
```

Exemples (records2.ml)

```
type r1 =
    {a: string; b: int};;
type r2 =
    {a: string; c: float};;

{a="john"; b=17};; (* error *)
{a="john"; c=17.3};;
let f x = x.a;;
```


Exemples (records3.ml)

```
type complex = {re: float; im: float};;

let add x y = {re=x.re+.y.re; im=x.im+.y.im};;

(* filtrage par motif implicite *)
let partie_reelle {re=xr; im=xi} = xr;;
(* plus court *)
let partie_reelle {re=xr} = xr;;

partie_reelle {re=1.; im= -1.};;
```


Avantages des enregistrements

- On n'a pas besoin de se rappeler l'ordre des éléments, comme dans le cas d'un n-uplet.
- On n'a même pas besoin de connaître le nombre exact de champs pour accéder à un champ d'un enregistrement.
- Il y a moins de modifications à faire dans le code quand on ajoute un champ à un enregistrement, que quand on ajoute un composant à un n-uplet.

Les types énumérés

- Nécessitent une définition de type
- Énumération des constructeurs de ce type.
- Les constructeurs commencent obligatoirement sur une lettre en majuscule (ce que les distingue des identificateurs).
- Un constructeur ne peut appartenir qu'à un seul type
- Il s'agit d'un cas particulier des types somme.

Vers des types algébriques

Trois généralisations à partir des types numérés :

1. Les constructeurs peuvent prendre un argument
2. Polymorphie (similaires aux listes polymorphes)
3. Récurence dans les définitions de type (par exemple : un type pour les arbres)

Les types sommes

- Généralisation des types énumérés : les constructeurs peuvent avoir un argument
- Les constructeurs sont déclarés avec, le cas échéant, le type de leur argument
- Le type de l'argument peut être un n-uplet.
- Le même constructeur ne peut pas être déclaré avec plusieurs types d'arguments différents.

Exemples (somme3.ml)

```
type couleur = Pique | Coeur | Carreau | Trefle;;
type carte =
  | As of couleur
  | Roi of couleur
  | Dame of couleur
  | Valet of couleur
  | Numero of int * couleur
;;
Dame Coeur;;
Numero (9, Trefle);;
```


Exemples (somme4.ml)

```
let valeur atout carte =
  match carte with
    | As(_) -> 11
    | Roi(_) -> 4
    | Dame(_) -> 3
    | Valet(c) -> if c=atout then 20 else 2
    | Numero(10,_) -> 10
    | Numero(9,c) -> if c=atout then 14 else 0
    | Numero(_,_) -> 0
;;
valeur Trefle (Dame Coeur);;
valeur Trefle Dame(Coeur);; (* erreur de syntaxe *)
valeur Trefle (Numero(9,Trefle));;
```

Types sommes polymorphes

- Un type somme est polymorphe quand des types des constructeurs contiennent des variables de type
- Les variables de type commencent sur un apostrophe '
- Une variable de type dénote un type arbitraire
- Il faut indiquer les variables de type dans la définition de type (déclaration des variables de type)

Exemples (somme5.ml)

```
type ('a,'b) pair = Pair of 'a*'b;;
let x = Pair(1, "toto");;
let y = Pair(5.8, int_of_string);;
x<y;; (* types differents ! *)
```


Exemples (somme6.ml)

```
(* defini dans la bibliotheque Option *)
type 'a option = None | Some of 'a;;
let option_print x =
  begin
    match x with
      | None -> print_string "no option"
      | Some x -> print_string ("option "^x)
  end;
  print_newline ()
;;
option_print None;;
option_print (Some "coocoo");;
```


Types somme récursifs

- La définition d'un type somme peut être récursif : les constructeurs peuvent avoir comme types de leurs arguments le type qu'on est en train de définir.
- Il n'y a pas de "type rec"
- Utile seulement quand il y a un cas de base.

Exemples (somme7.ml)

```
type 'a liste =
  | Vide
  | Cons of 'a * 'a liste;;
let x = Cons(42, Cons(17,Vide));;
```


Exemples (somme8.ml)

```
type 'a bintree =
  | Leaf of 'a
  | Node of 'a * 'a bintree * 'a bintree
;;

let rec size x = match x with
  | Leaf _ -> 1
  | Node(_,left,right) -> 1+(size left)+(size right)
;;

size (Node(2,
           Leaf 7,
           Node(42, Leaf 9, Leaf 256)));;
```


Exemples (somme9.ml)

```
type formula =
    Var of string
  | Negation of formula
  | And of formula*formula
  | Or of formula*formula
;;

And(Var "x", Or (Var "y", Negation (Var "z")));;
```


Exemples (somme10.ml)

```
(* suite du precedent *)

let rec taille p = match p with
  | Var _ -> 1
  | Negation p -> 1 + (taille p)
  | And(p1,p2) | Or(p1,p2) -> 1 + (taille p1) + (ta

taille (And(Var "x", Or (Var "y", Negation (Var "z"

(* abbreviation *)

let rec taille =function
  | Var _ -> 1
  | Negation p -> 1 + (taille p)
  | And(p1,p2) | Or(p1,p2) -> 1 + (taille p1) + (ta

(* explication : voir le cours suivant *)
```