

Algorithmique et Approche Fonctionnelle

Cours 3 : Types Structurés

29 Septembre 2008

- les types de base (`int`, `float`, etc.) sont insuffisants pour représenter des données structurées (dates, matrices etc.)
- des codages existent, mais ils ne sont pas naturels (et parfois aussi très inefficaces)

dans ce cours nous allons étudier trois structures de données (et leurs opérations associées)

- 1 les **produits** (*n-uplets*)
- 2 les **produits nommés** (*enregistrements*)
- 3 les **sommes** (*constructeurs*)

le produit cartésien

la structure de données la plus simple pour former des valeurs complexes est la **paire** (ou *produit cartésien*)

```
# (1,2);;
```

```
- : int * int = (1,2)
```

les composantes des paires peuvent être de types différents

```
# ('a',2.7);;
```

```
- : char * float = ('a',2.7)
```

Les Types Produits

les fonctions `fst` et `snd` permettent respectivement d'accéder à la première et la deuxième composante d'une paire

```
# snd (1,2);;
- : int = 2
```

```
# fst ((1,2.5), 'a');;
- : int * float = (1,2.5)
```

```
# let p = (1,2) in (snd p, fst p);;
- : int * int = (2,1)
```

le produit cartésien (binaire) se généralise facilement afin de regrouper les valeurs en **n-uplets**

```
# (2+3, false || true, "bonjour");;
- : int * bool * string = (5, true, "bonjour")
```

les n-uplets et les paires peuvent se mélanger

```
# ('a', 1.2, (true, 0));;
- : char * float * (bool * int) = ('a', 1.2, (true, 0))
```

... les types suivants

```
int * int * int      (int * int) * int      int * (int * int)
```

- le premier désigne un **triplet** d'entiers
- le second est une **paire** dont la **première** composante est une **paire** d'entiers et la deuxième un entier
- le troisième est une **paire** dont la première composante est un entier et la **deuxième** composante est une **paire** d'entiers

on utilise une forme généralisée du `let`

```
let motif = e
```

ou une construction `match-with`

```
match e with
  motif -> ...
```

où le *motif* permet de **filtrer** le n-uplet représenté par l'expression `e`

```
# let v = ('a', 1.2, "bonjour");;
val v : char * float * string = ('a', 1.2, "bonjour")
```

on récupère les éléments de `v` avec `(x,y,z)` comme motif

```
# let (x,y,z) = v;;
val x : char = 'a'
val y : float = 1.2
val z : string = "bonjour"
```

```
# let v = (1,('a',2.3));;
val v : int * (char * float) = (1,('a',2.3))
```

accès aux composantes d'une paire

```
# let (x,c) = v;;
val x : int = 1
val c : char * float = ('a',2.3)
```

accès aux composantes d'une composante

```
# let (x,(y,z)) = v;;
val x : int = 1
val y : char = 'a'
val z : float = 2.3
```

les n-uplets peuvent être passés en arguments aux fonctions

```
# let f (x,y,z) = x + y * (int_of_float z);;
val f : int * int * float -> int = <fun>
```

```
# f (1,2,3.5);;
- : int = 7
```

ou retournés comme résultats

```
# let rec division n m =
  if n<m then (0,n)
  else
    let (q,r) = division (n - m) m in (q + 1,r);;
val division : int -> int -> int * int = <fun>
```

exemple

calcul efficace de la fonction de Fibonacci

fibonacci.ml

```
let fibonacci n =
  (* fonction intermédiaire retournant (fib(n) , fib(n+1)) *)
  let rec fib_rapide n =
    if n=0 then (0,1)
    else let (x,y) = fib_rapide (n-1) in (y,x+y)
  in
  fst (fib_rapide n)

print_int (fibonacci 15);;
```

compilation

```
ocamlc -o fibonacci fibonacci.ml
```

exécution

```
./fibonacci
610
```

inconvenients des n-uplets

(1/2)

- les objets représentés par des n-uplets ne sont pas identifiés de manière unique
- le système de types du langage ne peut donc pas être utilisé pour garantir de "bonnes" propriétés

exemple : on représente les nombres complexes par des paires (r,i) de type `float*float` où r et i sont respectivement la partie réelle et la partie imaginaire du complexe

- malheureusement ce type peut tout aussi bien représenter des nombres complexes en notation polaire, ou des intervalles etc.
- comment alors garantir que la fonction suivante est bien utilisée pour ajouter des complexes ?

```
# let add (r1,i1) (r2,i2) = (r1+r2 , i1+i2);;
val add : float*float -> float*float -> float*float = <fun>
```

les n-uplets avec un grand nombre de composantes deviennent très vite inutilisables en pratique

exemple : une fiche d'un fichier de personnes (nom, prénom, adresse, date de naissance, téléphone fixe, téléphone portable, etc.)

```
# let v =
  ("Durand", "Jacques",
   "2 rue J.Monod", "Orsay Cedex", 91893),
  (10,03,1967), "0130452637", "0645362738" ...)
```

- la consultation des informations devient vite pénible
- plusieurs éléments du n-uplet peuvent avoir le même type (ex. nom et prénom) et il est facile de les confondre (sans que le système de type puisse nous aider)

les enregistrements

le produit nommé permet de définir des **enregistrements** : des n-uplets dont les éléments (**champs**) ont chacun un nom *distinct*

en OCAML, chaque produit nommé (ou **type enregistrement**) possède un nom donné par l'utilisateur

```
# type complexe = { re : float ; im : float };;
type complexe = { re : float ; im : float }
```

on crée des valeurs de type complexe de la manière suivante

```
# { re = 1.4 ; im = 0.5 };;
- : complexe = { re = 1.4 ; im = 0.5 }
```

Produits Nommés

accès aux éléments d'un enregistrement

(1/2)

l'accès le plus simple aux champs d'un enregistrement se fait à l'aide de la notation

objet . nom_du_champ

```
# let v = { re = 1.3 ; im = 0.9 };;
val v : complexe = { re = 1.3 ; im = 0.9 }
# v.re;;
- : float = 1.3
```

le filtrage permet un accès **partiel** et en **profondeur** aux champs d'un enregistrement

```
# type t = { a : int ; b : float * char ; c : string };;
```

```
type t = { a : int ; b : float * char ; c : string }
```

```
# let v = { a = 1 ; b = (3.4, 'a') ; c = "bonjour" };;
```

```
val v : t = { a = 1 ; b = (3.4, 'a') ; c = "bonjour" }
```

```
# let { b = (_,x) ; c = y } = v;;
```

```
val x : char = 'a'
```

```
val y : string = "bonjour"
```

les définitions de types suivantes sont équivalentes

```
type t = { a : int ; b : char ; c : bool }
```

```
type t = { b : char ; c : bool ; a : int }
```

de même ces valeurs sont égales

```
# { a = 1 ; b = 't' ; c = true } = { b = 't' ; c = true ; a = 1 } ;;
```

```
- : bool = true
```

le filtrage est également insensible à l'ordre des champs

```
# let { c = x ; b = y } = { b = 't' ; c = true ; a = 1 } ;;
```

```
val x : bool = true
```

```
val y : char = 't'
```

structurer l'information

le mélange des n-uplets et des enregistrements permet de définir des objets complexes

```
# type adresse = { rue : string ; ville : string ; cp : int };;
```

```
# type fiche = {
  nom : string ;
  prenom : string ;
  adresse : adresse ;
  date_naissance : int * int * int ;
  tel_fixe : string ;
  portable : string
};;
```

création de nouvelles valeurs

```
# let v1 = { a = 1 ; b = false ; c = 'r' };;
```

```
val v1 : t = { a = 1 ; b = false ; c = 'r' }
```

on peut créer un nouvel enregistrement v2 en utilisant le contenu des champs de v1

```
# let v2 = { a = v1.a ; b = true ; c = v1.c };;
```

```
val v2 : t = { a = 1 ; b = true ; c = 'r' }
```

le raccourci syntaxique suivant permet d'arriver au même résultat

```
{v with c1 = e1 ; ... cn=en}
```

```
# let v3 = { v1 with b = true };;
```

```
val v3 : t = { a = 1 ; b = true ; c = 'r' }
```

une fonction prenant un enregistrement en argument

```
# let f v = v.a;;  
val f : t -> int
```

```
# f {a = 1; b = false; c = 'e'};;  
- : int = 1
```

les enregistrements peuvent aussi être retournés en résultat

```
# let f {a=x} v = { v with a = x+v.a } ;;  
val f : t -> t -> t
```

```
# let v = {a=1;b=true;c='r'} in f v v;;  
- : t = {a=2;b=true;c='r'}
```

Sommes

les types sommes

- modélisation de **domaines finis**
- réalisation de **sommes disjointes** permettant de réunir dans un même type des valeurs pouvant appartenir à des types différents

constructeurs constants

on peut modéliser un domaine fini comportant exactement n valeurs avec un type somme

exemple, les couleurs d'un jeu de carte

```
# type couleur = Pique | Coeur | Carreau | Trefle;;  
type couleur = Pique | Coeur | Carreau | Trefle
```

- les identificateurs Pique, Coeur, Carreau et Trefle sont des **constructeurs** (les majuscules sont obligatoires pour définir des constructeurs)
- le nom du domaine fini est couleur

l'unique manière de créer des valeurs d'un type somme est d'utiliser un constructeur

```
# Trefle;;
```

```
- : couleur = Trefle
```

```
# let v = (Pique , Coeur);;
```

```
val v : couleur * couleur = (Pique , Coeur)
```

```
# Pique = Coeur;;
```

```
- : bool = false
```

la construction **match-with** permet de définir de manière compacte une **analyse par cas** d'un type somme

```
# let points v =
  match v with
  | Pique -> 1
  | Trefle -> 2
  | Coeur -> 3
  | Carreau -> 4;;
```

```
val points : couleur -> int = <fun>
```

```
# points Coeur;;
```

```
- : int = 3
```

les constructeurs peuvent également avoir des **arguments**

par exemple,

```
# type num = Int of int | Float of float
```

```
type num = Int of int | Float of float
```

le mot-clé **of** indique que le constructeur attend un argument

on crée des valeurs en appliquant les constructeurs à des arguments du bon type

```
# Int(5);;
```

```
- : num = Int(5)
```

on utilise la construction **match-with** pour récupérer les arguments associés à un constructeur

```
# match Int(5) with Int(x) -> x+2;;
```

```
- : int = 7
```

en réalité, la réponse que l'on obtient est celle-là

```
Warning P : this pattern-matching is not exhaustive.
Here is an example of a value that is not matched :
Float _
- : int = 7
```

le filtrage exige de faire une analyse par cas **complète** en fonction du **type** de l'objet filtré et non de sa valeur

la complétude de l'analyse peut être obtenue en exécutant `failwith` "explication" pour les cas impossibles

```
# match Int(5) with
  Int(x) -> x+2
  | Float(x) -> failwith "cas impossible";;
- : int = 7
```

l'addition de valeurs de type `num` peut s'écrire ainsi

```
# let ajoute x y =
  match (x,y) with
    (Int(m) , Int(n)) -> Int(m + n)
  | (Int(m) , Float(n)) -> Float((float_of_int m) +. n)
  | (Float(m) , Int(n)) -> Float(m +. (float_of_int n))
  | (Float(m) , Float(n)) -> Float(m +. n);;
val ajoute : num -> num -> num = <fun>
```

on utilise simplement cette fonction de la manière suivante

```
# ajoute (Float(3.5)) (Int(5));;
- : num = Float(8.5)
```

autre exemple

le type des cartes d'un jeu de cartes

```
type valeur = Roi | Reine | Valet | Num of int
type couleur = Coeur | Pique | Trefle | Carreau
type carte = valeur * couleur
```

```
# let compare (c1,_) (c2,_) =
  if c1 = c2 then 0
  else match c1,c2 with
    | Roi, _ -> 1
    | Reine, Roi -> -1
    | Reine, _ -> 1
    | Valet, Roi -> -1
    | Valet, Reine -> -1
    | Valet, _ -> 1
    | Num(x), Num(y) -> (x - y) / (abs (x - y))
    | _ -> -1
```

allocation mémoire

- tous les objets sont alloués dans une zone mémoire appelée **tas** (sauf les types de base et sans compter les nombreuses optimisations du compilateur)
- l'allocation mémoire est réalisée par un **garbage collector** (GC) ce qui permet une récupération automatique de la mémoire et une allocation efficace