Constructeurs & Enregistrements

Alain Camanes

alain.camanes@free.fr

Stanislas

Option Informatique
2021-2022

Plan

Type tuple

```
# let (a,b) = (1.5, 1) ;;
val a : float = 1.5
val b : int = 1
# let c = (a,b) ;;
val c : float * int = (1.5, 1)
```

↪ Structure de données non mutable.

↪ Paires. *Ne* se généralise *pas*.

```
# fst c ;;
- : float = 1.5
# snd c ;;
- : int = 1
```

Égalité structurelle vs. physique

```
# let x = (1, 2);;
val x : int * int = 1, 2
# let y = (1, 2) ;;
val y : int * int = 1, 2
# let z = x ;;
val z : int * int = 1, 2
```

```
# x = y ;;
- : bool = true
# x == y ;;
- : bool = false
```

```
# x = z ;;
- : bool = true
# x == z ;;
- : bool = true
```

Motifs & Tuples

↪ Motif. Observation des composantes du tuple.

```
# let a = (2*3, 4*6, 72) ;;
val a : int * int * int = 6, 24, 72
# let (x, _, y) = a ;;
val x : int = 6
val y : int = 72
```

↪ _. Valeur ignorée

Motifs - Erreurs fréquentes

↪ Arité.

```
# let (x, _) = (1, 2, 3) ;;
Characters 13-22:
  let (x, _) = (1, 2, 3) ;;
               ^^^^^^^^^

Error: This expression has type 'a * 'b * 'c
       but an expression was expected of type 'd*'e
```

↪ Ne pas répéter deux fois la même variable.

```
# let (x, x) = (1, 1) ;;
Characters 8-9:
  let (x, x) = (1, 1) ;;
          ^

Error: Variable x is bound several times in this
       matching
```

Pattern matching

↪ Reconnaissance de motifs.

```
# let a = (1, 2) ;;
val a : int * int = 1, 2
# match a with
    |(1, 3) -> 3
    |(0, 1) -> 2
    |(_, 2) -> 4
    | _ -> 5 ;;
- : int = 4
```

↪ Fonctions.

```
let f n =
  if n = 0 then 1
  else
    if n = 1 then 2
    else 2 * n + 1 ;;
```

```
let f n =
  match n with
    | 0 -> 1
    | 1 -> 2
    | _ -> 2 * n + 1 ;;
```

Syntaxe

↪ Comparer l'argument à des *motifs*.

```
match expression with
  | p_1 -> e_1
  | p_2 -> e_2
  ...
  | p_n -> e_n
```

↪ Comportement analogue à des `elif`.

```
# match (1, 1) with
  | (_, 1) -> 1
  | (1, _) -> 2
  | _ -> 3;;
- : bool = 1
```

↪ Joker (*wildcar pattern*) : toutes les valeurs possibles.

Conditions de filtrage

↪ Une variable n'apparait qu'*une seule fois*.

Exhaustivité des motifs.

↪ Condition sur le motif : `when`.

```
let paire x y =
  match (x, y) with
    | (a, b) when a = b -> true
    | _ -> false ;;
```

↪ Stockage de la valeur du motif : `as`.

```
let min_biz a b =
  match (a,b) with
    | ((0,0), n) -> n
    | (m, (0,0)) -> m
    | (((c,d) as m), ((e,f) as n)) ->
      if (c <= e) && (d <= f) then m else n ;;
```

Plan

Nouveaux types

↪ Faciliter la compréhension des programmes.

```
# type color = int ;;
type color = int
```

↪ Définir de nouveaux objets,...

```
type 'a arbre_binaire = Vide
  | Noeud of 'a * 'a arbre_binaire * 'a arbre_binaire
```

↪ Nom des types doit commencer par une minuscule.

↪ Structures.

Type *Somme*	Type *Produit*
Énuméré	Enregistrement
Ou	*Et*
Union disjointe	Produits nommés à champs nommés
`bool` = {`true`, `false`}	`monome` = {`coeff`, `degre`}

Énumérations - Types sommes

Définition. Constantes commencent par des majuscules.

```
type peinture =
  | Bleu | Rouge | Vert ;;
```

Utilisation.

```
let est_bleu p = match p with
    | Bleu -> true
    | _ -> false ;;

# let k = Bleu and v = Vert ;;
val k : peinture = Bleu
val v : peinture = Vert
```

```
# est_bleu k ;;
- : bool = true
```

```
# est_bleu v ;;
- : bool = false
```

Constructeurs

↪ Définition. Constructeurs commencent par des majuscules.

Constantes = Constructeurs d'arité 0.

```
type peinture =
  | Bleu | Rouge | Vert
  | Numero of int ;;
```

↪ Utilisation.

```
let numero p = match p with
    | Numero n -> n
    | _ -> failwith "Couleur primaire" ;;

# let k = Bleu and p = Numero 10 ;;
val p : peinture = Numero 10
```

```
# numero p ;;
- : int = 10
```

```
# numero k ;;
Exception: Failure
    "Couleur primaire".
```

Récursifs

↪ Définition.

```
type peinture =
  | Bleu | Rouge | Vert
  | Numero of int
  | Melange of peinture * peinture ;;
```

↪ Utilisation.

```
let rec contient_bleu p = match p with
    | Bleu -> true
    | Melange (p1, p2) -> (contient_bleu p1)
        || (contient_bleu p2)
    | _ -> false ;;

# let r = Melange (Bleu, Numero 10) ;;
val r : peinture = Melange (Bleu, Numero 10)
# contient_bleu r ;;
- : bool = true
```

Enregistrements

↪ Définition.

```
type monome = {coeff : int; degre : int} ;;

let addition_monomes m1 m2 =
  if m1.degre = m2.degre then
    {coeff = m1.coeff + m2.coeff;
     degre = m1.degre}
  else failwith "N'ont pas les memes degres" ;;
```

↪ Utilisation.

```
let m = {coeff = 3; degre = 2} and
    p = {coeff = 6; degre = 2} ;;
# addition_monomes m p ;;
- : monome = {coeff = 9; degre = 2}
```

Enregistrements - Vocabulaire

↪ Généralisation des n-uplets.

↪ Le nom est appelé Étiquette, Rubrique ou Champ.

↪ L'ordre n'est pas important.

Champs mutables

↪ Définition.

```
type point = {mutable x:float; mutable y:float} ;;

type vecteur = {vx : float; vy : float} ;;

let translate a v =
  a.x <- a.x +. v.vx;
  a.y <- a.y +. v.vy ;;
```

↪ Utilisation.

```
# let a = {x = 1.; y = 2.1} and
    v = {vx = 3.1; vy = 2.3}
in translate a v; a ;;
   - : point = {x = 4.1; y = 4.4}
```

Plan

Type Option

↪ Une valeur est de type `t option` si

- soit cette valeur est `None`,
- soit elle est une valeur de type `Some v` avec `v` de type `t`.

↪ Utilisation. Renvoyer une valeur même si l'appel de la fonction n'admet pas de résultat intéressant.

Type Option : Exemple (I)

↪ *Pente* d'une droite passant par deux points

```
type point = float * float;;

# let pente (p1:point) (p2:point) =
    let (x1, y1) = p1 and (x2, y2) = p2 in
    let xd = x2 -. x1 in
    match xd with
    | 0. -> None
    | _ ->    Some ((y2 -. y1) /. xd)

val pente : point -> point -> float option = <fun>
```

Type Option : Exemple (II)

↪ *Racines réelles* d'un trinôme.

```
# let racines_reeles a b c =
    let dg = b**2. -. 4. *. a *. c in
    if dg < 0. then None
    else let d = sqrt dg in
    Some ((-.b-.d)/.(2.*.a), (-.b+.d)/.(2.*.a));;

val racines_reeles : float -> float -> float
          -> (float * float) option = <fun>
```

```
# let somme_racines_reelles a b c =
    match racines_reelles a b c with
    | None -> None
    | Some (a, b) -> Some (a +. b);;

val somme_racines_reelles : float -> float -> float
                          -> float option = <fun>
```