Programmation Fonctionnelle
Cours 05

Michele Pagani

Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

pagani@pps.univ-paris-diderot.fr

17 octobre 2016

Type Unit

Unit = la 0-uplet !

```
# ();;
- : unit = ()
```

- il est le type avec une seule valeur possible: ()
- l'intérêt d'une expression de type `unit` n'est pas dans sa valeur mais dans ses effets de bord

```
# print_string "Hello world\n";;
Hello world
- : unit = ()
```

  - modification de l'état du système (e.g. affichage à l'écran, écriture sur un fichier, changement de la mémoire...)
  - on sort ici du cadre purement fonctionnel

- on peut enchainer des expressions de type `unit` en utilisant ;

```
# print_string "Hello "; print_string "world\n";;
Hello world
- : unit = ()
```

Unit = la 0-uplet !

```
# ();;
- : unit = ()
```

- il est le type avec une seule valeur possible: `()`
- l'intérêt d'une expression de type `unit` n'est pas dans sa valeur mais dans ses effets de bord

  ```
  # print_string "Hello world\n";;
  Hello world
  - : unit = ()
  ```

  - modification de l'état du système (e.g. affichage à l'écran, écriture sur un fichier, changement de la mémoire...)
  - on sort ici du cadre purement fonctionnel

- on peut enchainer des expressions de type `unit` en utilisant `;`

  ```
  # print_string "Hello "; print_string "world\n";;
  Hello world
  - : unit = ()
  ```

Unit = la 0-uplet !

```
# ();;
- : unit = ()
```

- il est le type avec une seule valeur possible: `()`
- l'intérêt d'une expression de type `unit` n'est pas dans sa valeur mais dans ses effets de bord

  ```
  # print_string "Hello world\n";;
  Hello world
  - : unit = ()
  ```

  - modification de l'état du système (e.g. affichage à l'écran, écriture sur un fichier, changement de la mémoire...)
  - on sort ici du cadre purement fonctionnel

- on peut enchainer des expressions de type `unit` en utilisant `;`

  ```
  # print_string "Hello "; print_string "world\n";;
  Hello world
  - : unit = ()
  ```

Unit (fonctions)

print_char : char->unit	affiche un caractère
print_int : int->unit	affiche un entier
print_float : float->unit	affiche un nombre réel
print_string : string->unit	affiche une chaine de caractères
print_endline : string->unit	affiche une chaine suivie d'un changement de ligne
print_newline : unit->unit	affiche un changement de ligne
read_line : unit->string	lit une chaine de caractères
read_int : unit->unit	lit un entier
read_float : unit->float	lit un nombre réel

- plus sur `unit` quand on étudiera les traits impératifs de OCaml.

Unit (exemples)

```
(*notez la difference !*)
# let effet = print_endline "hello";;
hello
val effet : unit = ()

# let string = "hello";;
val string : string = "hello"

# string^" world";;
- : string = "hello world"

# effet^" world";;
    ^^^
Error: This expression has type unit but an expression
was expected of type string

# string ; 3;;
    ^^^
Warning 10: this expression should have type unit.
- : int = 3

# effet ; 3;;
- : int = 3
```

Unit (exemples)

```
# read_line ();;
Hello
- : string = "Hello"

# let hello = print_endline "Comment␣tu␣t'appelle?" ;
              let s = read_line () in
                  print_endline ("Bonjour␣"^s^"!");;
Comment tu t'appelle?
Michele
Bonjour Michele!
val hello : unit = ()

(*Qu'est-ce que fait cette fonction ?*)
# let rec perroquet x =
              print_endline "Je␣suis␣un␣perroquet" ;
              let s = read_line () in
                    print_endline s ;  perroquet ();;
val perroquet : unit -> 'a = <fun>
```

Unit (Exercice)

Écrire une function qui choisit un entier n au hasard (utiliser la fonction `Random.int`) et demande à l'utilisateur de deviner n. Si l'utilisateur choisit un entier plus grand (resp. plus petit) le programme lui affiche `trop grand` (resp. `trop petit`) et répète la demande. Le programme s'arrête lorsque l'utilisateur devine le nombre n.

Type enregistrement

Enregistrements = produits avec champs nommés

```
type typename = {identifier1 : type1;
                 ... ;
                 identifiern : typen}
```

- en Anglais: record
- défini par la mot clé `type` comme un ensemble de champs nommés par des identificateurs
- on ne peut pas utiliser le même nom de champs dans deux types d'enregistrement différents
  - nécessaire pour une inférence de type efficace
- une valeur est un ensemble associant à chance champ une valeur du type correspondant
- accès aux champs (projections) en notation pointée ou par filtrage par motif

Enregistrements (Exemples)

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

# let today = {
    day = 7; month = "october"; year = 2014;
  };;
val today : date = {day = 7; month = "october"; year = 2014}

# let tomorrow = {
  year = 2014;  day = 8;  month = "october";
  };;                              (*ordre des champs pas important *)
val tomorrow : date = {day = 8; month = "october"; year = 2014}

# today.year;;                                    (*notation pointee*)
- : int = 2014

# let getday {year=y; day=d; month=o} = d;;(*filtrage implicite*)
val getday : date -> int = <fun>
```

Enregistrements (Exemples)

```
# type r1 =
  {a: string; b:int};;
type r1 = { a : string; b : int; }

# type r2 =
  {a:string; b:float};;
type r2 = { a : string; b : float; }

# {a="john"; b=17;};;
                ^^^
Error: This expression has type int but an expression
         was expected of type float

# {a="john"; b=10.7};;
- : r2 = {a = "john"; b = 10.7}
```

Enregistrements vs *n*-uplets

Avantages des enregistrements:

- pas besoin de se rappeler l'ordre des éléments d'un enregistrement
- même pas besoin de connaître le nombre exact de champs pour accéder à un champ d'un enregistrement
- 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

Enregistrements vs *n*-uplets

Avantages des enregistrements:

- pas besoin de se rappeler l'ordre des éléments d'un enregistrement
- même pas besoin de connaître le nombre exact de champs pour accéder à un champ d'un enregistrement
- 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

Enregistrements vs *n*-uplets

Avantages des enregistrements:

- pas besoin de se rappeler l'ordre des éléments d'un enregistrement
- même pas besoin de connaître le nombre exact de champs pour accéder à un champ d'un enregistrement
- 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

Enregistrements (Exercise)

1. Définir un type nombre complexe
2. Définir la fonction qui calcule la valeur absolue d'un nombre complexe
3. Définir la fonction qui calcule la multiplication de deux nombres complexes

Filtrage par motifs (Pattern matching)

Filtrage par motif

S'applique à n'importe quel type (sauf fonctions et objets):

```
(*sur le listes*)
let rec map f list = match list with
    [] -> []
    | t::q -> (f t) :: (map f q);;

(*sur les types produits*)
let trd triplet = match triplet with (x,y,z) -> z;;

(*sur les types sommes*)
type 'a tree = Nil | Node of 'a * 'a tree * 'a tree;;
let rec height t = match t with
    | Nil -> 0
    | Node (_,t1,t2) -> 1+ (max (height t1) (height t2));;

(*sur les types de base*)
let rec fact n = match n with
  0 -> 1
  | n -> n* fact (n-1);;
```

Motif

```
# let rec map f list = match list with
    [] -> []
    | t::q -> (f t) :: (map f q);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- Un motif (en angl. pattern) est construit seulement de variables, valeurs et constructeurs
  - constructeurs liste: :: et []
  - constructeurs produit: (,)
  - constructeurs somme: définit avec la lettre majuscule, p.ex. Nil, Node
- Si un motif s'applique, tous les variables dans le motif sont liés. Leur portée : l'expression à droite du motif
- On peut dans un motif utiliser une variable générique _, dans ce cas il n'y a pas de liason

Motif

```
# let rec map f list = match list with
    [] -> []
    | t::q -> (f t) :: (map f q);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- Les motifs doivent être linéaires
  (pas de répétition d'identificateur dans le même motif)
- Les motifs sont essayés dans l'ordre donné
- OCaml vérifie qu'aucun cas n'a été oublié:
  l'ensemble des motifs doit être exhaustif.
- La non-exhaustivité donne lieu à un warning
- Il est fortement conseillé de faire des distinctions de cas exhaustifs

Filtrage (exemples)

Les motifs doivent être linéaires:

```
# let rec even_length l = match l with
    [] -> true
  | [_] -> false
  | t::t:: reste -> even_length reste;;
         ^^
Error: Variable t is bound several times in this matching
```

```
# let rec even_length l = match l with
    [] -> true
  | [_] -> false
  | _::_:: reste -> even_length reste;;
val even_length : 'a list -> bool = <fun>

# even_length [1;2;3];;
- : bool = false

# even_length [1;2;3;4];;
- : bool = true
```

Filtrage (exemples)

Les motifs doivent être linéaires:

```
# let rec even_length l = match l with
    [] -> true
  | [_] -> false
  | t::t:: reste -> even_length reste;;
          ^^
Error: Variable t is bound several times in this matching
```

```
# let rec even_length l = match l with
    [] -> true
  | [_] -> false
  | _::_:: reste -> even_length reste;;
val even_length : 'a list -> bool = <fun>

# even_length [1;2;3];;
- : bool = false

# even_length [1;2;3;4];;
- : bool = true
```

Filtrage (exemples)

Un motif est construit seulement de variables, valeurs et constructeurs:

```
# let rec fact n = match n with
  0 -> 1
  |n+1 -> (n+1)*(fact n);;
   ^^^
Error: Syntax error

# let rec length list = match list with
  [] -> 0
  |[t]@q -> 1+length q;;
     ^^^
Error: Syntax error
```

Filtrage (exemples)

Les motifs sont essayés dans l'ordre donné

```
# let rec fact n = match n with
   n -> n* fact (n-1)
   | 0 -> 1;;
Warning 11: this match case is unused.
val fact : int -> int = <fun>

# fact 2;;
Stack overflow during evaluation (looping recursion?).
```

Filtrage (exemples)

L'ensemble des motifs doit être exhaustif

```
# let hd list = match list with
  t::q -> t;;

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val hd : 'a list -> 'a = <fun>

# hd [];;
Exception: Match_failure ("//toplevel//", 113, -6).
```

Filtrage (exemples)

```
(* quel est l'erreur ? *)

# let rec trouve a list = match list with
    [] -> false
  |a::_-> true
  |b::q -> trouve a q;;
val trouve : 'a -> 'b list -> bool = <fun>

# trouve 1 [1;2;3];;
- : bool = true

# trouve 42 [1;2;3];;
- : bool = true
```

Tous les identificateurs dans le motif sont liés

Filtrage (exemples)

```
(* quel est l'erreur ? *)

# let rec trouve a list = match list with
    [] -> false
  |a::_-> true
  |b::q -> trouve a q;;
val trouve : 'a -> 'b list -> bool = <fun>

# trouve 1 [1;2;3];;
- : bool = true

# trouve 42 [1;2;3];;
- : bool = true
```

Tous les identificateurs dans le motif sont liés

Filtrage (exemples)

```
(* la fonction trouve corrige *)

# let rec trouve a list = match list with
    [] -> false
  |b::q -> if b=a then true else trouve a q;;
val trouve : 'a -> 'a list -> bool = <fun>

# trouve 1 [1;2;3];;
- : bool = true

# trouve 42 [1;2;3];;
- : bool = false
```

Filtrage (exemples)

- motifs avec des alternatives

```
let rec fib n = match n with
0 | 1 -> n
  | n -> fib (n-1)+fib (n-2);;
val fib : int -> int = <fun>

# fib 10;;
- : int = 55
```

- motifs avec des conditions

```
let rec trouve a list = match list with
  | [] -> false
  | b::q when b = a -> true
  | _::q -> trouve a q;;
val trouve : 'a -> 'a list -> bool = <fun>

# trouve 1 [1;2;3];;
- : bool = true
# trouve 42 [1;2;3];;
- : bool = false
```

Filtrage (exemples)

- motifs nommés

```
# let min_rat pr = match pr with
    ((_,0),p2) ->  p2
  | (p1,(_,0)) ->  p1
  | (((n1,d1) as r1), ((n2,d2) as r2)) ->
          if (n1 *  d2 ) < (n2 * d1) then r1 else r2;;
val min_rat : (int * int) * (int * int) -> int * int = <f
```

- filtrage sur plusieurs paramètres

```
let implication x y = match x,y with
  | (true, false) -> false
  | (true, true) -> true
  | (false, true) -> true
  | (false, false) -> true
val implication : bool -> bool -> bool = <fun>
```