Programmation Fonctionnelle : des concepts aux applications web (NFP119)

Polymorphisme (généricité)

Tristan Crolard

Laboratoire CEDRIC
Equipe « Systèmes Sûrs »

tristan.crolard@cnam.fr

cedric.cnam.fr/sys/crolard

Polymorphisme

Question. Qu'infère le système de type si on ne donne pas assez d'information ?

Réponse. Le type le plus général (autrement dit, le plus générique).

Exemples

```
# let id x = x;;
val id : 'a -> 'a = <fun>
```

On peut instancier cette fonction en précisant son type, par exemple :

```
# let idchar : char -> char = id;;
val idchar : char -> char = <fun>
```

Mais c'est inutile, c'est fait automatiquement.

```
# let c = idchar 'A';;
val c : char = 'A'
# let c = id 'A';;
val c : char = 'A'
```

Polymorphisme et couples

```
# let fst (x, _) = x;;
val fst : 'a * 'b -> 'a = <fun>
# let snd (_, y) = y;;
val snd : 'a * 'b -> 'b = <fun>
# let exch (x, y) = (y,x);;
val exch : 'a * 'b -> 'b * 'a = <fun>
# let x = fst (3, 'Z');;
val x : int = 3
# let c = snd (3, 'Z');;
val c : char = 'Z'
# exch (3, 'Z');;
- : char * int = ('Z', 3)
```

Polymorphisme et listes

Quelques fonctions prédéfinies du module `List` :

```
List.length : 'a list -> int
List.hd : 'a list -> 'a
List.tl : 'a list -> 'a list
List.rev : 'a list -> 'a list
```

Application

```
# let addlast (x, l) = List.rev (x :: (List.rev l));;
val addlast : 'a * 'a list -> 'a list = <fun>
# let l = addlast (4, [1; 2; 3]);;
val l : int list = [1; 2; 3; 4]
# let mirror l = l @ (List.rev l);;
val mirror : 'a list -> 'a list = <fun>
# let k = mirror [1; 2; 3; 4];;
val k : int list = [1; 2; 3; 4; 4; 3; 2; 1]
```

Conditionnelle

La conditionnelle est bien sûr aussi une expression. Par conséquent, la partie `else` est obligatoire et doit avoir le même type que la partie `then`.

```
# let s = if (1 = 2) then "a" else "b";;
```

```
val s : string = "b"
```

```
# let s = if (1 = 2) then "a" else 9;;
```

```
Error: This expression has type int but an expression was expected of
type string
```

Récursivité

Une fonction peut s'appeler elle-même : on utilise alors `let rec` pour la définir (au lieu de `let`). Si plusieurs fonctions sont mutuellement récursives, on utilise alors le mot-clé `and` à la place de `let`.

Exemples

```
# let rec factorial n =
    if (n = 0) then 1
    else n * factorial (n-1);;
val factorial : int -> int = <fun>
# let rec odd n =
    if n = 0 then false else even (n-1)
  and even n =
    if n = 0 then true else odd (n-1);;
val odd : int -> bool = <fun>
val even : int -> bool = <fun>
```

Définition par cas (instruction `match`)

```
# let rec factorial n =
    match n with
    | 0 -> 1
    | _ -> n * factorial (n-1);;
val factorial : int -> int = <fun>
# let rec odd n =
    match n with
    | 0 -> false
    | _ -> even (n-1)
  and even n =
    match n with
    | 0 -> true
    | _ -> odd (n-1);;
val odd : int -> bool = <fun>
val even : int -> bool = <fun>
```

Filtrage (instruction `match`)

« définition par cas pour les listes »

Exemples

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

# let rec length l =
    match l with
    | [] -> 0
    | (_::t) -> 1 + length t;;
val length : 'a list -> int = <fun>
```

```
# let rec sum l =
    match l with
    | [] -> 0
    | (h::t) -> h + sum t;;
val sum : int list -> int = <fun>
```

```
# let rec nth (l, x) =
    match (l, x) with
    | (h::t, 0) -> h
    | (h::t, n) -> nth (t, n-1);;
val nth : 'a list * int -> 'a = <fun>
```

```
# let rec zip (l1, l2) =
    match (l1, l2) with
    | (h1::t1, h2::t2) -> (h1, h2)::(zip (t1, t2))
    | (_, _) -> [];;
val zip : 'a list * 'b list -> ('a * 'b) list = <fun>
```

```
# let rec unzip pl =
    match pl with
    | [] -> ([],[])
    | ((h1, h2)::t) ->
      let (t1, t2) = unzip t in
      (h1::t1, h2::t2);;
val unzip : ('a * 'b) list -> 'a list * 'b list = <fun>

# let rec unzip pl =
    match pl with
    | [] -> ([],[])
    | ((h1, h2)::t) ->
      match (unzip t) with (t1, t2) ->
      (h1::t1, h2::t2);;
val unzip : ('a * 'b) list -> 'a list * 'b list = <fun>
```

Filtrage en Python 3.12 (instruction `match`)

La fonction length

```
def length[A](l: list[A]) -> int:
    match l:
        case [_, *t]:
            return 1 + length(t)
        case _:
            return 0
```

La fonction unzip

```
def unzip[A, B](l: list[tuple[A, B]]) -> tuple[list[A], list[B]]:
    match l:
        case [(h1, h2), *t]:
            (t1, t2) = unzip(t)
            return ([h1, *t1], [h2, *t2])
        case _:
            return ([], [])
```

Filtrage en Java 21 (instruction `switch`)

Les listes simplement chaînées

```
sealed interface List<A> {}
record Nil<A>() implements List<A> {}
record Cons<A>(A head, List<A> tail) implements List<A> {}
```

La fonction length

```
public static <A> Integer length(List<A> l) {
    switch (l) {
        case Nil() -> return 0;
        case Cons(var _, var t) -> return 1 + length(t);
    }
}
```

Les couples

```
record Pair<A,B>(A fst, B snd) {}
```

La fonction unzip

```
public static <A,B> Pair<List<A>, List<B>> length(List<Pair<A,B>> pl) {
    switch (pl) {
        case Nil() -> return new Pair<>(new Nil<>(), new Nil<>());
        case Cons(Pair(var h1, var h2), var t) -> {
            switch (unzip(t)) {
                case (t1, t2) ->
                    return new Pair<>(new Cons<>(h1, t1), new Cons<>(h2, t2));
            }
        }
    }
}
```