IPRF – Types, Listes

Christophe Mouilleron

ensiie
école nationale supérieure d'informatique
pour l'industrie et l'entreprise

Plan

Plan

Types énumérés

Syntaxe :

```
type nom = Constructeur1 | ... | Constructeurk;;
```

Exemple :

```
# type couleur = Rouge | Vert | Bleu ;;
type couleur = Rouge | Vert | Bleu
```

Manipulation d'un objet de type `couleur` ?

Types énumérés

Syntaxe :

```
type nom = Constructeur1 | ... | Constructeurk;;
```

Exemple :

```
# type couleur = Rouge | Vert | Bleu ;;
type couleur = Rouge | Vert | Bleu
```

Manipulation d'un objet de type `couleur` ?

- soit par branchement — `Rouge` vu comme une valeur
- soit par filtrage — `Rouge` vu comme un constructeur

$\rightsquigarrow$ question de goût ici !

Types énumérés : Exemple de la conversion en chaîne

Par branchement :

```
# let string_of_couleur = fun c ->
    if c = Rouge then "rouge"
    else if c = Vert then "vert"
    else "bleu"
  ;;
val string_of_couleur : couleur -> string = <fun>
```

Par filtrage :

```
# let string_of_couleur = fun c -> match c with
  | Rouge -> "rouge"
  | Vert -> "vert"
  | Bleu -> "bleu"
  ;;
val string_of_couleur : couleur -> string = <fun>
```

Types sommes (union)

Les constructeurs peuvent avoir un argument :

- syntaxe : `Constructeur of type`
- permet de mixer plusieurs types de façon fiable

```
# type nombre = Int of int | Real of float ;;
type nombre = Int of int | Real of float

# Int 17, Real 42.0 ;;
- : nombre * nombre = (Int 17, Real 42.)
```

Types sommes (union)

Les constructeurs peuvent avoir un argument :

- syntaxe : `Constructeur of type`
- permet de mixer plusieurs types de façon fiable

```
# type nombre = Int of int | Real of float ;;
type nombre = Int of int | Real of float

# Int 17, Real 42.0 ;;
- : nombre * nombre = (Int 17, Real 42.)
```

Manipulation d'un objet dont le type est un type somme :

- par filtrage structure = constructeur utilisé

Types sommes : Exemple

On peut définir l'addition de deux éléments de type `nombre` par :

Types sommes : Exemple

On peut définir l'addition de deux éléments de type `nombre` par :

```
# let add = fun a -> fun b -> match a, b with
  | Int i, Int j -> Int (i + j)
  | Int i, Real j -> Real (float_of_int i +. j)
  | Real i, Int j -> Real (i +. float_of_int j)
  | Real i, Real j -> Real (i +. j) ;;
val add : nombre -> nombre -> nombre = <fun>

# add (Int 1) (Real 3.14) ;;
- : nombre = Real 4.1400000000000057
```

$\rightsquigarrow$ Un branchement ne marche pas ici :
impossible de récupérer les valeurs *i* et *j*.

Types inductifs

Le type de l'argument d'un constructeur peut contenir celui qu'on est en train de définir :

- permet de définir des structures inductives
- manipulations possibles via le filtrage

Exemples :

```
(* un type pour les listes d'entiers *)
type iliste = Vide | Cons of int * iliste ;;

(* un type pour les arbres binaires sans valeurs *)
type arbre = Feuille | Noeud of arbre * arbre ;;
```

Polymorphisme

Enfin, un type peut dépendre d'un type :

- on parle de polymorphisme
- essentiel pour définir des structures de données

Notation pour les variables de type = ' suivi d'un mot

Exemples :

```
(* listes polymorphes *)
type 'a liste = Vide | Cons of 'a * 'a liste ;;

(* arbres avec valeurs dans les noeuds internes *)
type 'a arbre = Feuille
              | Noeud of 'a * 'a arbre * 'a arbre ;;
```

Plan

Plan

Listes : interface

Liste = structure linéaire

Interface :

- cons = ajout en tête $O(1)$
- head = lecture du premier élément $O(1)$
- tail = suppression du premier élément $O(1)$
- test si liste vide $O(1)$

Autres opérations usuellement demandées :

- calcul de la longueur $O(n)$
- lecture du i-ème élément $O(i)$
- renverser l'ordre la liste $O(n)$
- recherche dans une liste de taille n $O(n)$

Listes : exemple de définition en OCaml

Définition possible en OCaml :

```
type 'a liste = Vide | Cons of 'a * 'a liste ;;
```

Implantation de l'interface :

```
let cons = fun x -> fun l -> Cons (x, l) ;;

let head = fun l -> match l with Cons (h, _) -> h ;;

let tail = fun l -> match l with Cons (_, t) -> t ;;

let is_empty = fun l -> match l with
  | Vide -> true
  | _ -> false ;;
```

Listes : remarques

Quid de l'accès au dernier élément ?

Listes : remarques

Quid de l'accès au dernier élément ?

- l'interface dit $O(n)$ c'est le cas en OCaml
- mais l'implantation peut être meilleure en pratique
  $\rightsquigarrow$ $O(1)$ via les listes doublement chaînées

Listes : remarques

Quid de l'accès au dernier élément ?

- l'interface dit $O(n)$ c'est le cas en OCaml
- mais l'implantation peut être meilleure en pratique
  $\rightsquigarrow$ $O(1)$ via les listes doublement chaînées

En OCaml, on utilisera le support fourni par le langage :

- listes simplement chaînées
- définition inductive à partir de `[]` et `::`

Listes : remarques

Quid de l'accès au dernier élément ?

- l'interface dit $O(n)$ c'est le cas en OCaml
- mais l'implantation peut être meilleure en pratique
  $\rightsquigarrow$ $O(1)$ via les listes doublement chaînées

En OCaml, on utilisera le support fourni par le langage :

- listes simplement chaînées
- définition inductive à partir de `[]` et `::`

Ne pas confondre interface et implantation !

Fonctions usuelles sur les listes

Le module `List` de OCaml fournit (entre autres) :

- `List.hd` / `List.tl` = récupération de la tête/ la queue d'une liste non vide $O(1)$
  - $\rightsquigarrow$ aussi possible par filtrage
- `List.length` = calcul de la longueur (nb d'éléments) $O(n)$
- `List.mem` = test d'appartenance $O(n)$
- `List.rev` = renverse l'ordre de la liste $O(n)$
- `List.nth` = lecture du i-ème élément de `l` via `List.nth l i;;` $O(i)$
- `List.sort` = tri selon l'ordre passé en argument $O(n \log n)$

Plan

Et pour les autres opérations sur les listes ?

Pour le reste, on peut écrire des fonctions récursives :

- toujours faisable avec du filtrage,
- code long à écrire / lire.

Et pour les autres opérations sur les listes ?

Pour le reste, on peut écrire des fonctions récursives :

- toujours faisable avec du filtrage,
- code long à écrire / lire.

Deux cas particuliers où on peut éviter le filtrage :

- itération = appliquer une fonction à chaque élément d'une liste
- agrégation = appliquer une fonction à tous les éléments d'une liste

Itération sur une liste

Le module `List` fournit :

```
# List.map ;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

En entrée :

- une fonction *f* de `'a` dans `'b`
- une liste d'éléments de type `'a` `[ x1; x2; ...; xn]`

En sortie :

- une nouvelle liste `[ f(x1); f(x2); ...; f(xn) ]`

Itération : Exemple 1

Ajouter 10 à tous les éléments d'une liste d'entiers :

```
# let l = [1; 2; 3] ;;
val l : int list = [1; 2; 3]

# List.map (fun x -> x + 10) l ;;
- : int list = [11; 12; 13]

# List.map (fun x -> x + 10) ;;
- : int list -> int list = <fun>
```

$\rightsquigarrow$ Beaucoup plus compact qu'une fonction récursive avec filtrage.

Itération : Exemple 2

Afficher tous les éléments d'une liste de caractères :

```
# let affiche_char = fun c ->
    Printf.printf "%c " c
  ;;
val affiche_char : char -> unit = <fun>

# let l = [ 'h'; 'e'; 'l'; 'l'; 'o' ] ;;
val l : char list = ['h'; 'e'; 'l'; 'l'; 'o']

# List.map affiche_char l ;;
h e l l o - : unit list = [(); (); (); (); ()]
```

Agrégation

Le module `List` fournit aussi :

```
# List.fold_left ;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

En entrée :

- une fonction f — fonction d'agrégation
- une valeur x_0 de type `'a` — valeur initiale
- une liste d'éléments de type `'b` — `[x1; x2; ...; xn]`

En sortie :

- une valeur de type `'a` — `f(...(f(f(x0, x1), x2), ...), xn)`

Agrégation : Exemple 1

Ajouter tous les éléments d'une liste d'entiers :

```
# let l = [1; 2; 3] ;;
val l : int list = [1; 2; 3]

# List.fold_left (fun acc e -> acc + e) 0 l ;;
- : int = 6

# List.fold_left (+) 0 l ;;
- : int = 6

# List.fold_left (+) ;;
- : int -> int list -> int = <fun>
```

Agrégation : Exemple 2

Taille d'une liste sans `List.length`.

Fonction récursive avec filtrage :

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

Solution plus compacte :

```
# let length = fun l ->
    List.fold_left (fun acc _ -> 1+acc) 0 l ;;
val length : 'a list -> int = <fun>
```

Agrégation avec parcours de droite à gauche

Le module `List` fournit enfin :

```
# List.fold_right ;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Exemple :

```
# let concat = fun c -> fun s ->
    (string_of_int c) ^ s
  ;;
val concat : int -> string -> string = <fun>

# List.fold_right concat [1;2;3] "4" ;;
- : string = "1234"
```