Programmation fonctionnelle

ENSIIE

Semestre 4 – 2019/20

Types définis par le programmeur

Enregistrements

Syntaxe

```
type enr = {
  champ1 : type1;
  champ2 : type2;
  ...
  champn : typen
}

let v = { champ1 = e1;
  champ3 = e3; ...;
  champ2 = e2 }

let u = v.champ4
```

Persistance

Attention : fonctionnel $\rightarrow$ non modifiable
`v.champ1 = e` : test d'égalité

Possibilité d'utiliser :

```
let w = { v with champ5 = e5; champ1 = e1' }
```

- `w` nouvelle valeur identique à `v` sauf pour les champs `champ5` et `champ1`

Types énumérés

Syntaxe

```
type t = Constr1 | Constr2 | ... | Constrn
```

Manipulation par test ou par filtrage

- Ici, question de goût

Types somme

Syntaxe

```
type t = Constr1 of type1
   | Constr2 (* volontairement sans type *)
   | ...
   | Constrn of typen
```

Manipulation par filtrage

- permet de récupérer la valeur associée

Types inductifs

Syntaxe

```
type t = Constr1 of type1
   | Constr2 (* volontairement sans type *)
   | ...
   | Constrn of typen
```

Possibilité d'utiliser `t` dans `type1`, ..., `typen`
Manipulation par filtrage et récursion

Exemples

Listes :

```
type intlist = Nil | Cons of int * intlist
```

Arbres binaires

```
type inttree =
    Leaf of int
  | Node of int * inttree * inttree
```

Polymorphisme

Enfin, un type peut dépendre d'un type :

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

Notation pour les variables de type = `'ident`

Listes polymorphes :

```
type 'a list = Nil | Cons of 'a * 'a list
```

Arbres binaires

```
type 'a tree =
    Leaf of 'a
  | Node of 'a * 'a tree * 'a tree
```

Alias de type

Nouveau nom pour un type existant

```
type t = type existant
```

Exemple :

```
type intlist = int list
```

Attention ! on peut utiliser le nouveau nom dans les types, mais en général c'est celui existant qui est retourné par OCaml

Listes

Interface des listes

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$^{\text{e}}$ élément $O(i)$
- renverser l'ordre la liste $O(n)$
- recherche dans une liste de taille n $O(n)$

Implémentation standard en OCaml

Deux constructeurs

- `[] : 'a list`
- `x::q : 'a * 'a list -> 'a`

Type construit inductivement

- utilisation de la récursivité et du filtrage

Récursion terminale

Fonction récursive

- dépassement de pile si trop d'appels récursifs

Sauf si :

- appel récursif terminal

Dans ce cas, la trame courante est dépilée avant de passer la main à l'appel récursif

- car plus utile
- la pile ne croît pas !

Utilisation de fonctions auxilliaires

Il faut parfois généraliser la fonction pour la rendre récursive terminale

```
let rec fact n =
  if n <= 1 then 1
  else n * fact (n-1) (* pas terminal *)
```

VS

```
let rec fact_aux acc n =
  if n <= 1 then acc
  else fact (n * acc) (n-1) (* terminal *)

let fact = fact_aux 1
```

Fonctions utiles sur les listes

- longueur
- inversion
- n^{e} élément
- concaténation
- appartenance
- recherche
- itération (map)
- test (pour tout/il existe)
- filtre
- agrégation (fold)

Bibliothèque

Ne pas réinventer la roue :

- utiliser le module `List`

Cf. `https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html`

Usage :

```
List.nom_fonction
```

ou alors

```
open List
...
nom_fonction
```

Pour le reste ?

Possibilité d'utiliser filtrage et récursion

- code long et parfois peu lisible

Mieux :

- utilisation d'itération et d'agrégation

Exemple

Ajouter 10 à tous les éléments d'une liste

Sans map :

```
let rec add_10 l = match l with
    [] -> []
  | x::q -> (x+10) :: add_10 q
```

Avec map

```
let add_10 = List.map (fun x -> x + 10)
```

voire

```
let add_10 = List.map ((+) 10)
```

Autre exemple

Calcul de la longueur :

```
let length =
  List.fold_left (fun acc _ -> 1 + acc) 0
```

Somme des éléments d'une liste

```
let sum = List.fold_left (+) 0
```

Au delà du fonctionnel

Quasiment toutes les fonctions peuvent s'écrire comme combinaison d'itération/agrégation

- Patron d'architecture Map Reduce pour la programmation distribuée