Programmation fonctionnelle, OCaml #2

Pierre Karpman

Lycée Champollion MP2I

https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

Table des matières

Type liste

Comment manipuler des collections d'objets de types homogènes ?

- Approche impérative classique (par ex. C) : avec des tableaux
- Approche fonctionnelle classique (par ex. OCaml) : avec des listes
- (Approche ??? : avec des trucs qui s'appellent `list` qui sont en fait surtout des tableaux où les éléments n'ont pas besoin d'avoir le même type)

Liste fonctionnelle

Une liste fonctionnelle (ou juste « liste ») est un type de données permettant (typiquement) :

- de stocker un nombre arbitraire d'objets (ses éléments) d'un **même** type
- de tester si une liste est vide (ne contient pas d'éléments)
- si une liste est non vide, de « récupérer » sa *tête* (*head*) (son premier élément)
- si une liste est non vide, de « récupérer » sa *queue* (*tail*) (toute la liste sauf son premier élément)
- étant donnée une liste *xs* (éventuellement vide) et un objet *x du bon type*, construire une nouvelle liste dont *x* est la tête et *xs* est la queue

Listes en OCaml

`'a list`

OCaml fournit un type `'a list` prédéfini

- Type *polymorphe* des listes contenant des objets tous du même type `'a`
- Peut être spécialisé par ex. en `int list`, `bool list`, `int list list`...

Littéraux

- Liste vide : `[]`, toujours de type `'a list`
- Listes non vides : `[x1; x2; ...]` d'éléments `x1`, `x2` etc. de type `t` ; la liste est de type `t list` (n'est plus polymorphe)

  Exemples :
  - `([1; 2]:int list)`
  - `([true]:bool list)`
  - `([ [1] ]:int list list)`
  - `([ [] ]:'a list list)`
  - `([ fun x -> x ]:('a -> 'a) list)`

Listes en OCaml *bis*

Fonctions de base

- Tester si une liste est vide : `List.is_empty`
  S'évalue à `true` si son argument est une liste vide, `false` sinon
- « Récupérer » la tête : `List.hd`
  Pour une liste `x` **non vide**, `List.hd x` s'évalue en la tête de celle-ci
- « Récupérer » la queue : `List.tl`
  Pour une liste `x` **non vide**, `List.tl x` s'évalue en la queue (éventuellement vide) de celle-ci
- Construire une liste : `List.cons`
  Pour une liste `xs` et un élément `x` « d'un bon type », `List.cons x xs` construit une nouvelle liste dont `x` est la tête et `xs` la queue

Bientôt : rien de tout ça

Les fonctions ci-dessus s'utilisent en pratique assez peu ; on leur préfère généralement le *filtrage de motif*

Listes en OCaml *ter*

Exemples

```
List.is_empty [] (* true *)
List.is_empty [ [] ] (* false *)
List.hd [1] (* 1 *)
List.tl [1] (* [] *)
List.hd [1; 2; 3; 4] (* 1 *)
List.tl [1; 2; 3; 4] (* [2; 3; 4] *)
List.cons 1 [2 ; 3] (* [1; 2; 3] *)
List.hd [] (* Exception: Failure "hd" *)
List.cons true [1] (* Error: The constant 1 has type int but an
                      expression was expected of type bool *)
```

Équations 🙈

Pour tout `(x:'a)`, `(xs:'a list)` on a :

- `List.hd (List.cons x xs) = x`
- `List.tl (List.cons x xs) = xs`

Listes en OCaml *quater* : immuabilité & partage

Listes immuables

Les `list` OCaml sont *immuables* : on ne peut pas les modifier (**JAMAIS**)

- On **ne peut pas** ajouter/supprimer d'éléments
- On **ne peut pas** modifier la « valeur » d'un élément
  - Mais des fois ces éléments eux-mêmes permettent de modifier d'*autres choses* ; quand ils s'assimilent à des pointeurs par ex.

Partage et coût mémoire

L'immuabilité permet le *partage* : lorsque l'on construit `List.cons x xs`, il n'est pas nécessaire de recopier `xs`, puisqu'elle est garantie ne jamais pouvoir changer

- Coût mémoire (marginal) : taille de `x` seulement !
- Coût temporel (marginal) : constant !

En OCaml, les types immuables bénéficient du partage « par défaut »

- Permet de faire des choses efficaces/rigolotes (qui se retrouvent parfois aux concours)

Table des matières

Intermède : durée de vie & gestion mémoire

Problématique

En OCaml, les `list` (par exemple) peuvent être (quasiment) *arbitrairement longues*.
Concrètement il faut donc :

- allouer de la mémoire pour les stocker
- libérer la mémoire occupée par les listes qui ne servent plus (c'est mieux)

Gestion automatique de la mémoire

En OCaml, la gestion mémoire est *automatique* : aucune allocation ou libération explicite n'est nécessaire de la part de la / du programmeur

- tout objet que l'on peut référencer syntaxiquement est vivant
- l'environnement d'exécution se charge de récupérer la mémoire des objets inaccessibles (grâce à un *garbage collector* (*GC*) (« ramasse-miette »))

Avantages : allège la programmation ; évite de nombreux bugs (accès illégaux, fuites mémoires)

Inconvénients : moins efficace qu'une gestion manuelle (quand elle est bien faite…)

Table des matières

Types produit

Les listes fonctionnelles :

- stockent un nombre non borné d'objets de même type
- remplissent (pas mal) le rôle des tableaux en programmation impérative

Quid des enregistrements ?

(Avec des enregistrements OCaml, mais avant ça… :)

Types produit

Soit $\alpha_1, \alpha_2, \ldots \alpha_n$ *n* types (pas forcément distincts), leur produit $\alpha_1 \times \alpha_2 \times \cdots \alpha_n$ est le type des *n*-uplets ordonnés dont le *i*ème élément est de type α_i

Exemples OCaml

- `(1,2:int * int)` : paire d'entiers `int`
- `(1,true,2:int*bool*int)` : triplet de deux `int` avec un `bool` au milieu
- `'a * 'a` : paire de deux éléments d'un même type quelconque
- `'a * 'b` : paire de deux éléments de deux types quelconques (possiblement identiques)

Types produit en OCaml

Syntaxe

- Écriture du type : « avec des * » (*cf.* ci-dessus)
- Construction d'une paire, d'un triplet… : en séparant les éléments par des **virgules** (pareil). Parenthésage pas systématiquement nécessaire
- Récupération des éléments : plusieurs possibilités :
  - `let x1, x2, x3 = t` pour récupérer les trois éléments du **triplet** `t` (dans cet ordre)
  - `let x1, _, x3 = t` pour récupérer le premier et troisième seulement (`_` : *wildcard*)
  - `fst p`, `snd p` pour une paire seulement (techniquement hors programme)
  - Par filtrage (*cf.* ci-dessous)

Dans tous les cas : adapté au nombre d'éléments du type

Exercice

Quels types pour :

```
[1, 2, 3]
[1; 2; 3]
```

?

Types produit en OCaml *bis*

Contraintes de type

Si `t` est un triplet, on ne peut par ex. pas faire :

```
let x1, x2 = t
```

pour seulement récupérer les deux premiers éléments (`Error: This expression has type 'a * 'b * 'c but an expression was expected of type 'd * 'e`)

- Raison pour laquelle `fst` ne fonctionne **que pour les paires** (visible dans son type : `'a * 'b -> 'a`)

Immuabilité

Les objets de type produits OCaml sont immuables

- On ne peut pas modifier leurs éléments
- Ils bénéficient du partage
  - Par ex., construire une paire de listes (elles-mêmes immuables) ne nécessite pas de recopier les listes

Table des matières

Types somme

Types somme

Soit $\alpha_1, \alpha_2, \dots \alpha_n$ *n* types (pas forcément distincts), leur somme $\alpha_1 + \alpha_2 + \cdots \alpha_n$ est le type des objets pouvant être de type α_1, ou α_2, ou … α_n

- Autres noms possibles : types *énumérés*, types *union*

… Récursifs

En OCaml, les types sommes peuvent être récursifs : on peut définir un type somme $\sigma = \sigma + \cdots$

Utilité

Notamment :

- Pour représenter une alternative (par ex. présence / absence de valeur)
- Pour construire (récursivement) des types complexes

Types somme en OCaml

Principe

Les types somme OCaml doivent être nommés, et soit $\sigma = \alpha_1 + \cdots$ un tel type non vide, chaque α_i doit être associé à un constructeur (qui permettra d'identifier le *sous-type* d'un objet de type σ)

Syntaxe de définition (sur des exemples)

Cas simple :

```
type nom_du_type = Cons0 | Cons1 of t1 | Cons2 of (t21 * t22) |...
```

- Les noms des constructeurs commencent par une majuscule (réciproquement, les identifiants quelconques ne peuvent pas commencer par une majuscule)
- Ils prennent zéro, un, deux… « paramètres » pour désigner les sous-types des arguments associés (*cf.* syntaxe ci-dessus, ci-dessous)

Cas paramétrés :

```
type 'a t = Cons1 of 'a
type ('a, 'b) t = Cons1 of 'a | Cons2 of 'b | Cons3 of ('a * 'b)
```

Types somme en OCaml *bis* : exemples

Redéfinition de types standards :

Type option

```
type 'a option = None | Some of 'a
```

Type result

```
type ('a, 'e) result = Ok of 'a | Error of 'e
```

Type list

```
type 'a list = Nil | Cons of 'a * 'a list  (* pseudo version *)
type 'a list = [] | (::) of 'a * 'a list (* version standard *)
```

- La syntaxe `(::)` indique que `::` est un constructeur binaire *en notation infixe*. Avec `[]`, c'est le seul constructeur qui n'est pas un identifiant débutant par une majuscule (**à ne pas redéfinir**…)

Types somme en OCaml *ter*

Syntaxe de construction (sur des exemples)

```
(None:'a option)
(Some 12:int option)
Cons 12, Nil (* Error: The constructor Cons expects 2 argument(s),
                but is applied here to 1 argument(s) *)
Cons (12, Nil)
12::[] (* Équivalent à ci-dessus *)

Cons ((), Cons (12, Nil)) (* Error: The constant 12 has type int but
                             an expression was expected of type unit *)
Cons (11, Cons (12, Nil))
11::12::[] (* Équivalent à ci-dessus *)
```

Comment « extraire » ?

Avec du filtrage de motifs (*cf.* ci-dessous)

Table des matières

Intermède : renommage de type en OCaml

`type`

On peut *définir* de nouveaux types en OCaml avec `type`, mais également renommer (ou « aliaser ») des types existants :

```
utop # type poney = int;;
type poney = int
utop # 12;;
-: poney = 12
```

Ceci est à éviter à tout prix pour les types de bases (aucun intérêt), mais peut éventuellement être utile pour simplifier l'écriture de types produits

- Attention néanmoins à ce que cela ne nuise pas à la compréhension !
- (Mon avis : rarement utile / une bonne idée)

Table des matières

Motifs

Types $\subseteq$ syntaxe

Les types appartiennent au « monde syntaxique » ; ils décrivent la *forme* des objets, sans s'intéresser à leur *sens* (c'est le rôle d'une *sémantique*)

- Exemple : `([1;2;3]:int list)` et `((1,2,3):int*int*int)` n'ont pas la même forme, quand bien-même on pourrait leur prêter le même sens

Problématique

Un type somme permet de définir (généralement) plusieurs formes pour ses habitants ; un mécanisme est nécessaire pour distinguer ces cas

Motifs

Un *motif* est une construction syntaxique permettant de décrire l'ensemble des objets d'une certaine forme (les motifs ne sont pas des expressions)

- Premier exemple (bien d'autres à suivre) :

  `_::_`
  (motif des `'a list` non vides)

Filtrage de motifs

`match with`

OCaml fournit une **expression** de filtrage de motif (en anglais : *pattern matching*) qui permet d'appliquer un traitement par cas en fonction des motifs

Syntaxe (version élémentaire) :

```
match e with
| <pat1> -> e1 (* premier | et retour à la ligne non significatif *)
| <pat2> -> e2
...
```

où `e` est une expression de type α, `<pat1>`, `<pat2>`… des motifs pour le type α, et `e1`, `e2`… des expressions d'un même type β (éventuellement égal à α)

Sémantique : s'évalue en (l'évaluation de) `e1` si (l'évaluation de) e correspond au motif `<pat1>`, en […] `e2` si […] au motif `<pat2>` **et pas au motif** `<pat1>`…

Exhaustivité

Un filtrage doit (pour nous) être exhaustif : l'ensemble des cas doit traiter toutes les formes possible pour α (mais on peut écrire un cas générique, *cf.* ci-dessous)

Motifs : règles d'écriture

Un motif définit la forme d'un objet : il s'écrit en faisant appel aux constructeurs

Respect de l'arité

L'ensemble des objets construits avec un constructeur `Cons` se dénote par le motif :

- `Cons` si `Cons` est zéro-aire (ne prend aucun argument)
- `Cons x` si `Cons` est unaire (prend un argument)
- `Cons (x, y)` si `Cons` est binaire

etc.

Où `x`, `y`… sont des identifiants (quelconques, qui doivent être *distincts*) **nouvellement** liés aux arguments utilisés lors de la construction, qui peuvent être utilisés dans l'expression associée au cas dans un filtrage : ils permettent la déconstruction

Motifs : règles d'écriture *bis*

Attention : les identifiants introduits dans un motif **occulteront tout éventuel identifiant du même nom**

Exemple

`fun x y -> match y with Some x -> true` ... ne fait probablement pas ce que l'on veut (à moins d'être vicieux)

wildcard

Tout identifiant peut être remplacé par la *wildcard* « _ » quand on ne souhaite pas vraiment l'utiliser

Exemple : `match x with Some _ -> true | None -> false`

Motifs : précision

On peut aussi écrire des motifs plus ou moins précis, par ex.

- `x` (ou `_`) est *universel* (satisfait par tout le monde)
- `x::xs` (ou `_::_`) est satisfait par les listes non vides
- `x1::x2::xs` (ou `_::_::_`) est satisfait par les listes de *longueur* au moins deux
- `(x1, x2)::xs` (ou `(_, _)::_`) est satisfait par les listes non vides contenant des paires

Règle pour satisfaire un motif

Les identifiants dans un motif peuvent être utilisés dans une expression : les valeurs satisfaisant un motif sont seulement celles qui peuvent être construites en substituant une expression valide aux identifiants (éventuellement « _ ») du motif

Exemple : `[1;2;3]` `(* 1::2::3::[] *)`

- satisfait le motif `_::_` ou `x::xs` (peut-être construite comme `x::xs` en prenant `x` égal à `1` et `xs` égal à `[2;3]`)
- ne satisfait pas le motif `_::[]` ou `x::[]` (il faudrait avoir `x` égal à `1::2::3`, qui n'est pas une expression valide)

Motifs et filtrage : bilan intermédiaire

Utilité

Le filtrage de motif permet de :

- Distinguer plusieurs cas d'un type somme
- Déconstruire les valeurs (obtenues par application d'un constructeur…)

C'est une opération essentielle en OCaml (où les valeurs manipulées ont très souvent un type somme)

Erreurs de filtrage classiques

- Filtrage non exhaustif (peut provoquer des erreurs à l'exécution)
- Cas redondants (nuisent à la lisibilité ; probablement pas ce que l'on veut). Par ex. :

  ```
  match x with _::_ -> 1 | _::[] -> 2 | [] -> 3
  ```

  le cas du milieu ne sera **jamais** évalué car inclus dans le précédent
- Occultation d'identifiants
- Vouloir filtrer sur des « motifs » d'application de fonction :

  ```
  match x with 2 * _ -> true | _ -> false (* Error: Syntax error *)
  ```

Le compilateur/interpréteur émet un avertissement dans les deux premiers cas (mais pas (directement) le troisième)

Premiers exemples

```
(* déconstruction d'un type produit *)
match (1, 2, 3) with x, _, _ -> x (* 1 *)

match Some 12 with Some _ -> true | None -> false (* true *)

match None with None -> false (* false *)
(* Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched: Some _ *)

match Some 12 with None -> false (* Exception: Match_failure *)

match Some (11,12) with Some (_, x) -> x | None -> 0 (* 12 *)

match [12] with x::_ -> Some x | _ -> None (* Some 12 *)

match [] with x::_ -> Some x | _ -> None (* None *)
```

Exercice

Expliquez la fonction suivante :

```
let rec len v = match v with [] -> 0 | _::xs -> 1 + (len xs)
```

(Quel type ; quelles spécifications ; comment fonctionne-t'elle ?)

Filtrage : motifs littéraux

On peut écrire des motifs qui représentent **une** valeur, et utiliser un filtrage comme un `if` à cas multiples. Par ex. :

```
match x with [1] -> 1 | [1;2] -> 12 | [1;2;3] -> 123 | _ -> 0
```

- Mais jamais *nécessaire* (on « sait » déjà déconstruire), et rarement plus simple/clair qu'un `if`
- À utiliser avec une certaine parcimonie

Filtrage : `match` imbriqués

Un `match` est une expression : il peut se retrouver dans l'expression traitant un cas d'un autre `match`

Il est souvent nécessaire (ou plus clair) de parenthéser un `match` interne pour délimiter ses cas. Exemple :

```
match v with
    | x::_ -> (match x with Some y -> y | None -> 0)
    | _ -> 0
```

On peut parfois s'économiser des imbrications en utilisant des motifs plus précis, par ex. :

```
match v with
   | (Some x)::_ -> x
   | _ -> 0
match (v, w) with
   | ([], []) -> ....
```

Filtrage : re: exhaustivité

Un filtrage doit être exhaustif

- On peut toujours inclure un cas « par défaut » `_ -> ...` pour assurer l'exhaustivité d'un filtrage
- Si présent, celui-ci doit **toujours** se trouver à la fin (pourquoi ?)

Problème

On peut avoir des filtrages où l'on sait/a prouvé que certains cas *ne se produisent jamais*

- Il faut néanmoins les inclure dans le filtrage
- Mais quelle expression leur associer ?
  - Qui soit pertinente ? Qui respecte les contraintes de type ?

Filtrage : re: exhaustivité et `assert false`

Sur un exemple

On souhaite écrire un filtrage d'une expression (`v:'a list`) *que l'on sait non vide* et qui s'évalue en la tête de celle-ci

- Solution 1 : `match v with x::_ -> x`
  Défaut : non exhaustif
- Solution 2 : `match v with x::_ -> Some x | _ -> None`
  Défaut : change le type de l'expression (nécessitera *encore* un `match`)
- « Solution » 3 : `match v with x::_ -> x | _ -> 0`
  Défauts : change radicalement le type ; pas clair à la lecture (la pire de toutes)
- Solution 4 : `match v with x::_ -> x | _ -> assert false`
  Défauts : aucun \o/

`assert false` : principe (rapide)

Expression dont l'évaluation *lève* (en anglais : *raises*) une *exception* (en l'occurrence : `Assert_failure`) qui interrompt l'évaluation : celle-ci ne termine jamais, et l'on peut donc typer `assert false` en n'importe quel type

Plus sur `assert` et les exceptions… un peu plus tard…

Filtrage : syntaxes alternatives

`function`

Un cas courant : vouloir écrire `fun v -> match v with <pat1> -> ...` où v n'est pas réutilisé dans le `match`

On peut dans ce cas utiliser la syntaxe raccourcie `function <pat1> -> ...`

Exemple

```
fun v -> match v with x::_ -> x | _ -> assert false
function x::_ -> x | _ -> assert false (* équivalent *)
```

Filtrage : syntaxes alternatives *bis*

Il peut arriver que plusieurs cas partagent le même traitement (sont associés à *exactement* la même expression) :

```
match v with
| <pat1> -> e1
| <pat2> -> e1
| ...
```

On peut dans ce cas utiliser la syntaxe allégée :

```
match v with
| <pat1> | <pat2> -> e1
| ...
```

Exemple

```
type ('a, 'b, 'c) t = O of 'a | Tw of 'a * 'b | Th of 'a  * 'b * 'c
function O x | Tw (x, _) | Th (x, _, _) -> x
```

Filtrage : syntaxes alternatives *ter* : filtrage dans les identifiants

On peut filtrer directement sur les identifiants (dans un `let` dans les paramètres d'une `fun`, mais ce filtrage **est souvent non exhaustif**)

Cas exhaustifs courants : déconstruction des types sommes à **un** cas ; des types produits (qui en fait cachent du filtrage) !!

Exemples purement syntaxiques

```
let v = Some 12
let Some x = v (* x lié à 12, mais filtrage non exhaustif ! *)
let _ = 3 (* okay *)
fun (Some x) -> true (* non exhaustif ; argument None non traité *)
fun (x, y, z) -> x (* okay *)
```

Exemples avec motifs littéraux

```
let 3 = 3 (* non exhaustif mais pas d'erreur possible *)
fun 3 -> true (* non exhaustif ; args de valeur <> 3 non traités *)
let () = if true then () (* exhaustif *)
fun () -> 0 (* pareil *)
```

Filtrage : syntaxes alternatives *quater* : réécriture des `let in` et `fun`

(À quelques subtilités près)

```
let p = v in e
```

avec p un motif (rappel : un nom d'identifiant est un motif universel) correspond exactement à :

```
match v with p -> e
```

et donc à :

```
(function p -> e) v
```

De même :

```
fun p -> e
```

correspond exactement à :

```
function p -> e
```

Arité des constructeurs : subtilité

On considère :

```
type t1 = Cons1 of int * int
type t2 = Cons2 of (int * int)
```

- `Cons1` est un constructeur *binaire* acceptant deux arguments `int`
- `Cons2` est un constructeur *unaire* acceptant un argument `int * int`

Impact dans l'écriture des motifs :

```
function Cons1 x -> x (* Error: The constructor Cons1 expects 2
argument(s), but is applied here to 1 argument(s) *)
function Cons1 (x,y) -> (x,y) (* fine *)
function Cons2 x -> x (* fine *)
function Cons2 (x,y) -> (x,y) (* actually still fine *)
```