Programmation Fonctionnelle (PF)

INFO4

Cours 2 : fonctions, sommes et filtrage, récurrence

Jean-François Monin, Benjamin Wack

POLYTECH GRENOBLE

Plan

Structuration des données : sommes et produits

Deux formes essentielles de structuration de données

- Juxtaposition : un truc avec un machin
  couples, n-uplets, *records*, tableaux, produit cartésien
  Disponibles (primitifs) dans tous les langages de programmation
- Choix : un truc ou un machin
  grand oublié des langages usuels
  en tant que structure de composition
  - bit (chiffre binaire ; booléen)
  - entiers, données atomiques
  - énumération
  - pointeur vide ou alloué (*)

  Réhabilité (primitif) dans les langages fonctionnels typés comme OCaml

Structuration des données : sommes et produits

Connexion profonde avec la logique (*)

- Juxtaposition $\rightarrow$ conjonction
  Une preuve de $A \wedge B$ est obtenue à partir d'une preuve de A juxtaposée avec une preuve de B
- Choix $\rightarrow$ disjonction
  Une preuve de $A \vee B$ est obtenue à partir d'une preuve de A ou d'une preuve de B

En pratique

Aide à la conception et au raisonnement

Structuration des données : questions essentielles

- Comment construire
- Comment utiliser
- Comment calculer

Structuration des données : questions essentielles

Comment construire

- un couple, un n-uplet, etc.
- une somme : constructeurs
- une fonction `fun ... -> ...`

En logique (*) : principe d'introduction

Structuration des données : questions essentielles

Comment utiliser (décomposer, analyser)

- un couple, un n-uplet, etc. : projection
  récupération du premier, second,... composant
- une somme : filtrage `match`
- une fonction : application à un argument

En logique (*) : principe d'élimination

Structuration des données : questions essentielles

Comment calculer (réduire)

Confrontation d'une construction et d'une décomposition

- projection à partir un couple, un n-uplet, etc.
- filtrage d'une valeur dans un type somme
- fonctions : substitution des paramètres effectifs aux paramètres formels

En logique (***) : simplification par élimination des coupures

Sommes généralisées

Un cas peut embarquer plusieurs composants
On a donc une somme de produits

Exemples de type somme

```
type legume = Haricot | Carotte | Courge
type fleur = Rose | Hortensia
type fruit = Poire | Banane
type couleur = Rouge | Jaune | Blanc
type plante =
   | Legume of legume
   | Fleur of fleur * couleur
   | Fruitier of fruit
type objet =
  | Plante of plante
  | ...
```

Représentation graphique (au tableau)

Arbres

Type somme et filtrage

Un type définit exhaustivement les valeurs possibles après réduction
Le filtrage couvre toutes ces valeurs par des motifs

Motif

Arbre à trous, où chaque trou représente un sous-arbre quelconque (du type approprié)

```
match o with
  | Plante (Fleur (f, c)) -> ... f ... c...
  | ...
```

Représentation graphique

Motif de filtrage arborescent

Recette pour faire des motifs

Le bon, la brute et le truand

L'humanité est divisée en deux catégories : il y a ceux qui tiennent un pistolet chargé et il y a ceux qui creusent. Toi, tu creuses

Recette

- Prendre un arbre
- Creuser
- Nommer les trous : `x`, `y`, `n`, `l`...
  (ou les laisser anonymes : `_`)

Avantage du filtrage sur le if

Assurance gratuite que l'on baigne dans le bon environnement

Voir plus loin

Ce qu'effectue le filtrage

- Reconnaissance de forme
- En cas de succès : nommage de sous-arbres
  (liaison de sous-arbres à des noms)

Exhaustivité

Toutes les valeurs possibles du type doivent être couvertes
Rappel : une valeur est ce qu'on obtient après calcul

Match est ordonné

```
let foo y = match y with
| Nil -> ... (* 1er test *)
| Cons (a, e) -> ... (* 2eme test*)
| Cons (a, Nil) -> ... (* jamais atteint*)
```

En OCaml les motifs sont évalués de haut en bas

Motifs du match

Un motif ne peut contenir que des constructeurs

- constructeurs de somme
- couple, triplet ...
- listes : `Cons` et `Nil` (en bibliothèque : `::` et `[]`)

Un motif doit être linéaire : pas de `Constr(... x ... x ...)`

Pas d'opération nécessitant un calcul

- pas d'appel de fonction, y compris `+`, `&&`, `@`
- pas de `if` , de **let** , de `match` ,...

Motif universel (ou joker)

Une variable `x` est un motif universel :
capture tous les cas (non encore capturés par un motif précédent)

L'underscore `_` est une variable “qui s'oublie”
(aucune liaison n'est créée)

Dans un motif l'underscore `_` est un motif universel qui ne crée pas de liaison

```
(x, _) ->
|_ ->
```

Match est exhaustif

Sinon Warning et exception !

Exemple

```
# let foo p = match p with Legume(l) -> 1
Warning P : this pattern-matching is not exhaustive. Here is an
example of a value that is not matched : Fruitier (Poire)
val foo : plante -> int = <fun>
```

```
# foo (Fruitier (Poire)) Exception : Match_failure ("", 1, 11).
```

Match : partage de résultats pour plusieurs cas

Exemple : moitié d'un chiffre décimal

```
match x with
| 0 | 2 | 4 | 6 | 8  -> x/2
| 1 | 3 | 5 | 7 | 9  -> (x+1)/2
|_  -> 0
```

Match : nommage d'un sous-motif

Exemple

```
match a with
| N (N (g1, x1, d1) as gxd1,
     y, N (g2, x2, d2))  -> ... gxd1 ...
|_  -> ...
```

Match conditionnel (when) – DÉLICAT

Égalité

```
# let eg x = match x with
(a, b) when a = b -> true
|(a, b) -> false
```

1. évaluation de la condition du `when`
2. si vrai, l'expression associée est évaluée
3. sinon passage au cas suivant.

```
eg (1, 1)
 - : bool =
true
eg (1,3)
 - : bool =
false
```

Match implicite : **let** filtrant I

Somme à un cas

- **type** eb = EB `of` `int` × `bool`
  `match` x `with` EB(n, b) →. . . n . . . b . . .
  ≡
  let EB(n, b) = x `in` . . . n . . . b . . .

Attention à l'inversion de l'ordre

Match implicite : **let** filtrant II

Filtrage sur un couple

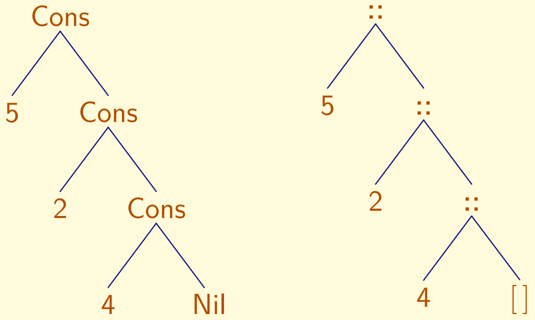
- `match` couple `with` $(n, b) \to \ldots n \ldots b \ldots$
  $\equiv$
  let $(n, b) =$ couple `in` $\ldots n \ldots b \ldots$

Type somme récursif : listes

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

Exemple : [5 ; 2 ; 4]

Cons (5, Cons (2, Cons (4, Nil)))

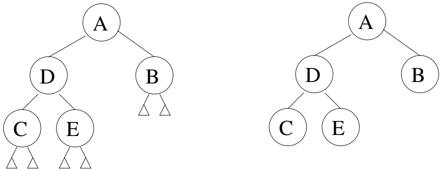


Un type récursif arbre binaire

Arbre binaire

Un arbre **binaire** est :

- soit l'arbre vide ;
- soit un nœud constitué d'une *étiquette* et de 2 *sous-arbres* binaires (gauche et droit).



On suit cette définition pour modéliser les arbres en OCaml :

```
type arbre = FV | N of arbre * int * arbre
```

Programmation par analyse de cas

Canevas

- Chercher une solution pour chaque cas
- Dans chaque cas, on peut appliquer des fonctions auxiliaires sur ses composants

Programmation par analyse de cas

Exemples

- tester qu'une liste est vide
- tester qu'un arbre est vide
- queue et tête d'une liste
- sous-arbres gauche ou droit

Programmation par récursion structurelle

Canevas

- Chercher une solution pour chaque cas
- Dans chaque cas, on peut appliquer des fonctions auxiliaires ou la fonction en cours de définition sur ses composants
- idée sous-jacente : supposons le résultat obtenu sur
  - la queue de la liste considérée
  - les sous-arbres gauche et droit de l'arbre considéré
  - etc.

  construisons le résultat de la liste (de l'arbre) considéré(e) au moyen de ces résultats intermédiaires

Programmation par récursion structurelle

Programmation impérative

Que faire (dans tel cas) ?

Programmation fonctionnelle

Que vaut le résultat (dans tel cas) ?

Exemples

- longueur d'une liste
- concaténation de 2 listes
- liste des clés d'un arbre
- taille d'un arbre (nombre de feuilles, nombre de nœuds)

Exemple : récursion structurelle sur une liste

Idée sous-jacente

On donne le résultat sur

- la liste vide
- une liste `x :: q` en supposant connu le résultat sur la queue `q`

On aura ainsi le résultat sur n'importe quelle liste.

Raisonner par récurrence structurelle

C'est pareil que programmer par récursion structurelle.

Exemple : récurrence structurelle sur une liste

Idée sous-jacente

Si on peut démontrer une propriété sur

- la liste vide
- une liste `x :: q` en supposant la propriété démontrée sur la queue `q`

On aura ainsi une preuve de la propriété sur n'importe quelle liste.

Récurrence sur les listes

Soit P un prédicat sur les listes.

Si $\left.\begin{array}{r} P([]) \\ \text{et } \forall x, q,\ P(q) \Rightarrow P(x :: q) \end{array}\right\}$ on peut en déduire que $\forall l,\ P(l)$.

Prouver que le bégaiement double la longueur

$P(l) \ \stackrel{\text{déf}}{=\!=} \ \text{long}(\text{begaie } l) = 2 \times \text{long } l$

```
let rec begaie l = match l with
  | [] → []
  | x :: q → x :: x :: (begaie q)
```

```
let rec long l = match l with
  | [] → 0
  | x :: q → 1 + (long q)
```

Prouver : $\forall l \ P(l)$

- $\text{long}(\text{begaie } []) = \text{long}([]) = 0 = 2 \times 0 = 2 \times \text{long } []$
- $\forall\, q\, x$, hypothèse de récurrence : $\text{long}(\text{begaie } q) = 2 \times \text{long } q$

$$\begin{aligned}
\text{long } (\text{begaie } (x :: q)) &= \text{long } (x :: x :: \text{begaie } q) \\
&= 1 + 1 + \text{long } (\text{begaie } q) \\
&= 1 + 1 + \ 2 \times \text{long } q \qquad \textit{(hyp rec)} \\
&= 2 \times (1 + \text{long } q) \\
&= 2 \times (\text{long } (x :: q))
\end{aligned}$$

Raisonnement par récurrence structurelle

Idée sous-jacente

On donne le résultat sur

- l'arbre élémentaire `FV`
- un arbre `N(g, x, d)` en supposant connu le résultat sur les sous-arbres `g` et `d`

On aura ainsi le résultat sur n'importe quel arbre binaire.

Récurrence sur les arbres binaires

Soit P un prédicat sur les arbres binaires

$$\left.\begin{array}{l}\text{De } P(\mathsf{FV}) \\ \text{et } \forall g, x, d,\ P(g) \wedge P(d) \Rightarrow P(\mathsf{N}(g, x, d))\end{array}\right\} \text{ on infère } \forall a,\ P(a).$$

Ceci se généralise à tous les types somme inductifs.

Preuve sur le nombre de feuilles et de clés

$P(a) \stackrel{\text{déf}}{=\!=} \text{nbf } a = \text{nbc } a + 1$

```
let rec nbf a = match a with
  | FV → 1
  | N (g, x, d) → nbf g + nbf d
```

```
let rec nbc a = match a with
  | FV → 0
  | N (g, x, d) → nbc g + 1 + nbc d
```

Prouver $\forall a\ P(a)$ par récurrence structurelle sur a.

- $\text{nbf FV} = 1 = 0 + 1 = \text{nbc FV} + 1$
- Soient g, x et d tels que $\text{nbf } g = \text{nbc } g + 1$ et idem pour d.

$$\begin{aligned}
\text{nbf N } (g, x, d) &= \text{nbf } g + \text{nbf } d \\
&= (\text{nbc } g + 1) + (\text{nbc } d + 1) && \textit{(hyps rec)} \\
&= (\text{nbc } g + 1 + \text{nbc } d) + 1 \\
&= (\text{nbc N } (g, x, d)) + 1
\end{aligned}$$

Raisonnements par récurrence sur les entiers

Récurrence sur les listes

Soit P un prédicat sur les listes.
De $P([])$ et $\forall x, q,\ P(q) \Rightarrow\ P(x :: q)$ on infère $\forall l,\ P(l)$.

Récurrence sur les entiers naturels

Soit P un prédicat sur les entiers naturels.
De $P(0)$ et $\forall n,\ P(n) \Rightarrow\ P(n+1)$ on infère $\forall n,\ P(n)$.

Conceptuellement

```
type nat = Zero | Succ of nat
```

où $\mathrm{Succ}(n)$ représente $n+1$.

Programmation récursive sur les entiers

En pratique

Quelques différences entre `int` et `nat`

- représentation interne efficace
- `int` est borné
- `int` comprend des entiers négatifs

Programmation récursive sur `int`

Filtrage remplacé par :

- test à 0
- l'utilisation de `n-1` lors d'un appel récursif