PF chapître 2 : fonctions, sommes et filtrage, récurrence

Jean-François Monin

POLYTECH GRENOBLE

Plan

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 :

Structuration des données : questions essentielles

Comment construire

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

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. :

Structuration des données : questions essentielles

Comment utiliser (décomposer, analyser)

- un couple, un n-uplet, etc. : projection
  récupération du premier, second,... composant
- une somme :

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 :

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

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

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

Plan

Plan

Exemples de type somme

```
type legume = Haricot | Carotte | Courge
type fleur = Rose | Hortensia
type fruit = Poire | Banane
type couleur = Rouge | Jaune | Blanc
type plante =
      | Leg of legume
      | Flr of fleur * couleur
      | Arb of fruit
type objet =
      | Plt of plante
      | ...
```

Exemples de type somme

```
type legume = Haricot | Carotte | Courge
type fleur = Rose | Hortensia
type fruit = Poire | Banane
type couleur = Rouge | Jaune | Blanc
type plante =
      | Leg of legume
      | Flr of fleur * couleur
      | Arb of fruit
type objet =
      | Plt of plante
      | ...
```

Représentation graphique (au tableau)

Arbres

Type somme et filtrage

Une déclaration de type somme définit exhaustivement ses valeurs possibles (ce que l'on va obtenir après réduction)
Le filtrage couvre toutes ces valeurs par des motifs

Motif (ce qui est placé entre la barre ``|'' et la flèche ``->'')

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

```
match o with
  | Plt (Flr (f, c)) -> ... f ... c ...
  | ...
```

Représentation graphique

Motif de filtrage arborescent

Ce qu'effectue le filtrage

- Reconnaissance de forme efficace
- 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

Recette pour faire des motifs, localité

Recette

- Prendre un arbre
- Creuser

Recette pour faire des motifs, localité

Recette

- Prendre un arbre
- Creuser
- Nommer les trous : `x`, `y`, `n`, `l`...

Recette pour faire des motifs, localité

Recette

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

Recette pour faire des motifs, localité

Recette

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

Localité des noms

Les noms `x`, `y`, `n` sont locaux
(disponibles uniquement dans l'expression qui suit la flèche ``->''`).

Match est ordonné

```
match expr_de_plante with
| Flr (fl, Jaune) -> ...      (* fleur jaune *)
| Leg (l) -> ...              (* légume *)
| Flr (f, c) -> ...           (* fleur non jaune *)
| Leg (l) -> ...              (* légume *)
| Flr (Hortensia, c) -> ...   (* jamais atteint *)
| Arb (fr) -> ...             (* arbre *)
```

En OCaml les motifs sont évalués de haut en bas

Plan

Restrictions sur les motifs

Choix de conception important pour l'efficacité

Un motif ne peut contenir que des constructeurs

- constructeurs de somme déclarés dans les types
- couple, triplet ...
- listes de la bibliothèque standard : `[]` et `::`

Restrictions sur les motifs

Choix de conception important pour l'efficacité

Un motif ne peut contenir que des constructeurs

- constructeurs de somme déclarés dans les types
- couple, triplet ...
- listes de la bibliothèque standard : `[]` et `::`

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

Restrictions sur les motifs

Choix de conception important pour l'efficacité

Un motif ne peut contenir que des constructeurs

- constructeurs de somme déclarés dans les types
- couple, triplet ...
- listes de la bibliothèque standard : `[]` 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`,...

Filtrage >> `if` + projections

Conséquence de la localité des noms dans les motifs

Programmation par filtrage plus simple, sûre et efficace qu'un style basé sur des `if` (ou un `switch` du langage C) et des projections ad-hoc.

Idée développée en fin de chapître.

Plan

Motif universel, joker

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

Motif universel spécial : l'underscore `_`
à voir comme une variable "à oublier" (aucune liaison n'est créée)

Exemple :

```
| (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 ...
|_  -> ...
```

Motifs conditionnels (when) – DÉLICAT

```
type nombre = Ent of int | Flo of float

match expr with
| Ent (n) when n = 0                        -> 0
| Flo (x) when (10000 *. x *. x  < 1.)      -> 1
| _                                          -> 100
```

L'expression booléenne placée après when doit s'évaluer à true pour que le motif soit accepté.

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

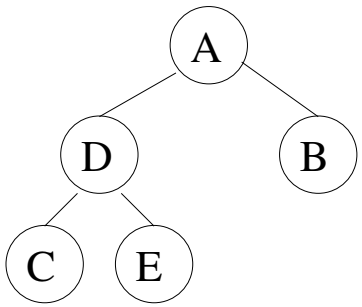
Filtrage sur un couple

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

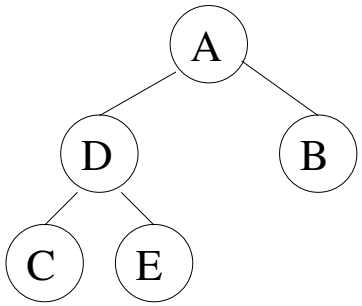
Plan

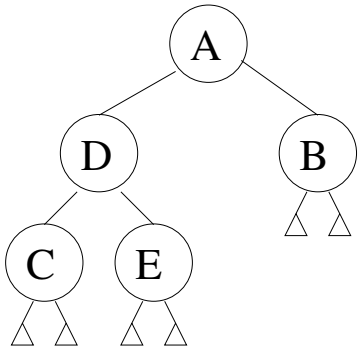
Plan

Un type somme récursif : arbre binaire

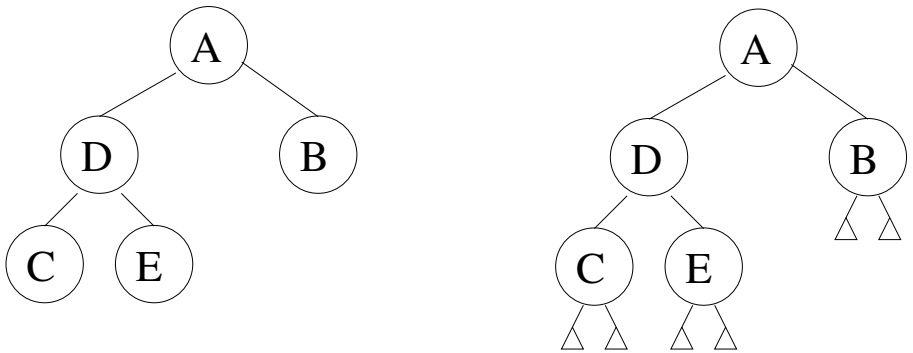


Un type somme récursif : arbre binaire



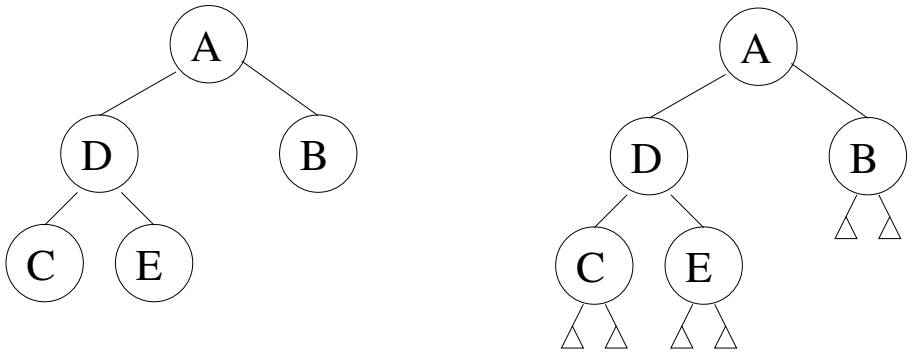


Un type somme récursif : arbre binaire



- soit l'arbre vide ;
- soit un nœud ternaire comportant 1 *étiquette* et 2 *sous-arbres*

Un type somme récursif : arbre binaire

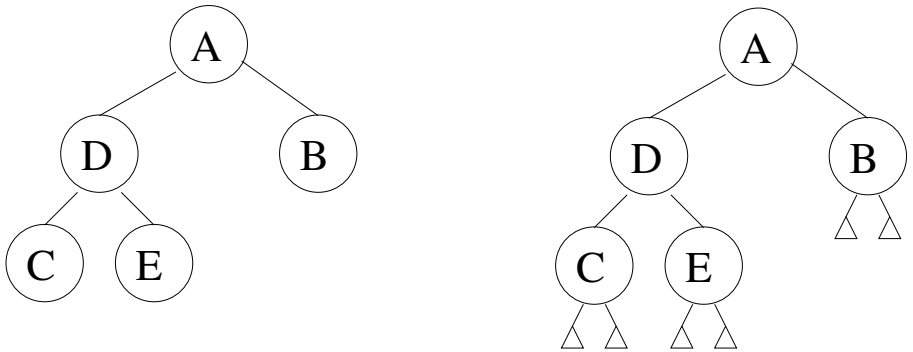


- soit l'arbre vide ;
- soit un nœud ternaire comportant 1 *étiquette* et 2 *sous-arbres*

En OCaml :

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

Un type somme récursif : arbre binaire



- soit l'arbre vide ;
- soit un nœud ternaire comportant 1 *étiquette* et 2 *sous-arbres*

En OCaml :

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

Dessiner l'arbre ci-dessus en faisant apparaître les constructeurs

Type somme récursif : liste

Liste = arbre unaire

- soit la liste vide ;
- soit un nœud binaire comportant 1 *étiquette* et 1 *sous-liste*

Détails plus bas

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

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
  - les sous-arbres gauche et droit de l'arbre considéré
  - la queue de la liste considérée
  - etc.

  construisons le résultat de l'arbre considéré au moyen de ces résultats intermédiaires

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
  - les sous-arbres gauche et droit de l'arbre considéré
  - la queue de la liste considérée
  - etc.

  construisons le résultat de l'arbre considéré au moyen de ces résultats intermédiaires

Nouveau mot clé

let rec

Programmation par récursion structurelle

Programmation impérative

Que faire (dans tel cas) ?

Programmation fonctionnelle

Que vaut le résultat (dans tel cas) ?

Programmation par récursion structurelle

Programmation impérative

Que faire (dans tel cas) ?

Programmation fonctionnelle

Que vaut le résultat (dans tel cas) ?

Exemples

- nombre de feuilles vides d'un arbre binaire
- nombre de clés d'un arbre binaire

Exemple : nombre de feuilles ou de clés d'un arbre

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

Exemple : nombre de feuilles ou de clés d'un arbre

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

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

Plan

Raisonner par récurrence structurelle...

... c'est pareil que

programmer par récursion structurelle

Programmation récursive et 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.

Programmation récursive et récurrence structurelle

Idée sous-jacente

Si on peut démontrer une propriété sur

- l'arbre vide,
- tout arbre N (g, x, d) en supposant la propriété démontrée sur les sous-arbres g et d,

on aura alors une preuve de la propriété sur n'importe quel arbre binaire de ce type.

Programmation récursive et récurrence structurelle

Idée sous-jacente

Si on peut démontrer une propriété sur

- l'arbre vide,
- tout arbre $\mathsf{N}\,(g, x, d)$ en supposant la propriété démontrée sur les sous-arbres g et d,

on aura alors une preuve de la propriété sur n'importe quel arbre binaire de ce type.

Récurrence sur les arbres binaires

Soit P un prédicat sur les arbres binaires

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

Preuve sur le nombre de feuilles et de clés

$$P(a) \quad \underline{\underline{\text{déf}}} \quad \text{nbf } a = \text{nbc } a + 1$$

Preuve sur le nombre de feuilles et de clés

$$P(a) \quad \underline{\underline{\text{déf}}} \quad \text{nbf } a = \text{nbc } a + 1$$

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

```
let rec nbc = fun 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.

Preuve sur le nombre de feuilles et de clés

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

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

```
let rec nbc = fun 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.

- nbf FV $= 1 = 0 + 1 =$ nbc FV $+ 1$

Preuve sur le nombre de feuilles et de clés

$$P(a) \quad \underline{\underline{\text{déf}}} \quad \text{nbf } a = \text{nbc } a + 1$$

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

```
let rec nbc = fun 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.

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

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

Programmation récursive et récurrence structurelle

Ce principe est valable sur tous les types somme récursifs

Plan

Listes

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

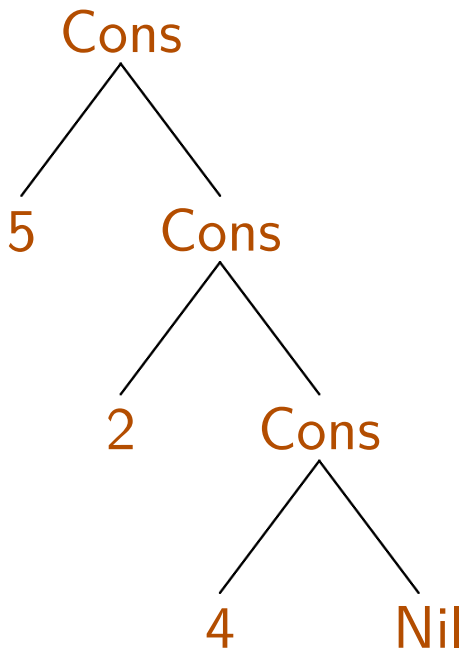
Exemple : [5 ; 2 ; 4]

Listes

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

Exemple : [5 ; 2 ; 4]

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

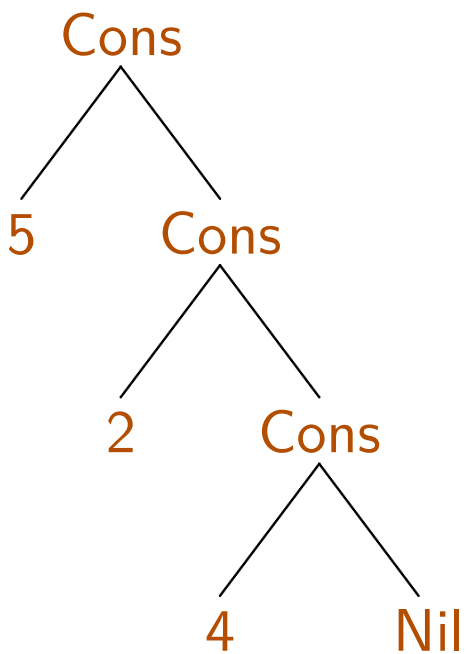


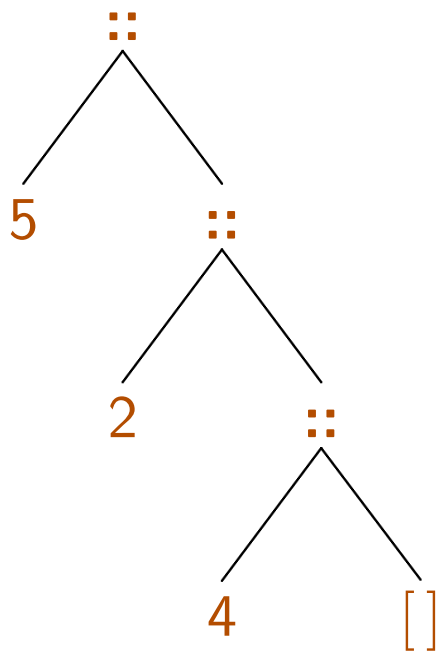
Listes

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

Exemple : [5 ; 2 ; 4]

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





Filtrage sur les listes

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

match l with
  | Nil → true
  | Cons (x, l) → false

match l with
  | [] → 0
  | x :: l →  1 + longueur l
```

Récursion et récurrence structurelle sur les listes

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.

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 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.

De $\left.\begin{array}{l} P\,[] \\ \text{et } \forall x\, q,\ P\,q \Rightarrow P(x :: q) \end{array}\right\}$ on infère $\forall l,\ P\,l$.

Prouver que le bégaiement double la longueur

$$P(l) \quad \underline{\underline{\text{déf}}} \quad \text{long (begaie } l) = 2 \times \text{long } l$$

Prouver que le bégaiement double la longueur

$$P(l) \quad \underline{\underline{\text{déf}}} \quad \text{long (begaie } l) = 2 \times \text{long } l$$

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

Prouver que le bégaiement double la longueur

$$P(l) \quad \underline{\underline{\text{déf}}} \quad \text{long (begaie } l) = 2 \times \text{long } l$$

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

- long (begaie []) = long ([]) = $0 = 2 \times 0 = 2 \times$ long []

Prouver que le bégaiement double la longueur

$P(l) \quad \underline{\underline{\text{déf}}} \quad \text{long (begaie } l) = 2 \times \text{long } l$

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

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

Prouver que le bégaiement double la longueur

$P(l) \quad \underline{\underline{\text{déf}}} \quad \text{long (begaie } l) = 2 \times \text{long } l$

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

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

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

Plan

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 $\forall l,\ P\,l$.

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 $\forall n,\ P\,n \Rightarrow P\,(1 + n)$
on infère $\forall n,\ P\,n$.

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 $\forall n,\ P\,n \Rightarrow P\,(1+n)$
on infère $\forall n,\ P\,n$.

Conceptuellement

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

où Succ (n) représente $1+n$.

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

Plan

Se passer du filtrage : `if` + projections (1)

Prenons un langage de programmation ordinaire. Les données de type `plante` vont être représentées par une mixture de légumes, de fleurs colorées et de fruits. Intuitivement :

`mixture` = `legume` $\cup$ (`fleur` * `couleur`) $\cup$ `fruit`

Dans un langage à la C :

- un `struct` avec des champs `legume`, `fleur`, `couleur` et `fruit`
- plus économique en mémoire : un type `union`

On se donne des projections permettant de récupérer, étant donné une plante `p`, le légume, la fleur, la couleur, le fruit sous-jacent à `p` lorsque ce composant a un sens.
On écrira par exemple : `p.legume`

Se passer du filtrage : `if` + projections (2)

Et on se donne un moyen de tester l'espèce de plante de `p` en utilisant un champ supplémentaire `sorte` dont le type est une énumération.

```
type enum_plante = Leg  |  Flr  |  Arb
```

```
if p.sorte = Leg then ...  p.legume ...
else if p.sorte = Flr then ... p.fleur ... p.couleur ...
else if p.sorte = Arb then ... p.fruit ...
```

Problème

Dans un programme quelconque rien ne garantit que les projections sont utilisées à bon escient, par exemple que `p.couleur` n'est invoqué qu'avec l'assurance que `p.sorte = Flr`.

Se passer du filtrage : `if` + projections (3)

Solution...

Vérifier systématiquement qu'on est dans le bon cas d'utilisation d'une projection au moyen de fonctions

```
let couleur_of = fun p →
    if p.sorte = Flr then p.couleur
    else levée d'exception
```

Se passer du filtrage : `if` + projections (3)

Solution...

Vérifier systématiquement qu'on est dans le bon cas d'utilisation d'une projection au moyen de fonctions

```
let couleur_of = fun p →
    if p.sorte = Flr then p.couleur
    else levée d'exception
```

... inefficace !

- test répété à chaque utilisation
- inutile de faire le test pour `p.couleur` puis pour `p.fleur`

Se passer du filtrage : `if` + projections (3)

Solution...

Vérifier systématiquement qu'on est dans le bon cas d'utilisation d'une projection au moyen de fonctions

```
let couleur_of = fun p →
    if p.sorte = Flr then p.couleur
    else levée d'exception
```

... inefficace !

- test répété à chaque utilisation
- inutile de faire le test pour `p.couleur` puis pour `p.fleur`

... imparfaite !

Rien n'empêche d'utiliser `p.couleur` à la place de `couleur_of`

En bref : dilemne infernal

On est constamment empoisonné par l'obligation de justifier le droit d'accéder à tel ou tel composant.

Bienfaits silencieux du filtrage (1)

Les problèmes précédents disparaissent tout seuls

- assurance gratuite que les composantes visées existent et ont un sens
- elles sont récupérées une fois pour toutes

```
type plante =
  | Leg of legume
  | Flr of fleur * couleur
  | Arb of fruit

match p with
  | Leg (l) => ... l ... l ...
  | Flr (fl, c) =>  ... c ... fl ... c ... fl ...
  | Arb (fr) => ... fr ...
```

Bienfaits silencieux du filtrage (2)

Le filtrage fournit des équations simples

```
let truc = fun a → match a with
    | FV → 0
    | N (g, x, d) → machin g + 1 + chouette d
```

On a les équations

- `truc (FV) = 0`
- `truc (N (g, x, d)) = machin g + 1 + chouette d`

Résumé

Le filtrage délivre le programmeur de l'obligation d'effectuer des raisonnements fastidieux.

Techniquement :
obligations de preuve remplacées par du typage statique.

Analogie entre filtrage (+ recursivité) et équations (récursives).