UNIVERSITÉ Grenoble Alpes

INF201
Algorithmique et Programmation Fonctionnelle
Cours 10 : Ordre supérieur (fin)

Année 2022

Plan

Quelques rappels

Ordre supérieur sur les listes

Curryfication

Rappels du cours précédent : types polymorphes

Utilisation de types ou fonctions avec types paramètres (notation `'x`)

Syntaxe

```
type ('x, 'y, ...) nom_type = ... (* def. du type avec 'x, 'y, ..
   *)
```

Rappels du cours précédent : types polymorphes

Utilisation de types ou fonctions avec types paramètres (notation `'x`)

Syntaxe

```
type ('x, 'y, ...) nom_type = ... (* def. du type avec 'x, 'y, ..
   *)
```

Exemples

- types produits :

```
type 'x cple = 'x * 'x
      (2.3, 4.5) : float cple    (true, false) : bool cple
type ('x, 'y, 'z) tple = 'x * 'y * 'z
      (3, true, 4) : (int, bool, int) tple
```

Rappels du cours précédent : types polymorphes

Utilisation de types ou fonctions avec types paramètres (notation `'x`)

Syntaxe

```
type ('x, 'y, ...) nom_type = ... (* def. du type avec 'x, 'y, ..
   *)
```

Exemples

- types produits :

```
type 'x cple = 'x * 'x
      (2.3, 4.5) : float cple    (true, false) : bool cple
type ('x, 'y, 'z) tple = 'x * 'y * 'z
      (3, true, 4) : (int, bool, int) tple
```

- types sommes :

```
type ('x, 'y) somme = C1 of 'x | C2 of 'y
      C1(5) : (int, 'x) somme   C2('A') : (char, 'y) somme
type 'x ens = Nil | Cons of 'x * 'x ens
      Cons (5, Cons(2, Nil)) : int ens
```

Rappels du cours précédent : types polymorphes

Utilisation de types ou fonctions avec types paramètres (notation `'x`)

Syntaxe

```
type ('x, 'y, ...) nom_type = ... (* def. du type avec 'x, 'y, ..
   *)
```

Exemples

- types produits :
  `type 'x cple = 'x * 'x`
  (2.3, 4.5) : `float cple` (`true`, `false`) : `bool cple`
  `type ('x, 'y, 'z) tple = 'x * 'y * 'z`
  (3, `true`, 4) : (`int`, `bool`, `int`) `tple`
- types sommes :
  `type ('x, 'y) somme = C1 of 'x | C2 of 'y`
  `C1`(5) : (`int`, `'x`) `somme` `C2`('A') : (`char`, `'y`) `somme`
  `type 'x ens = Nil | Cons of 'x * 'x ens`
  `Cons` (5, `Cons`(2, `Nil`)) : `int ens`
- listes (prédéfinies) polymorphes (`'x list`) :
  [1; 2 ; 3 ; 4 ; 5] : `int list`
  ['A' ; 'B'] : `char list`

Rappels du cours précédent : fonctions polymorphes

La définition de la fonction ne dépend pas (complètement) du type de ses paramètres . . .

Syntaxe

```
let f (p1 : 'x) (p2 : 'y) (...) : 'r = ... (* corps de f *)
```

Rappels du cours précédent : fonctions polymorphes

La définition de la fonction ne dépend pas (complètement) du type de ses paramètres . . .

Syntaxe

```
let f (p1 : 'x) (p2 : 'y) (...) : 'r = ... (* corps de f *)
```

Exemples

- identité

```
let identite (p1 : 'x) : 'x = p1
```

Rappels du cours précédent : fonctions polymorphes

La définition de la fonction ne dépend pas (complètement) du type de ses paramètres . . .

Syntaxe

```
let f (p1 : 'x) (p2 : 'y) (...) : 'r = ... (* corps de f *)
```

Exemples

- identité

```
let identite (p1 : 'x) : 'x = p1
```

- coupler 2 valeurs

```
let coupler (p1 : 'x) (p2 : 'y) : 'x * 'y = (p1, p2)
```

Rappels du cours précédent : fonctions polymorphes

La définition de la fonction ne dépend pas (complètement) du type de ses paramètres . . .

Syntaxe

```
let f (p1 : 'x) (p2 : 'y) (...) : 'r = ... (* corps de f *)
```

Exemples

- identité

```
let identite (p1 : 'x) : 'x = p1
```

- coupler 2 valeurs

```
let coupler (p1 : 'x) (p2 : 'y) : 'x * 'y = (p1, p2)
```

- dernier élément d'une liste

```
let rec dernier (l : 'x list) : 'x =
  match l with
    [] → failwith "pas de dernier"
  | [e] → e
  | e1::e2::s → (dernier (e2::s))
```

Rappels du cours précédent : ordre supérieur

→ utilisation de **fonctions** comme paramètres et/ou résultat de **fonctions**

- augmente l'expressivité du langage
- programmes plus concis, plus abstraits
- schémas de programmation (ex : traitement de listes)

Rappels du cours précédent : ordre supérieur

→ utilisation de **fonctions** comme paramètres et/ou résultat de **fonctions**

- augmente l'expressivité du langage
- programmes plus concis, plus abstraits
- schémas de programmation (ex : traitement de listes)

Syntaxe

- type d'une fonction à n paramètres de type `ti` et résultat de type `r` :

  `val f : t1 → t2 → .... → tn → r`

- définir une valeur de type fonction :

```
fun (x1 : t1) (x2 : t2) ... → ... (* expression de type r *)
```

Rappels du cours précédent : ordre supérieur

→ utilisation de **fonctions** comme paramètres et/ou résultat de **fonctions**

- augmente l'expressivité du langage
- programmes plus concis, plus abstraits
- schémas de programmation (ex : traitement de listes)

Syntaxe

- type d'une fonction à n paramètres de type `ti` et résultat de type `r` :

```
val f : t1 → t2 → .... → tn → r
```

- définir une valeur de type fonction :

```
fun (x1 : t1) (x2 : t2) ... → ... (* expression de type r *)
```

Exemple : que fait cette fonction ?

```
let rec m (f : 'x → 'x) (n : int) : ('x → 'x) = match n with
  0 → (fun (e : 'x) → e)
  |_ → let g = (m f (n-1)) in fun (e : 'x) → (f (g e))
```

Rappels du cours précédent : ordre supérieur

→ utilisation de **fonctions** comme paramètres et/ou résultat de **fonctions**

- augmente l'expressivité du langage
- programmes plus concis, plus abstraits
- schémas de programmation (ex : traitement de listes)

Syntaxe

- type d'une fonction à n paramètres de type `ti` et résultat de type `r` :

```
val f : t1 → t2 → .... → tn → r
```

- définir une valeur de type fonction :

```
fun (x1 : t1) (x2 : t2) ... → ... (* expression de type r *)
```

Exemple : que fait cette fonction ?

```
let rec m (f : 'x → 'x) (n : int) : ('x → 'x) = match n with
  0 → (fun (e : 'x) → e)
  |_ → let g = (m f (n–1)) in fun (e : 'x) → (f (g e))
```

Que vaut (`m (fun x → x+1) 5`) 3 ?

Généralisation de la somme des n premiers entiers

Somme des n premiers entiers :

$$1 + 2 + \ldots + (n-1) + n = \big(1 + 2 + \ldots + (n-1)\big) + n$$

Implémenté par :

```
let rec somme_entiers (n:int) =
  if n=0 then 0 else somme_entiers (n-1) + n
```

Généralisation de la somme des n premiers entiers

Somme des n premiers entiers :

$$1 + 2 + \ldots + (n-1) + n = \big(1 + 2 + \ldots + (n-1)\big) + n$$

Implémenté par :

```
let rec somme_entiers (n:int) =
  if n=0 then 0 else somme_entiers (n-1) + n
```

La somme des n premiers carrés est similaire :

$$1^2 + 2^2 + \ldots + (n-1)^2 + n^2 = \big(1^2 + 2^2 + \ldots + (n-1)^2\big) + n^2$$

Implémenté par :

```
let rec somme_carres (n:int) =
  if n=0 then 0 else somme_carres (n-1) + (n*n)
```

Généralisation de la somme des n premiers entiers

Somme des n premiers entiers :

$$1 + 2 + \ldots + (n-1) + n = \big(1 + 2 + \ldots + (n-1)\big) + n$$

Implémenté par :

```
let rec somme_entiers (n:int) =
  if n=0 then 0 else somme_entiers (n-1) + n
```

La somme des n premiers carrés est similaire :

$$1^2 + 2^2 + \ldots + (n-1)^2 + n^2 = \big(1^2 + 2^2 + \ldots + (n-1)^2\big) + n^2$$

Implémenté par :

```
let rec somme_carres (n:int) =
  if n=0 then 0 else somme_carres (n-1) + (n*n)
```

Généralisation

- Définir une fonction `sigma` qui calcule la somme des images par une fonction f des n premiers entiers
- Donnez une implémentation de `somme_entiers` et `somme_carres` en utilisant `sigma`

Plan

Quelques rappels

Ordre supérieur sur les listes

Curryfication

Généraliser les fonctions sur les listes ?

Deux exemples

Différents schémas de calcul sur les éléments d'une liste `l`

- appliquer une fonction `f` à chaque élt de `l` sans changer sa structure :

  [`e1` ; `e2` ; ... ; `en`] → [(`f e1`) ; (`f e2`) ; ... ; (`f en`)]

  exemples :
  - incrémenter chaque élt d'une liste d'entiers
  - transformer une liste d'entiers en liste de caractères (codes Ascii)
  - mettre un texte (liste de lettres) en majuscules

  fonction `map` : (`'x` → `'y`) → `'x list` → `y' list`

Généraliser les fonctions sur les listes ?

Deux exemples

Différents schémas de calcul sur les éléments d'une liste `l`

- appliquer une fonction `f` à chaque élt de `l` sans changer sa structure :

  `[e1 ; e2 ; ... ; en]` $\rightarrow$ `[(f e1) ; (f e2) ; ... ; (f en)]`

  exemples :
  - incrémenter chaque élt d'une liste d'entiers
  - transformer une liste d'entiers en liste de caractères (codes Ascii)
  - mettre un texte (liste de lettres) en majuscules

  fonction `map` : `('x` $\rightarrow$ `'y)` $\rightarrow$ `'x list` $\rightarrow$ `y' list`

- "replier" `l` en appliquant une fonction `f` à chacun de ses élts :

  `[e1 ; e2 ; ... ; en]` $\rightarrow$ `(f e1 (f e2 ... (f en acc)))`

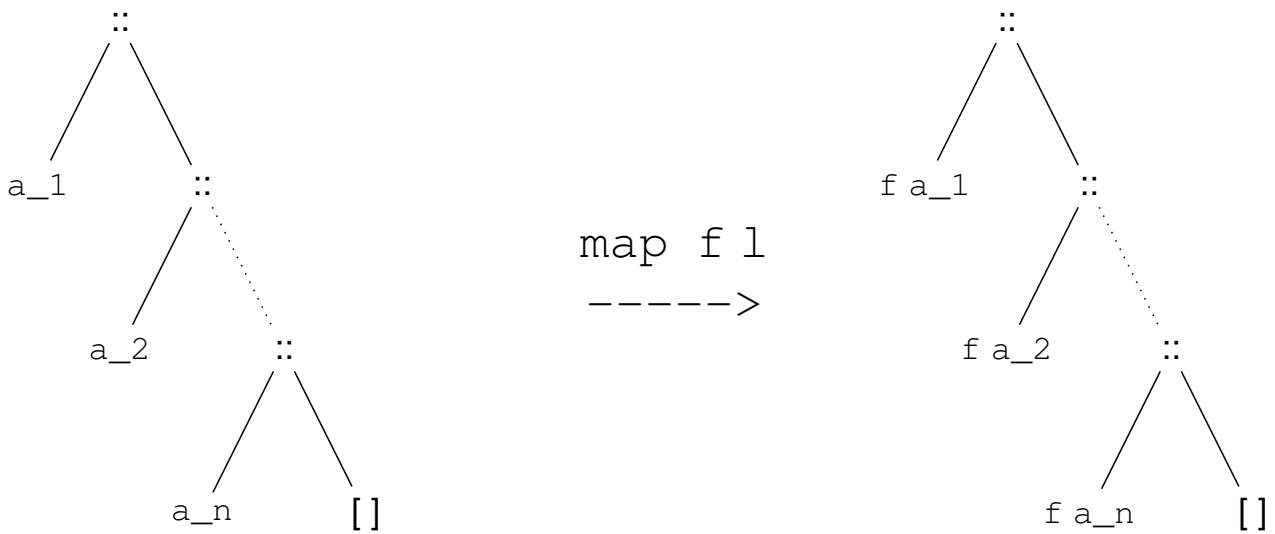
  exemples :
  - somme des éléments d'une liste
  - maximum d'une liste
  - une liste est-elle croissante ?

  fonction `fold_right` : `('x` $\rightarrow$ `'y` $\rightarrow$ `'y)` $\rightarrow$ `'y` $\rightarrow$ `'x list` $\rightarrow$ `'y`

Spécification de la fonction “map”

Etant données :

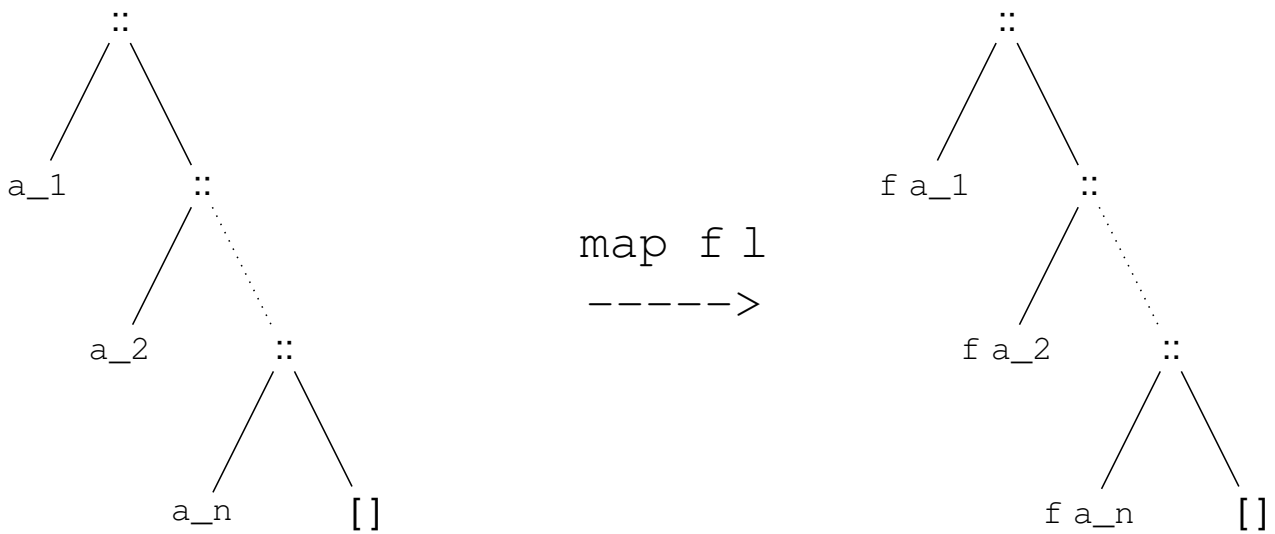
- une fonction `f` de type `'x → 'y`
- une liste [`a_1` ; `a_2` ; ... ; `an`] de type `'x list`



Spécification de la fonction “map”

Etant données :

- une fonction `f` de type `'x → 'y`
- une liste [`a_1` ; `a_2` ; ... ; `an`] de type `'x list`



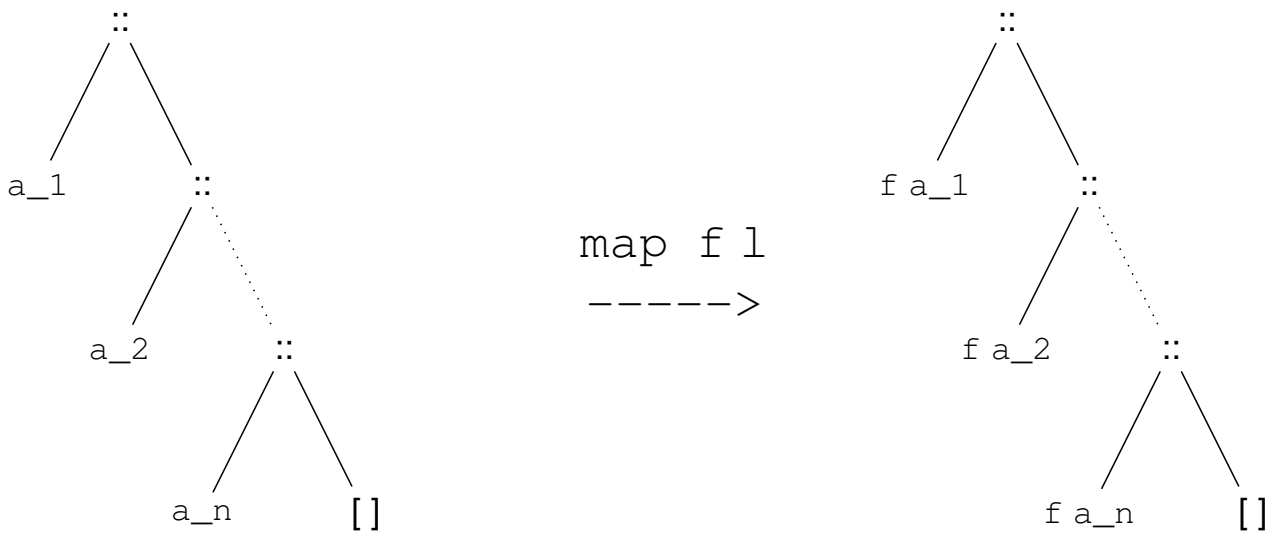
Remarque

- `map` renvoie la liste [(`f a_1`) ; (`f a_2`) ; ... ; (`f a_n`)]
- le résultat de `f a_i` ne dépend pas de la **position** de l'élément `a_i`
- `map` peut changer le type de la liste

Spécification de la fonction “map”

Etant données :

- une fonction `f` de type `'x → 'y`
- une liste [`a_1` ; `a_2` ; ... ; `an`] de type `'x list`



Remarque

- `map` renvoie la liste [(`f a_1`) ; (`f a_2`) ; ... ; (`f a_n`)]
- le résultat de `f a_i` ne dépend pas de la **position** de l'élément `a_i`
- `map` peut changer le type de la liste

□

Typage : `map ('x → 'y) → 'x list → y' list`

si `l` est de type `'x list` et `f` est de type `'x→'y`
alors (`map f l`) est de type `'y list`

Réalisation de la fonction “map”

Etant données :

- une fonction `f` de type `'x` $\rightarrow$ `'y`
- une liste `l` = [`e1` ; `e2` ; ... ; `en`] de type `'x list`

$\rightarrow$ construire la liste [(`f e1`) ; (`f e2`) ; ... ; (`f en`)] ?

Réalisation de la fonction “map”

Etant données :

- une fonction `f` de type `'x` $\rightarrow$ `'y`
- une liste `l` = [`e1` ; `e2` ; ... ; `en`] de type `'x list`

$\rightarrow$ construire la liste [(`f e1`) ; (`f e2`) ; ... ; (`f en`)] ?

Equation récursives

$$\text{map (f, [])} \quad = $$

Réalisation de la fonction “map”

Etant données :

- une fonction `f` de type `'x` $\rightarrow$ `'y`
- une liste `l` = [`e1` ; `e2` ; ... ; `en`] de type `'x list`

$\rightarrow$ construire la liste [(`f e1`) ; (`f e2`) ; ... ; (`f en`)] ?

Equation récursives

$$\begin{aligned} \text{map (f, [])} &= [] \\ \text{map (f, e::s)} &= \end{aligned}$$

Réalisation de la fonction “map”

Etant données :

- une fonction `f` de type `'x` $\rightarrow$ `'y`
- une liste `l` = [`e1` ; `e2` ; ... ; `en`] de type `'x list`

$\rightarrow$ construire la liste [(`f e1`) ; (`f e2`) ; ... ; (`f en`)] ?

Equation récursives

$$\begin{aligned} \text{map (f, [])} &= \text{[]} \\ \text{map (f, e::s)} &= \text{(f e)::(map f s)} \end{aligned}$$

Réalisation de la fonction “map”

Etant données :

- une fonction `f` de type `'x` → `'y`
- une liste `l` = [`e1` ; `e2` ; ... ; `en`] de type `'x list`

→ construire la liste [(`f e1`) ; (`f e2`) ; ... ; (`f en`)] ?

Equation récursives

$$\begin{aligned} \text{map (f, [])} &= [] \\ \text{map (f, e::s)} &= \text{(f e)::(map f s)} \end{aligned}$$

Code Caml

```
let rec map (f : 'x → 'y) (l : 'x list) : 'y list =
  match l with
      [] → []
    | e::s → (f e)::(map f s)
```

fonction prédéfinie : `List.map`

Applications de la fonction “map”

Exemple : Vectorisation

- Spécification:
  - Profil: vectorise: Seq(Elt) $\rightarrow$ Seq(Vect(Elt)), où Vec est l’ensemble des listes à un élément
  - Sémantique :
    `vectorise` [`e1`;...;`en`] = [[`e1`] ; ... ; [`en`]]

Applications de la fonction “map”

Exemple : Vectorisation

- Spécification:
  - Profil: vectorise: Seq(Elt) $\rightarrow$ Seq(Vect(Elt)), où Vec est l’ensemble des listes à un élément
  - Sémantique :
    `vectorise` [`e1`;...;`en`] = [[`e1`] ; ... ; [`en`]]
- Implémentation:

Applications de la fonction “map”

Exemple : Vectorisation

- Spécification:
  - Profil: vectorise: Seq(Elt) $\rightarrow$ Seq(Vect(Elt)), où Vec est l’ensemble des listes à un élément
  - Sémantique :
    `vectorise` [e1;...;en] = [[e1] ; ... ; [en]]
- Implémentation:

```
let vectorise (l:'x list) :'x list list =
    (List.map (fun e → [e]) l)
```

Applications de la fonction “map”

Exemple : Vectorisation

- Spécification:
  - Profil: vectorise: Seq(Elt) $\rightarrow$ Seq(Vect(Elt)), où Vec est l'ensemble des listes à un élément
  - Sémantique :
    `vectorise` [e1;...;en] = [[e1] ; ... ; [en]]
- Implémentation:

```
let vectorise (l:'x list) :'x list list =
    (List.map (fun e → [e]) l)
```

Exemple : Concatène à chaque

- Spécification:
  - Profil : Seq(Seq(Elt)) $\times$ Seq(Elt) $\rightarrow$ Seq(Seq(Elt))
  - Semantique :
    `concatene_a_chaque` ([a1; ...; an], v) = [a1@v ; ... ; an@v]

Applications de la fonction “map”

Exemple : Vectorisation

- Spécification:
  - Profil: vectorise: Seq(Elt) $\rightarrow$ Seq(Vect(Elt)), où Vec est l'ensemble des listes à un élément
  - Sémantique :
    `vectorise` [e1;...;en] = [[e1] ; ... ; [en]]
- Implémentation:

```
let vectorise (l:'x list) :'x list list =
    (List.map (fun e → [e]) l)
```

Exemple : Concatène à chaque

- Spécification:
  - Profil : Seq(Seq(Elt)) $\times$ Seq(Elt) $\rightarrow$ Seq(Seq(Elt))
  - Semantique :
    `concatene_a_chaque` ([a1; ...; an], v) = [a1@v ; ... ; an@v]
- Implémentation :

Applications de la fonction “map”

Exemple : Vectorisation

- Spécification:
  - Profil: vectorise: Seq(Elt) $\rightarrow$ Seq(Vect(Elt)), où Vec est l'ensemble des listes à un élément
  - Sémantique :
    `vectorise` [e1;...;en] = [[e1] ; ... ; [en]]
- Implémentation:

```
let vectorise (l:'x list) :'x list list =
    (List.map (fun e → [e]) l)
```

Exemple : Concatène à chaque

- Spécification:
  - Profil : Seq(Seq(Elt)) $\times$ Seq(Elt) $\rightarrow$ Seq(Seq(Elt))
  - Semantique :
    `concatene_a_chaque` ([a1; ...; an], v) = [a1@v ; ... ; an@v]
- Implémentation :

```
let concatene_a_chaque (l :'x list list)
          (v :'x list) :'x list list =
              List.map (fun (a :'x list) → a@v) l
```

Applications de la fonction “map”

Exercices

Utilisez la fonction `List.map` pour réaliser les fonctions suivantes

- `enCarre`: éléve au carré tous les éléments d’une liste d’entiers
- `enAscii`: construit la liste des codes Ascii d’une liste de caractères
- `enMajuscule`: met en majuscule toutes les lettres d’une liste de caractères

Replier les éléments d'une liste : motivations . . .

Exemple : Somme des éléments d'une liste

```
let rec some (l: int list) : int = match l with
    [] → 0
    | elt::fin → elt + (somme fin)
```

Replier les éléments d'une liste : motivations ...

Exemple : Somme des éléments d'une liste

```
let rec some (l: int list) : int = match l with
    [] → 0
    | elt::fin → elt + (somme fin)
```

Exemple : Produit des éléments d'une liste

```
let rec produit (l: int list) : int = match l with
    [] → 1
    | elt::fin → elt * (produit fin)
```

Replier les éléments d'une liste : motivations ...

Exemple : Somme des éléments d'une liste

```
let rec some (l: int list) : int = match l with
    [] → 0
    | elt::fin → elt + (somme fin)
```

Exemple : Produit des éléments d'une liste

```
let rec produit (l: int list) : int = match l with
    [] → 1
    | elt::fin → elt * (produit fin)
```

Exemple : Concatène les éléments d'une liste

```
let rec concatene (l : 'x list list) : 'x list = match l with
[] → []
| elt::fin → elt @ (concatene fin)
```

Replier les éléments d'une liste : motivations ...

Exemple : Somme des éléments d'une liste

```
let rec some (l: int list) : int = match l with
    [] → 0
    | elt::fin → elt + (somme fin)
```

Exemple : Produit des éléments d'une liste

```
let rec produit (l: int list) : int = match l with
    [] → 1
    | elt::fin → elt * (produit fin)
```

Exemple : Concatène les éléments d'une liste

```
let rec concatene (l : 'x list list) : 'x list = match l with
[] → []
| elt::fin → elt @ (concatene fin)
```

→ Les seules différences entre ces fonctions sont :

- le "cas de base" : ce que renvoie la fonction appliquée à une liste vide
- comment "combiner" l'élément de tête avec l'appel récursif sur la fin de la liste

Replier les éléments d'une liste : la fonction "fold_right"

Si on observe les calculs effectués par les fonctions précédentes :

- `somme` [a1;a2;...;an] = `+` a1 (+ a2 (... (+ an 0)...))
- `produit` [a1;a2;...;an] = `*` a1 (* a2 (... (* an 1)...))
- `concatene` [a1;a2;...;an] = `@` a1 (@ a2 (... (@ an [])...))

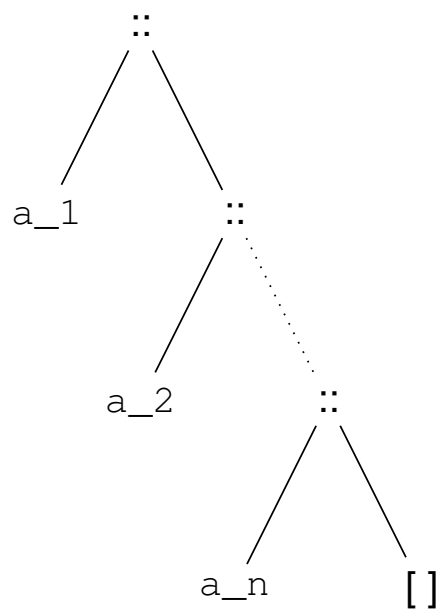
Replier les éléments d'une liste : la fonction "fold_right"

Si on observe les calculs effectués par les fonctions précédentes :

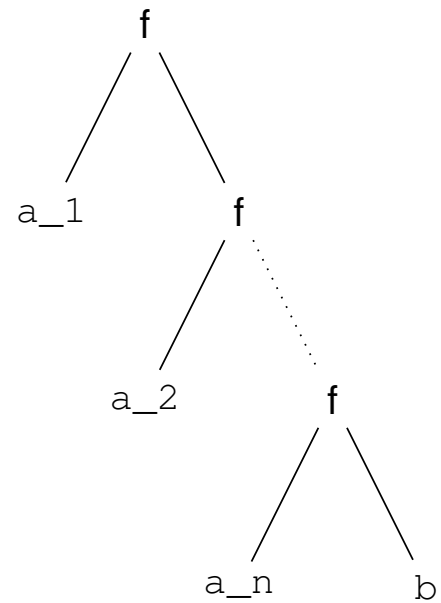
- `somme` [a1;a2;...;an] = + a1 (+ a2 (... (+ an 0)...))
- `produit` [a1;a2;...;an] = * a1 (* a2 (... (* an 1)...))
- `concatene` [a1;a2;...;an] = @ a1 (@ a2 (... (@ an [])...))

Plus généralement, étant données :

- une fonction `f` de type 'x → 'y → 'y,
- une liste `l` = [a_1 ; a_2 ; ... ; a_n] de type 'x list
- une valeur initiale `b` de type 'y



fold_right f l b
- - - - ->



→ résultat de type 'y

Réalisation de la fonction “fold_right”

Etant données :

- une fonction `f` de type `'x` $\rightarrow$ `'y` $\rightarrow$ `'y`
- une liste `l` = [`a1`;...;`an`] de type `'x list`
- une “valeur initiale” `b` de type `'y`

$\rightarrow$ calculer la valeur (`fold_right f l b`) = (`f a1` (`f a2` (... (`f an b`)))) ?

Réalisation de la fonction “fold_right”

Etant données :

- une fonction `f` de type `'x` $\rightarrow$ `'y` $\rightarrow$ `'y`
- une liste `l` = [`a1`;...;`an`] de type `'x list`
- une “valeur initiale” `b` de type `'y`

$\rightarrow$ calculer la valeur (`fold_right f l b`) = (`f a1` (`f a2` (... (`f an b`)))) ?

Equations récursives

fold_right(f, [], b) $=$

Réalisation de la fonction “fold_right”

Etant données :

- une fonction `f` de type `'x` $\rightarrow$ `'y` $\rightarrow$ `'y`
- une liste `l` = [`a1`;...;`an`] de type `'x list`
- une “valeur initiale” `b` de type `'y`

$\rightarrow$ calculer la valeur (`fold_right f l b`) = (`f a1` (`f a2` (... (`f an b`)))) ?

Equations récursives

$$\begin{aligned} \text{fold\_right(f, [], b)} &= \text{b} \\ \text{fold\_right(f, a::s, b)} &= \end{aligned}$$

Réalisation de la fonction “fold_right”

Etant données :

- une fonction `f` de type `'x` $\rightarrow$ `'y` $\rightarrow$ `'y`
- une liste `l` = [`a1`;...;`an`] de type `'x list`
- une “valeur initiale” `b` de type `'y`

$\rightarrow$ calculer la valeur (`fold_right f l b`) = (`f a1` (`f a2` (... (`f an b`)))) ?

Equations récursives

$$\begin{aligned} \text{fold\_right(f, [], b)} &= \text{b} \\ \text{fold\_right(f, a::s, b)} &= \text{f (a, (fold\_right (f, s, b))} \end{aligned}$$

Réalisation de la fonction “fold_right”

Etant données :

- une fonction `f` de type `'x → 'y → 'y`
- une liste `l` = [`a1`;...;`an`] de type `'x list`
- une “valeur initiale” `b` de type `'y`

→ calculer la valeur (`fold_right f l b`) = (`f a1 (f a2 (... (f an b))))` ?

Equations récursives

$$\begin{aligned} \text{fold\_right(f, [], b)} &= \text{b} \\ \text{fold\_right(f, a::s, b)} &= \text{f (a, (fold\_right (f, s, b))} \end{aligned}$$

Code Caml

```
let rec fold_right (f : 'x → 'y → 'y) (l : 'x list) (b : 'y) : 'y =
  match l with
    [] → b
   |a::s → (f a (fold_right f s b))
```

fonction prédéfinie : `List.fold_right`

Exercices : utilisation de `fold_right`

En utilisant la fonction `List.fold_right` :

1. Re-écrire les fonctions précédentes, `somme`, `produit` et `concatene`
   exemple :
   ```
   let somme (l : int list) : int = List.fold_right (+) l 0
   ```

2. nombre d'élémenst d'une liste

3. maximum d'une liste

4. une liste ne contient-elle que des entiers pairs ?

5. une liste est-elle croissante ?

6. etc.

Repliage “par la gauche”

la fonction `fold_left`

Etant données :

- une fonction `f` de type `'x` → `'y` → `'x`
- une “valeur initiale” `b` de type `'x`:
- une liste `l` = [`a1` ; `a2` ; ... ; `an`] de type `'y list`

→ calculer la valeur (`fold_left f b l`) = (`f` (.... (`f` (`f b a1`) `a2`)) `an`) ?

Repliage “par la gauche”

la fonction `fold_left`

Etant données :

- une fonction `f` de type `'x` $\rightarrow$ `'y` $\rightarrow$ `'x`
- une “valeur initiale” `b` de type `'x`:
- une liste `l` = [`a1` ; `a2` ; ... ; `an`] de type `'y list`

$\rightarrow$ calculer la valeur (`fold_left f b l`) = (`f` (.... (`f` (`f` `b` `a1`) `a2`)) `an`) ?

Equations récursives

fold_left(f, b, []) $=$

Repliage “par la gauche”

la fonction `fold_left`

Etant données :

- une fonction `f` de type `'x` $\rightarrow$ `'y` $\rightarrow$ `'x`
- une “valeur initiale” `b` de type `'x`:
- une liste `l` = [`a1` ; `a2` ; ... ; `an`] de type `'y list`

$\rightarrow$ calculer la valeur (`fold_left f b l`) = (`f` (.... (`f` (`f b a1`) `a2`)) `an`) ?

Equations récursives

$$\begin{aligned} \text{fold\_left(f, b, [])} &= \text{b} \\ \text{fold\_left(f, b, a::s)} &= \end{aligned}$$

Repliage “par la gauche”

la fonction `fold_left`

Etant données :

- une fonction `f` de type `'x` $\rightarrow$ `'y` $\rightarrow$ `'x`
- une “valeur initiale” `b` de type `'x`:
- une liste `l` = [`a1` ; `a2` ; ... ; `an`] de type `'y list`

$\rightarrow$ calculer la valeur (`fold_left f b l`) = (`f` (.... (`f` (`f b a1`) `a2`)) `an`) ?

Equations récursives

$$\begin{aligned} \text{fold\_left(f, b, [])} &= \text{b} \\ \text{fold\_left(f, b, a::s)} &= \text{(fold\_left (f, (f b a), s))} \end{aligned}$$

Repliage “par la gauche”

la fonction `fold_left`

Etant données :

- une fonction `f` de type `'x → 'y → 'x`
- une “valeur initiale” `b` de type `'x`:
- une liste `l` = [`a1` ; `a2` ; ... ; `an`] de type `'y list`

→ calculer la valeur (`fold_left f b l`) = (`f` (.... (`f` (`f b a1`) `a2`)) `an`) ?

Equations récursives

$$\begin{aligned} \text{fold\_left(f, b, [])} &= \text{b} \\ \text{fold\_left(f, b, a::s)} &= \text{(fold\_left (f, (f b a), s))} \end{aligned}$$

Code Caml

```
let rec fold_left (f : 'x → 'y → 'x) (b : 'x) (l : 'y list) : 'x =
 match l with
   [] → b
  |a::s → (fold_left f (f b a) s)
```

fonction prédéfinie : `List.fold_left`

Autre exemple : filtrer une liste par un prédicat

la fonction `filter`

Un prédicat est une fonction qui renvoie un booléen

Filtrage par rapport à un prédicat

Spécifier et implémenter une fonction `filter` qui extrait les éléments d'une liste `l` qui vérifient un prédicat `p`

Evaluer un prédicat sur les éléments d'une liste

- Définir une fonction `tous` qui teste si *tous* les éléments d'une liste satisfont un prédicat `p`
- Définir une fonction `existe` qui teste si *au moins un* élément d'une liste satisfait un prédicat `p`

Exercices supplémentaires

Exercice : map avec fold

- Redéfinir `map` en utilisant `fold_left`
- Redéfinir `map` en utilisant `fold_right`

Exercice : minimum et maximum en une ligne de code

Définir les fonctions `minimum` et `maximum` d'une liste en utilisant `fold_left` et/ou `fold_right`.

Plan

Quelques rappels

Ordre supérieur sur les listes

Curryfication

A propos de curryfication

Exemple : `let f (x1:int) (x2:int) (x3:int) : int = x1+x2+x3`

- `f` est de type “fonction à 3 paramètres entiers et à resultat entier”
- le résultat de (`f` 1 2 3) est l'entier 6

Résultat de (`f` 1) ? $\rightarrow$ “fonction à 2 paramètres entiers et à resultat entier”

$\Rightarrow$ L'application `f x1 x2 ... xn` est en fait une suite d'applications

$$(((f_1\ \ x1)\ \ x2)...)\ \ xn$$

Definition : application partielle

C'est l'application d'une fonction à n paramètres formels avec strictement moins de n paramètres effectifs. Le résultat d'une application partielle est donc une **fonction**.

Typage :

Si

- `f` est de type `t1` $\rightarrow$ `t2` $\rightarrow$ ... $\rightarrow$ `tn` $\rightarrow$ `t`, et
- `xi` est de type `ti` pour $i \in [1, j] \subseteq [1, n]$

Alors `f x1 x2 ... xj` est de type `t(j+1)` $\rightarrow$ ... $\rightarrow$ `tn` $\rightarrow$ `t`

A propos de curryfication

Quelques exemples

Exemple : Appliquer une fonction 2 fois
Retour sur la fonction `appliquer2fois`

```
let appliquer2fois (f:int → int) (x:int):int
        = f (f x)
```

Appliquer `appliquer2fois` avec un seul argument :

```
appliquer2fois (fun x → x +4)
```

renvoie la fonction

```
fun x → x + 8
```

DEMO: `appliquer2fois`

Intérêts de la curryfication

Définition d'une fonction qui prend $a \in A$ et $b \in B$ et renvoie $c \in C$

Sans curryfication	Avec currifycation
`f: tA * tB → tC`	`f: tA → tB → tC`
`f` a un seul paramètre (un couple) `f (a,b)` est de type `tC`	`f` a deux paramètres `f a b` est de type `tC` `f a` est de type `tB → tC`

DEMO: 2 définitions de l'addition sur les entiers et le (+) prédéfini de OCaml

A retenir

- La curryfication offre une certaine *flexibilité*
- Permet également de *spécialiser* une fonction

Remarque Lorsque l'on applique une fonction currifiée il est possible d'oublier un paramètre sans s'en rendre compte . . . □

Conclusion / Résumé

Polymorphisme

- types généraux
- type “paramètres”

Ordre supérieur

- prendre une fonction en paramètre et/ou renvoyer une fonction
- améliore la qualité du code (concision, schémas généraux)

Curryfication

- application partielle de fonctions
- spécialisation de fonctions