UNIVERSITÉ Grenoble Alpes

INF201
Algorithmique et Programmation Fonctionnelle
Cours 9 : Ordre supérieur

Année 2022

Rappel des épisodes précédents

- types prédéfinis : booléens, entiers, réels, . . .
- identificateurs (locaux et globaux)
- définition et utilisation de fonctions
- définition de types : synonyme, énuméré, produit, somme
- filtrage par pattern-matching
- récursion
  - fonctions récursives, terminaison
  - types récursifs
- listes avec constructeurs explicites (Cons/Nil) et notations OCaml ([], ::)

Plan

(Retour sur le) Polymorphisme

Ordre supérieur

Curryfication

Pourquoi le polymorphisme ? (1)

étendre la notion de fonction

Définition d'une fonction *identité* :

- Identité sur les `int` :

```
let id (x:int):int = x
```

```
val id : int → int = <fun>
```

Pourquoi le polymorphisme ? (1)

étendre la notion de fonction

Définition d'une fonction *identité* :

- Identité sur les `int` :

```
let id (x:int):int = x          val id : int → int = <fun>
```

- Identité sur les `float` :

```
let id (x:float):float = x   val id : float → float = <fun>
```

Pourquoi le polymorphisme ? (1)

étendre la notion de fonction

Définition d'une fonction *identité* :

- Identité sur les `int` :

```
let id (x:int):int = x          val id : int → int = <fun>
```

- Identité sur les `float` :

```
let id (x:float):float = x      val id : float → float = <fun>
```

- Identité sur les `char` :

```
let id (x:char):char = x        val id : char → char = <fun>
```

Pourquoi le polymorphisme ? (1)

étendre la notion de fonction

Définition d'une fonction *identité* :

- Identité sur les `int` :
  `let id (x:int):int = x` `val id : int → int = <fun>`
- Identité sur les `float` :
  `let id (x:float):float = x` `val id : float → float = <fun>`
- Identité sur les `char` :
  `let id (x:char):char = x` `val id : char → char = <fun>`

Inconvénients :

- **une fonction par type** pour lesquels la fonction est définie
- des noms différents sont nécessaires si ces fonctions doivent "cohabiter" dans un même programme . . .

Pourquoi le polymorphisme ? (2)

Fonctions sur les listes

Calcul de la longueur d'une liste

- d'entiers :

```
let rec longueur_int (l: int list):int=
 match l with
  | [ ] → 0
  | _::l → 1+ longueur_int l
```

Pourquoi le polymorphisme ? (2)

Fonctions sur les listes

Calcul de la longueur d'une liste

- d'entiers :

```
let rec longueur_int (l: int list):int=
 match l with
  | [ ] → 0
  | _::l → 1+ longueur_int l
```

- de caractères :

```
let rec longueur_char (l: char list):int=
 match l with
  | [ ] → 0
  | _::l → 1+ longueur_char l
```

Pourquoi le polymorphisme ? (2)

Fonctions sur les listes

Calcul de la longueur d'une liste

- d'entiers :

```
let rec longueur_int (l: int list):int=
 match l with
  | [ ] → 0
  | _::l → 1+ longueur_int l
```

- de caractères :

```
let rec longueur_char (l: char list):int=
 match l with
  | [ ] → 0
  | _::l → 1+ longueur_char l
```

- de pingouins . . .

```
let rec longueur_char (l: pingouins list):int=
 match l with ...
```

Pourquoi le polymorphisme ? (2)

Fonctions sur les listes

Calcul de la longueur d'une liste

- d'entiers :

```
let rec longueur_int (l: int list):int=
 match l with
  | [ ] → 0
  | _::l → 1+ longueur_int l
```

- de caractères :

```
let rec longueur_char (l: char list):int=
 match l with
  | [ ] → 0
  | _::l → 1+ longueur_char l
```

- de pingouins . . .

```
let rec longueur_char (l: pingouins list):int=
 match l with ...
```

Le corps de ces fonctions ne dépend pas du type des éléments . . .

⇒ il faut une notion de liste générique (ou polymorphe)

Pourquoi le polymorphisme ? (3)

Limitation de la notion (courante) de liste

→ Plusieurs définitions du type liste avec constructeurs explicites :

- `type list_int = Nil | Cons int * list_int`
  ex :`Cons (2, Cons (9,Nil))`
- `type list_char = Nil | Cons char * list_char`
  ex :`Cons ('t', Cons ('v',Nil))`

Pourquoi le polymorphisme ? (3)

Limitation de la notion (courante) de liste

→ Plusieurs définitions du type liste avec constructeurs explicites :

- `type list_int = Nil | Cons int * list_int`
  ex :`Cons (2, Cons (9,Nil))`
- `type list_char = Nil | Cons char * list_char`
  ex :`Cons ('t', Cons ('v',Nil))`

→ Plusieurs “types listes” en notation OCaml :

- liste d'entiers :
  [1;2] (ou 1::2::[ ]) de type `int list`
- liste de caractères :
  ['e'; 'n'] (ou 'e':: 'n'::[ ]) de type `char list`
- liste de booléens :
  [`true`; `false`] (ou `true`::`false`::[ ]) de type `bool list`

Fonctions polymorphes

Retour sur les différentes fonctions identité :

↪ et si on ne précisait pas le type du paramètre et du résultat ?

```
let id x = x
```

Fonctions polymorphes

Retour sur les différentes fonctions identité :

$\hookrightarrow$ et si on ne précisait pas le type du paramètre et du résultat ?

```
let id x = x
```

$\hookrightarrow$ le type renvoyé par OCaml est `'a` $\rightarrow$ `'a` (ou encore $\alpha \rightarrow \alpha$)

```
val id : 'a -> 'a = <fun>
```

Fonctions polymorphes

Retour sur les différentes fonctions identité :

↪ et si on ne précisait pas le type du paramètre et du résultat ?

```
let id x = x
```

↪ le type renvoyé par OCaml est `'a → 'a` (ou encore $\alpha \to \alpha$)

```
val id : 'a → 'a = <fun>
```

Inférence de type :

OCaml calcule le type "le plus général possible" pour `id`

Fonction polymorphe

`id` est une fonction polymorphe (un de ses paramètres peut prendre plusieurs types). On peut aussi le spécifier explicitement :

```
let id (x : 'a) : 'a = x
```

Fonctions polymorphes

Retour sur les différentes fonctions identité :

↪ et si on ne précisait pas le type du paramètre et du résultat ?

```
let id x = x
```

↪ le type renvoyé par OCaml est `'a → 'a` (ou encore $\alpha \to \alpha$)

```
val id : 'a → 'a = <fun>
```

Inférence de type :

OCaml calcule le type "le plus général possible" pour `id`

Fonction polymorphe

`id` est une fonction polymorphe (un de ses paramètres peut prendre plusieurs types). On peut aussi le spécifier explicitement :

```
let id (x : 'a) : 'a = x
```

Remarque

`'a` désigne un **type paramètre**

Comme pour tout paramètre, on peut lui donner n'importe quel nom :

`'b`, `'elt`, `'toto`, etc.

Types polymorphes

On peut aussi définir des types paramétrés par d'autres types . . .

Exemple :

```
type t2 = ('elt * 'elt)
type t2 = ('elt1 * 'elt2)
type 't liste = Nil | Cons of 't * 't liste
```

→ on parle de types polymorphes

Types polymorphes

On peut aussi définir des types paramétrés par d'autres types ...

Exemple :

```
type t2 = ('elt * 'elt)
type t2 = ('elt1 * 'elt2)
type 't liste = Nil | Cons of 't * 't liste
```

$\rightarrow$ on parle de types polymorphes

Le type `list` de Caml est un type polymorphe :

- le type de `[]` est `'a list` (ou encore α `list`)
- le type de `::` est `'a` $\rightarrow$ `'a list` $\rightarrow$ `'a list`
  (ou encore $\alpha \rightarrow \alpha$ `list` $\rightarrow \alpha$ `list`)

Remarque Les éléments d'une liste sont tous du même type `'a` □

Exemple de fonction que l'on ne peut pas définir sur une liste polymorphe ???

Outline

Introduction à l'ordre supérieur

fonctions = briques de base pour :

- découper un programme en éléments plus petits/lisibles
- calculer des valeurs

Introduction à l'ordre supérieur

fonctions = briques de base pour :

- découper un programme en éléments plus petits/lisibles
- calculer des valeurs

Mais . . .

- une fonction peut être un paramètre ou un résultat d'une autre fonction

Exemples :

- définir une fonction affine pour a et b : $f(x) = ax + b$
- définir la composition de deux fonctions $f \circ g$
- définir la dérivé f' d'une fonction f, etc.

Introduction à l'ordre supérieur

fonctions = briques de base pour :

- découper un programme en éléments plus petits/lisibles
- calculer des valeurs

Mais . . .

- une fonction peut être un paramètre ou un résultat d'une autre fonction

Exemples :

- définir une fonction affine pour a et b : $f(x) = ax + b$
- définir la composition de deux fonctions $f \circ g$
- définir la dérivé f' d'une fonction f, etc.

L'ordre supérieur permet aussi de définir des schémas de programmation :

- fonction qui applique une **fonction** à tous les éléments d'une liste
- fonction qui extrait d'une liste les éléments qui vérifient un **prédicat**

$\Rightarrow$ De nombreux intérêts du point de vue programmation . . .

Definition (Langage d'ordre supérieur)

C'est un langage (de programmation) dans lequel il est possible de transmettre des fonctions en paramètre **et** en résultat d'autres fonctions.

Definition (Fonction d'ordre supérieur)

C'est une fonction qui permet :

- soit de prendre une fonction en paramètre
- soit de renvoyer une fonction en résultat

Remarque Les fonctions qui ne sont pas d'ordre supérieur sont dites *de premier ordre*. □

Un peu de syntaxe et notations . . .

Type d'une fonction

- fonction à un paramètre de type t et résultat de type r
  en maths : $f : t \to r$ en OCaml : `f` : `t` $\to$ `r`

Un peu de syntaxe et notations . . .

Type d'une fonction

- fonction à un paramètre de type t et résultat de type r
  en maths : $f : t \to r$ en OCaml : `f : t -> r`
- fonction à n paramètres de type $t_1, t_2, \ldots t_n$ et résultat de type r
  - en maths : $f : t_1 \times t_2 \times \cdots \times t_n \to r$
  - en OCaml : `f : t1 -> t2 -> ... -> tn -> r`

  $\hookrightarrow$ on parle de forme curryfiée . . .

Un peu de syntaxe et notations . . .

Type d'une fonction

- fonction à un paramètre de type t et résultat de type r
  en maths : $f : t \to r$ en OCaml : `f : t → r`
- fonction à n paramètres de type $t_1, t_2, \ldots t_n$ et résultat de type r
  - en maths : $f : t_1 \times t_2 \times \cdots \times t_n \to r$
  - en OCaml : `f : t1 → t2 → ... → tn → r`

  $\hookrightarrow$ on parle de forme curryfiée . . .

Fonction à paramètres de type fonction

ex. : f a 2 paramètres de type fonction, $p1 : t1 \to r1$ et $p2 : t2 \to r2$

```
let f (p1 : t1 → r1) (p2 : t2 → r2) : r = ...
```

Le type de f est alors `f : (t1 → r1) → (t2 → r2) → r`

Un peu de syntaxe et notations . . .

Type d'une fonction

- fonction à un paramètre de type t et résultat de type r
  en maths : $f : t \to r$ en OCaml : `f : t → r`
- fonction à n paramètres de type $t_1, t_2, \ldots t_n$ et résultat de type r
  - en maths : $f : t_1 \times t_2 \times \cdots \times t_n \to r$
  - en OCaml : `f : t1 → t2 → ... → tn → r`

  $\hookrightarrow$ on parle de forme curryfiée . . .

Fonction à paramètres de type fonction

ex. : f a 2 paramètres de type fonction, $p1 : t1 \to r1$ et $p2 : t2 \to r2$

```
let f (p1 : t1 → r1) (p2 : t2 → r2) : r = ...
```

Le type de f est alors `f : (t1 → r1) → (t2 → r2) → r`

Fonction à résultat de type fonction

ex. : f a un paramètre entier et renvoie une fonction de type $t1 \times t2 \to r$

```
let f (x : int) : (t1 → t2 → r) =
  fun (a1 : t1) (a2 : t2) → ... (* expression de type r *)
```

Le type de f est alors `f : int → (t1 → t2 → r)`

Ordre supérieur : premiers exemples . . .

`trans`: transformation d'un point du plan

`trans`$(f, g, (x, y))$ renvoie $((f(x), g(y))$

- type de `trans` ?

Ordre supérieur : premiers exemples . . .

`trans`: transformation d'un point du plan

`trans`(f, g, (x, y)) renvoie $((f(x), g(y))$

- type de `trans` ?
  trans : $(\mathbb{N} \to \mathbb{N}) \times (\mathbb{N} \to \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}) \to (\mathbb{N} \times \mathbb{N})$
  `trans` : (`int` → `int`) → (`int` → `int`) → (`int * int`) → (`int * int`)
- réalisation de `trans` ?

Ordre supérieur : premiers exemples . . .

`trans`: transformation d'un point du plan

`trans`(f, g, (x, y)) renvoie $((f(x), g(y))$

- type de `trans` ?
  trans : $(\mathbb{N} \to \mathbb{N}) \times (\mathbb{N} \to \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}) \to (\mathbb{N} \times \mathbb{N})$
  `trans` : (`int` → `int`) → (`int` → `int`) → (`int * int`) → (`int * int`)
- réalisation de `trans` ?

```
let trans : (f: int → int) (g: int → int) (p: int * int) : (int *
int) =
            let (x,y) = p in ((f x), (g y))
```

Ordre supérieur : premiers exemples . . .

`trans`: transformation d'un point du plan

`trans`(f, g, (x, y)) renvoie $((f(x), g(y))$

- type de `trans` ?
  trans : $(\mathbb{N} \to \mathbb{N}) \times (\mathbb{N} \to \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}) \to (\mathbb{N} \times \mathbb{N})$
  `trans` : (`int` → `int`) → (`int` → `int`) → (`int * int`) → (`int * int`)
- réalisation de `trans` ?

```
let trans : (f: int → int) (g: int → int) (p: int * int) : (int *
int) =
            let (x,y) = p in ((f x), (g y))
```

`affine` :fonction affine sur a et b

`affine`(a,b) renvoie $f(x) = ax + b$

- type de `affine` ?

Ordre supérieur : premiers exemples . . .

`trans`: transformation d'un point du plan

`trans`(f, g, (x, y)) renvoie $((f(x), g(y))$

- type de `trans` ?
  trans : $(\mathbb{N} \to \mathbb{N}) \times (\mathbb{N} \to \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}) \to (\mathbb{N} \times \mathbb{N})$
  `trans` : `(int → int) → (int → int) → (int * int) → (int * int)`
- réalisation de `trans` ?

```
let trans : (f: int → int) (g: int → int) (p: int * int) : (int *
int) =
            let (x,y) = p in ((f x), (g y))
```

`affine` :fonction affine sur a et b

`affine`(a,b) renvoie $f(x) = ax + b$

- type de `affine` ?
  affine : $\mathbb{R} \times \mathbb{R} \to (\mathbb{R} \times \mathbb{R})$
  `affine` : `float → float → (float → float)`
- réalisation de `affine` ?

Ordre supérieur : premiers exemples . . .

`trans`: transformation d'un point du plan

`trans`(f, g, (x, y)) renvoie $((f(x), g(y))$

- type de `trans` ?
  trans : $(\mathbb{N} \to \mathbb{N}) \times (\mathbb{N} \to \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}) \to (\mathbb{N} \times \mathbb{N})$
  `trans` : `(int → int) → (int → int) → (int * int) → (int * int)`
- réalisation de `trans` ?

```
let trans : (f: int → int) (g: int → int) (p: int * int) : (int *
int) =
            let (x,y) = p in ((f x), (g y))
```

`affine` :fonction affine sur a et b

`affine`(a,b) renvoie $f(x) = ax + b$

- type de `affine` ?
  affine : $\mathbb{R} \times \mathbb{R} \to (\mathbb{R} \times \mathbb{R})$
  `affine : float → float → (float → float)`
- réalisation de `affine` ?

```
let affine (a: float) (b : float) : float → float =
  fun (x : float) → a *. x +. b
```

Quelques exemples de fonctions d'ordre supérieur

Fonctions numériques

Exemple : derivée d'une fonction f

On approche $f'(x)$ (valeur de la dérivée de f en x) par :

$$\frac{f(x+h)-f(x)}{h} \qquad \text{(avec } h \text{ petit)}$$

DEMO: Dérivée

Quelques exemples de fonctions d'ordre supérieur

Fonctions numériques

Rappels :

- Un *zero* d'une fonction f est une valeur x_0 t.q. $f(x_0) = 0$
- Théorème des *valeurs intermédiaires*:
  Soit f une fonction continue, a et b deux réels, si $f(a)$ et $f(b)$ sont de signe opposés, alors l'intervalle $[a, b]$ contient un *zero* pour f.
- $\sqrt{a}$ est *zero* de la fonction $x \mapsto x^2 - a$
- $\forall a \geq 0 : 0 \leq \sqrt{a} \leq \frac{1+a}{2}$

Quelques exemples de fonctions d'ordre supérieur

Fonctions numériques

Rappels :

- Un *zero* d'une fonction f est une valeur x_0 t.q. $f(x_0) = 0$
- Théorème des *valeurs intermédiaires*:
  Soit f une fonction continue, a et b deux réels, si $f(a)$ et $f(b)$ sont de signe opposés, alors l'intervalle $[a, b]$ contient un *zero* pour f.
- $\sqrt{a}$ est *zero* de la fonction $x \mapsto x^2 - a$
- $\forall a \geq 0 : 0 \leq \sqrt{a} \leq \frac{1+a}{2}$

Exercice: calcul du *zero* d'une fonction continue par dichotomie

- Définir une fonction `sign` qui indique si un réel est positif ou non ;
- En déduire une fonction `zero` qui renvoie le *zero* d'une fonction, à ϵ près, étant donnés deux réels vérifiant le théorème des valeurs intermédiaires ;
- En déduire une fonction qui approxime la racine carrée d'un réel.

Quelques exemples de fonctions d'ordre supérieur

Composition de fonctions

Composition de fonctions :

$$\begin{array}{rcll} f & : & C \longrightarrow D & \\ g & : & A \longrightarrow B & \\ g \circ f & : & C \longrightarrow B & \text{si } D \subseteq A \end{array}$$

Simplifions en prenant $D = A$, d'où $g \circ f : C \xrightarrow{f} A \xrightarrow{g} B$

Quelques exemples de fonctions d'ordre supérieur

Composition de fonctions

Composition de fonctions :

$$\begin{array}{rcll} f & : & C \longrightarrow D & \\ g & : & A \longrightarrow B & \\ g \circ f & : & C \longrightarrow B & \text{si } D \subseteq A \end{array}$$

Simplifions en prenant $D = A$, d'où $g \circ f : C \xrightarrow{f} A \xrightarrow{g} B$

Exercice : Définir la composition de fonctions en OCaml

- Spécifier la fonction `compose` qui compose deux fonctions (attention aux types !)
- Implémenter la fonction `compose`

En OCaml:
si `f` est une fonction de type `t1 → t2` et `g` est une fonction de type `t2 → t3` alors

- `compose g f` sera de type `t1→ t3`
- `compose` sera de type `(t2 → t3) → (t1 → t2) → (t1 → t3)`

DEMO: Implementation de `compose`

Quelques exemples de fonctions d'ordre supérieur

n-ième terme d'une suite et composition de fonctions

Soit une suite définie par :

$$\begin{aligned} u_0 &= a \\ u_n &= f(u_{n-1}), n \geq 1 \end{aligned}$$

Le n-ième terme u_n est $f(u_{n-1}) = f(f(u_{n-2})) = f(f(f(\ldots(u_0)\ldots)))$

Quelques exemples de fonctions d'ordre supérieur

n-ième terme d'une suite et composition de fonctions

Soit une suite définie par :

$$\begin{aligned} u_0 &= a \\ u_n &= f(u_{n-1}), n \geq 1 \end{aligned}$$

Le n-ième terme u_n est $f(u_{n-1}) = f(f(u_{n-2})) = f(f(f(\ldots(u_0)\ldots)))$

Exercice : *n*-ième terme d'une suite

Définir une fonction `nieme` qui calcule le n-ième terme de la suite définie ci-dessus pour une fonction f et un entier n

Quelques exemples de fonctions d'ordre supérieur

n-ième terme d'une suite et composition de fonctions

Soit une suite définie par :

$$\begin{aligned} u_0 &= a \\ u_n &= f(u_{n-1}), n \geq 1 \end{aligned}$$

Le n-ième terme u_n est $f(u_{n-1}) = f(f(u_{n-2})) = f(f(f(\ldots(u_0)\ldots)))$

Exercice : *n*-ième terme d'une suite

Définir une fonction `nieme` qui calcule le n-ième terme de la suite définie ci-dessus pour une fonction f et un entier n

Exercice: *n*-ième itération d'une fonction

Définir une fonction `iterate` qui calcule la **fonction** qui est la n-ième composition d'une fonction pour un n donné.

Plan

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 `f1 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 . . . □

Fonctions d'ordre supérieur

Avantages, et ce que l'on doit retenir

L'utilisation de fonctions d'ordre supérieur va permettre :

- l'écriture de programmes plus concis
- l'écriture de programmes plus faciles à étendre/maintenir
- l'émergence de schémas de programmation

Ce qu'il faudra retenir :

- la notion d'ordre supérieur
- le vocabulaire associé
- quand et comment utiliser cette notion . . .

$\rightarrow$ A développer, en pratiquant sur des exemples. . .

la suite au prochain cours !