Paradigmes de programmation

Cours 4 : Programmation fonctionnelle

Benoît Montagu — benoit.montagu@inria.fr

Préparation à l'agrégation d'informatique — Automne 2022

Qu'est-ce qu'un langage fonctionnel ?

Plusieurs critères peuvent servir à qualifier un langage de « fonctionnel » :

- Un langage où l'usage de structures immuables est fortement encouragé
- Un langage d'expressions (on construit des données)
  par opposition à un langage de commandes (programmation impérative)
- En particulier : les fonctions récursives sont préférées aux boucles
- Les effets de bord sont utilisés avec parcimonie
- Un langage où les fonctions sont des valeurs de 1re classe :
  - Fonctions comme arguments d'autres fonctions
  - Fonctions comme résultats d'autres fonctions

Une grande diversité de langages fonctionnels

Certains sont purement fonctionnels...

- Haskell, Agda, Coq, Elm...

... d'autres supportent des traits impératifs

- OCaml, Lisp, Scheme, Python...

Certains sont dynamiquement typés...

- Lisp, Scheme, Racket, Python...

... d'autres sont statiquement typés

- OCaml, Haskell, Coq, Agda...

Certains sont à portée dynamique...

- Lisp, Python...

... d'autres sont à portée statique

- Racket, OCaml, Haskell, Agda, Coq...

Dans ce cours

Des éléments fondamentaux de programmation fonctionnelle, illustrés en OCaml :

- Ordre supérieur
- Polymorphisme paramétrique
- Types de données paramétrés
- Récursion ouverte et points fixes

Fonctions de 1re classe, ordre supérieur

Ordre supérieur

- On dit que les fonctions sont de 1re classe dans un langage lorsqu'elles sont traitées comme n'importe quelle autre valeur
  En particulier, lorsque :
  - On peut calculer des fonctions comme résultats
  - On peut prendre des fonctions comme arguments
  - On peut enregistrer des fonctions dans des structures de données

Ordre supérieur

- On dit que les fonctions sont de 1re classe dans un langage lorsqu'elles sont traitées comme n'importe quelle autre valeur
  En particulier, lorsque :
  - On peut calculer des fonctions comme résultats
  - On peut prendre des fonctions comme arguments
  - On peut enregistrer des fonctions dans des structures de données
- En pratique, on confond souvent les notions de « fonctions de 1re classe » et d'« ordre supérieur »
- La notion d'« ordre supérieur » est basée sur la définition d'« ordre »

Ordre d'un type

Définition (Types simples)

$$\tau \ ::= \ \text{int} \ | \ \text{unit} \ | \ \tau \times \tau \ | \ \tau + \tau \ | \ \tau \to \tau$$

$$\tau_1 \times \tau_2 \quad \approx \quad \texttt{type prod = } \tau_1 \texttt{ * } \tau_2$$

$$\tau_1 + \tau_2 \quad \approx \quad \texttt{type sum = Left of } \tau_1 \texttt{ | Right of } \tau_2$$

Ordre d'un type

Définition (Types simples)

$$\tau \;::=\; \text{int} \;\mid\; \text{unit} \;\mid\; \tau \times \tau \;\mid\; \tau + \tau \;\mid\; \tau \to \tau$$

$\tau_1 \times \tau_2 \quad \approx$ `type prod = `τ_1` * `τ_2

$\tau_1 + \tau_2 \quad \approx$ `type sum = Left of `τ_1` | Right of `τ_2

Définition (Ordre d'un type)

$$\begin{aligned}
\text{ordre}(\text{int}) &= 0 \\
\text{ordre}(\text{unit}) &= 0 \\
\text{ordre}(\tau_1 \times \tau_2) &= \max(\text{ordre}\ \tau_1, \text{ordre}\ \tau_2) \\
\text{ordre}(\tau_1 + \tau_2) &= \max(\text{ordre}\ \tau_1, \text{ordre}\ \tau_2) \\
\text{ordre}(\tau_1 \to \tau_2) &= \max(1 + \text{ordre}\ \tau_1, \text{ordre}\ \tau_2)
\end{aligned}$$

Exemple

- $\text{ordre}(\text{int} \to \text{int} \to \text{int}) = \text{ordre}\,((\text{int} \times \text{int}) \to \text{int})) = 1$
- $\text{ordre}\,((\text{int} \to \text{int}) \to \text{int}) = 2$

Ordre d'un programme, ordre supérieur

- L'ordre d'un programme est l'ordre de son type
  Un programme est d'ordre supérieur lorsque son ordre est au moins 2

```
let apply f x = f x                (* `apply' est d'ordre 2 *)
(* val apply: ('a -> 'b) -> 'a -> 'b *)

let compose f g x = f (g x)        (* `compose' est d'ordre 2 *)
(* val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b *)
```

- Un langage est d'ordre supérieur lorsque l'ordre des programmes de ce langage est non borné
- Plus tard dans ce cours : plusieurs programmes non triviaux d'ordre 3

Fonctions curryfiées et décurryfiées (1/2)

Nom choisi en référence à Haskell Brooks Curry (USA, 1900-1982)

Auteur : Gleb.svechnikov

Fonctions curryfiées et décurryfiées (1/2)


Auteur : Gleb.svechnikov

Nom choisi en référence à Haskell Brooks Curry (USA, 1900-1982)

Fonctions **dé**curryfiées :

- Prennent un tuple d'arguments en paramètre
- Ont pour type $(\tau_1 \times \cdots \times \tau_n) \to \tau$ où le type de sortie τ ne contient pas de $\to$
- Exemple :

```
let plus (x, y) = x + y
(* val plus : int * int -> int *)
```

- C'est la forme des fonctions dans beaucoup de langages de programmation

Fonctions curryfiées et décurryfiées (1/2)


Auteur : Gleb.svechnikov

Nom choisi en référence à Haskell Brooks Curry (USA, 1900-1982)

Fonctions **dé**curryfiées :

- Prennent un tuple d'arguments en paramètre
- Ont pour type $(\tau_1 \times \cdots \times \tau_n) \to \tau$ où le type de sortie τ ne contient pas de $\to$
- Exemple :

```
let plus (x, y) = x + y
(* val plus : int * int -> int *)
```

- C'est la forme des fonctions dans beaucoup de langages de programmation

Fonctions curryfiées :

- Prennent un seul argument, et renvoient une fonction en résultat
- Ont pour type $\tau_1 \to \cdots \to \tau_n \to \tau$

```
let plus x y = x + y
(* val plus : int -> int -> int *)
```

- C'est généralement la forme des fonctions dans les langages fonctionnels

Fonctions curryfiées et décurryfiées (2/2)

- Curryfier : transformer une fonction non-curryfiée en une fonction curryfiée
- On peut donner une interface curryfiée à une fonction non-curryfiée avec la fonction suivante :

```
let curry f x y = f (x, y)
(* val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c *)
```

- Décurryfier : transformer une fonction curryfiée en une fonction décurryfiée
- On peut donner une interface non-curryfiée à une fonction curryfiée avec la fonction suivante :

```
let uncurry f (x, y) = f x y
(* val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c *)
```

Fonctions curryfiées et décurryfiées (2/2)

- Curryfier : transformer une fonction non-curryfiée en une fonction curryfiée
- On peut donner une interface curryfiée à une fonction non-curryfiée avec la fonction suivante :

```
let curry f x y = f (x, y)
(* val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c *)
```

- Décurryfier : transformer une fonction curryfiée en une fonction décurryfiée
- On peut donner une interface non-curryfiée à une fonction curryfiée avec la fonction suivante :

```
let uncurry f (x, y) = f x y
(* val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c *)
```

- ⚠ Ne pas confondre avec la décurryfication :
  - C'est une transformation de programmes (optimisation)
  - Utilisée dans les compilateurs pour langages fonctionnels

Fonctions curryfiées et décurryfiées (2/2)

- Curryfier : transformer une fonction non-curryfiée en une fonction curryfiée
- On peut donner une interface curryfiée à une fonction non-curryfiée avec la fonction suivante :

```
let curry f x y = f (x, y)
(* val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c *)
```

- Décurryfier : transformer une fonction curryfiée en une fonction décurryfiée
- On peut donner une interface non-curryfiée à une fonction curryfiée avec la fonction suivante :

```
let uncurry f (x, y) = f x y
(* val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c *)
```

⚠ Ne pas confondre avec la décurryfication :

- C'est une transformation de programmes (optimisation)
- Utilisée dans les compilateurs pour langages fonctionnels

❓ Quelle relation existe entre `curry` et `uncurry` ?

Fonctions curryfiées et décurryfiées (2/2)

- Curryfier : transformer une fonction non-curryfiée en une fonction curryfiée
- On peut donner une interface curryfiée à une fonction non-curryfiée avec la fonction suivante :

```
let curry f x y = f (x, y)
(* val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c *)
```

- Décurryfier : transformer une fonction curryfiée en une fonction décurryfiée
- On peut donner une interface non-curryfiée à une fonction curryfiée avec la fonction suivante :

```
let uncurry f (x, y) = f x y
(* val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c *)
```

- ⚠ Ne pas confondre avec la décurryfication :
  - C'est une transformation de programmes (optimisation)
  - Utilisée dans les compilateurs pour langages fonctionnels
- ❓ Quelle relation existe entre `curry` et `uncurry` ?
- ❓ Quelle relation existe entre l'*ordre* d'une fonction curryfiée et l'ordre de sa version non-curryfiée ?

Applications partielles, clôtures, portée des variables

- Application partielle : appel d'une fonction avec moins d'arguments que ce qui est requis par la fonction
- ```
  let plus x y = x + y
  let plus7 = plus 7
  (* plus7 ≡ fun y -> 7 + y *)
  let n = plus7 35
  (* n = 42 *)
  ```
- ```
  List.map ((+) 1) [0;1;2]
  ```

Applications partielles, clôtures, portée des variables

- Application partielle : appel d'une fonction avec moins d'arguments que ce qui est requis par la fonction
- ```
  let plus x y = x + y
  let plus7 = plus 7
  (* plus7 ≡ fun y -> 7 + y *)
  let n = plus7 35
  (* n = 42 *)
  ```
- `List.map ((+) 1) [0;1;2]`
- L'évaluation d'une application partielle produit une clôture :

  clôture = morceau de code + environnement

  L'environnement définit les arguments déjà fournis à la fonction

  Plus généralement : l'environnement définit les valeurs des variables libres
  ```

Applications partielles, clôtures, portée des variables

- Application partielle : appel d'une fonction avec moins d'arguments que ce qui est requis par la fonction
- ```
  let plus x y = x + y
  let plus7 = plus 7
  (* plus7 ≡ fun y -> 7 + y *)
  let n = plus7 35
  (* n = 42 *)
  ```
- `List.map ((+) 1) [0;1;2]`
- L'évaluation d'une application partielle produit une clôture :

  clôture = morceau de code + environnement

  L'environnement définit les arguments déjà fournis à la fonction

  Plus généralement : l'environnement définit les valeurs des variables libres
- Rappel : portée statique des variables

  Les valeurs des variables libres sont définies lorsque la fonction est *définie*

  Plus précisément : au moment de la création de la clôture
  ```

Applications partielles, clôtures, portée des variables

- Application partielle : appel d'une fonction avec moins d'arguments que ce qui est requis par la fonction
- ```
  let plus x y = x + y
  let plus7 = plus 7
  (* plus7 ≡ fun y -> 7 + y *)
  let n = plus7 35
  (* n = 42 *)
  ```
- `List.map ((+) 1) [0;1;2]`
- L'évaluation d'une application partielle produit une clôture :

  clôture = morceau de code + environnement

  L'environnement définit les arguments déjà fournis à la fonction

  Plus généralement : l'environnement définit les valeurs des variables libres
- Rappel : portée statique des variables

  Les valeurs des variables libres sont définies lorsque la fonction est *définie*

  Plus précisément : au moment de la création de la clôture
- Pour créer une clôture, il faut allouer de la mémoire (dans le tas)
  ```

Application complète d'une fonction curryfiée

- Application complète d'une fonction à 3 arguments :

```
let f q b r = q * b + r
let a = f 3 5 1
(* a = 16 *)
```

Application complète d'une fonction curryfiée

- Application complète d'une fonction à 3 arguments :

```
let f q b r = q * b + r
let a = f 3 5 1
(* a = 16 *)
```

- Une exécution naïve se ferait comme suit :

```
let f q b r = q * b + r
let f_3 = f 3

let f_3_5 = f_3 5

let a = f_3_5 1
(* a = 16 *)
```

Application complète d'une fonction curryfiée

- Application complète d'une fonction à 3 arguments :

```
let f q b r = q * b + r
let a = f 3 5 1
(* a = 16 *)
```

- Une exécution naïve se ferait comme suit :

```
let f q b r = q * b + r
let f_3 = f 3
(* f_3 est la clôture (fun b r -> q * b + r, [q ↦ 3]) *)
let f_3_5 = f_3 5

let a = f_3_5 1
(* a = 16 *)
```

Application complète d'une fonction curryfiée

- Application complète d'une fonction à 3 arguments :

```
let f q b r = q * b + r
let a = f 3 5 1
(* a = 16 *)
```

- Une exécution naïve se ferait comme suit :

```
let f q b r = q * b + r
let f_3 = f 3
(* f_3 est la clôture (fun b r -> q * b + r, [q ↦ 3]) *)
let f_3_5 = f_3 5
(* f_3_5 est la clôture (fun r -> q * b + r, [q ↦ 3, b ↦ 5]) *)
let a = f_3_5 1
(* a = 16 *)
```

Application complète d'une fonction curryfiée

- Application complète d'une fonction à 3 arguments :

```
let f q b r = q * b + r
let a = f 3 5 1
(* a = 16 *)
```

- Une exécution naïve se ferait comme suit :

```
let f q b r = q * b + r
let f_3 = f 3
(* f_3 est la clôture (fun b r -> q * b + r, [q ↦ 3]) *)
let f_3_5 = f_3 5
(* f_3_5 est la clôture (fun r -> q * b + r, [q ↦ 3, b ↦ 5]) *)
let a = f_3_5 1
(* a = 16, par exécution du code `q * b + r'
   dans l'environnement [q ↦ 3, b ↦ 5, r ↦ 1] *)
```

Application complète d'une fonction curryfiée

- Application complète d'une fonction à 3 arguments :

```
let f q b r = q * b + r
let a = f 3 5 1
(* a = 16 *)
```

- Une exécution naïve se ferait comme suit :

```
let f q b r = q * b + r
let f_3 = f 3
(* f_3 est la clôture (fun b r -> q * b + r, [q ↦ 3]) *)
let f_3_5 = f_3 5
(* f_3_5 est la clôture (fun r -> q * b + r, [q ↦ 3, b ↦ 5]) *)
let a = f_3_5 1
(* a = 16, par exécution du code `q * b + r'
   dans l'environnement [q ↦ 3, b ↦ 5, r ↦ 1] *)
```

Une clôture est allouée à chaque application intermédiaire : c'est coûteux !

Application complète d'une fonction curryfiée

- Application complète d'une fonction à 3 arguments :

```
let f q b r = q * b + r
let a = f 3 5 1
(* a = 16 *)
```

- Une exécution naïve se ferait comme suit :

```
let f q b r = q * b + r
let f_3 = f 3
(* f_3 est la clôture (fun b r -> q * b + r, [q ↦ 3]) *)
let f_3_5 = f_3 5
(* f_3_5 est la clôture (fun r -> q * b + r, [q ↦ 3, b ↦ 5]) *)
let a = f_3_5 1
(* a = 16, par exécution du code `q * b + r'
   dans l'environnement [q ↦ 3, b ↦ 5, r ↦ 1] *)
```

  Une clôture est allouée à chaque application intermédiaire : c'est coûteux !

- Le compilateur OCaml optimise ces appels pour éviter d'allouer ces clôtures intermédiaires lorsque c'est possible

Applications partielles et effets de bord

- Un effet de bord peut avoir lieu « entre » le passage de deux arguments, ou même « avant » le passage des arguments

Applications partielles et effets de bord

- Un effet de bord peut avoir lieu « entre » le passage de deux arguments, ou même « avant » le passage des arguments
-
```
(* Qu'imprime ce programme? *)
let f =
  print_endline "A";
  fun x ->
    print_endline "B";
    let y = 2 * x in
    fun z ->
      print_endline (string_of_int (z + y))
let g = f
let h1 = g 1
let v = h1 2
let v' = h1 4
let h2 = f 2
let w = h2 6
```

Applications partielles et effets de bord

- Un effet de bord peut avoir lieu « entre » le passage de deux arguments, ou même « avant » le passage des arguments
-
```
(* Qu'imprime ce programme? *)
let f =
  print_endline "A";
  fun x ->
    print_endline "B";
    let y = 2 * x in
    fun z ->
      print_endline (string_of_int (z + y))
let g = f
let h1 = g 1
let v = h1 2
let v' = h1 4
let h2 = f 2
let w = h2 6
```
- Un autre exemple est la fonction de mémoïsation `memo` du cours nº 2

Ordre supérieur et effets de bord

⚠ Utiliser dans un même programme ordre supérieur et effets de bord peut produire des comportements subtils

```
let awkward : (unit -> unit) -> unit =
  let r = ref 0 in
  fun f ->
    assert (!r mod 2 = 0);
    incr r;
    f ();
    incr r
```

Ordre supérieur et effets de bord

⚠ Utiliser dans un même programme ordre supérieur et effets de bord peut produire des comportements subtils

```
let awkward : (unit -> unit) -> unit =
  let r = ref 0 in
  fun f ->
    assert (!r mod 2 = 0);
    incr r;
    f ();
    incr r
```

▶ On s'attend au résultat suivant : quelle que soit la fonction passée à la fonction `awkward`, l'assertion `!r mod 2 = 0` sera toujours satisfaite

Ordre supérieur et effets de bord

⚠ Utiliser dans un même programme ordre supérieur et effets de bord peut produire des comportements subtils

```
let awkward : (unit -> unit) -> unit =
  let r = ref 0 in
  fun f ->
    assert (!r mod 2 = 0);
    incr r;
    f ();
    incr r
```

- On s'attend au résultat suivant : quelle que soit la fonction passée à la fonction `awkward`, l'assertion `!r mod 2 = 0` sera toujours satisfaite
- Que se passe-t-il avec le programme suivant ?
  `awkward (fun () -> awkward (fun () -> ()))`

Ordre supérieur et effets de bord

⚠ Utiliser dans un même programme ordre supérieur et effets de bord peut produire des comportements subtils

```
let awkward : (unit -> unit) -> unit =
  let r = ref 0 in
  fun f ->
    assert (!r mod 2 = 0);
    incr r;
    f ();
    incr r
```

- On s'attend au résultat suivant : quelle que soit la fonction passée à la fonction `awkward`, l'assertion `!r mod 2 = 0` sera toujours satisfaite
- Que se passe-t-il avec le programme suivant ?
  `awkward (fun () -> awkward (fun () -> ()))`
- Exemple issu de l'article :
  Andrew Pitts et Ian Stark (1998). « Operational Reasoning for Runctions with Local State ». In : Higher Order Operational Rechniques in Semantics 12, p. 227

Des fonctions dans des données : exemples

- Liste de fonctions :

```
let rec compose_list l x = match l with
| [] -> x
| f :: fs -> f (compose_list fs x)
(* val compose_list : ('a -> 'a) list -> 'a -> 'a *)
let n =
  let flip f x y = f y x in
  compose_list [flip (/) 2; (-) 0; (+) 1] (-86)
(* n = 42 *)
```

Des fonctions dans des données : exemples

- Liste de fonctions :

```
let rec compose_list l x = match l with
| [] -> x
| f :: fs -> f (compose_list fs x)
(* val compose_list : ('a -> 'a) list -> 'a -> 'a *)
let n =
  let flip f x y = f y x in
  compose_list [flip (/) 2; (-) 0; (+) 1] (-86)
(* n = 42 *)
```

- Exercice : définir l'équivalent de `compose_list` en Python. Que se passe-t-il si les fonctions dans la liste n'ont pas les mêmes types d'entrée et de sortie ?
- Faire le même exercice en OCaml. On définira un nouveau type de listes de fonctions. On doit pouvoir écrire, par exemple :

```
compose_list (cons (fun n -> n mod 2 = 0) (cons ((+) 1) nil)) 42
```

Des fonctions dans des données : exemples

- Liste de fonctions :

```
let rec compose_list l x = match l with
| [] -> x
| f :: fs -> f (compose_list fs x)
(* val compose_list : ('a -> 'a) list -> 'a -> 'a *)
let n =
  let flip f x y = f y x in
  compose_list [flip (/) 2; (-) 0; (+) 1] (-86)
(* n = 42 *)
```

- Objet comme enregistrement de fonctions :

```
type counter = { get: unit -> int; incr: unit -> unit }
let o =
  let n = ref 0 in
  { get = (fun () -> !n); incr = (fun () -> incr n) }
(* val o : counter *)
let m = o.incr (); o.incr (); o.get ()
(* m = 2 *)
```

Polymorphisme paramétrique

Type polymorphe, valeur polymorphe

- Un type polymorphe est un type très général, qui peut être instancié en des types moins généraux
- Un type polymorphe exhibe des quantifications universelles
- Une valeur est polymorphe si elle a un type polymorphe
- ⚠ Anglais : « polymorphic » / Français : « polymorphe » ~~polymorphique~~

Type polymorphe, valeur polymorphe

- Un type polymorphe est un type très général, qui peut être instancié en des types moins généraux
- Un type polymorphe exhibe des quantifications universelles
- Une valeur est polymorphe si elle a un type polymorphe
- ⚠ Anglais : « polymorphic » / Français : « polymorphe » ~~polymorphique~~
- Exemple : `let id x = x`
  Le type de la fonction `id` est polymorphe : $\forall \alpha. \alpha \to \alpha$
  Ce qui s'écrit en OCaml : `val id : 'a -> 'a`

Type polymorphe, valeur polymorphe

- Un type polymorphe est un type très général, qui peut être instancié en des types moins généraux
- Un type polymorphe exhibe des quantifications universelles
- Une valeur est polymorphe si elle a un type polymorphe
- ⚠ Anglais : « polymorphic » / Français : « polymorphe » ~~polymorphique~~
- Exemple : `let id x = x`
  Le type de la fonction `id` est polymorphe : $\forall \alpha. \alpha \to \alpha$
  Ce qui s'écrit en OCaml : `val id : 'a -> 'a`
- Par exemple, on peut instancier ce type de la manière suivante :
  - Instance sur le type int : int → int
  - Instance sur le type int → bool : (int → bool) → (int → bool)
  - Instance sur le type de l'identité, puis regénéralisé : $\forall \beta. (\beta \to \beta) \to (\beta \to \beta)$

Type polymorphe, valeur polymorphe

- Un type polymorphe est un type très général, qui peut être instancié en des types moins généraux
- Un type polymorphe exhibe des quantifications universelles
- Une valeur est polymorphe si elle a un type polymorphe
- ⚠ Anglais : « polymorphic » / Français : « polymorphe » ~~polymorphique~~
- Exemple : `let id x = x`
  Le type de la fonction `id` est polymorphe : $\forall \alpha. \alpha \to \alpha$
  Ce qui s'écrit en OCaml : `val id : 'a -> 'a`
- Par exemple, on peut instancier ce type de la manière suivante :
  - Instance sur le type int : int → int
  - Instance sur le type int → bool : (int → bool) → (int → bool)
  - Instance sur le type de l'identité, puis regénéralisé : $\forall \beta. (\beta \to \beta) \to (\beta \to \beta)$
- En OCaml, le polymorphisme est prénexe : les quantifications universelles sont situées *en tête* uniquement
  Le type $\forall \beta. (\forall \gamma. \gamma \to \gamma) \to (\beta \to \beta)$ n'est pas un type polymorphe prénexe

Polymorphisme paramétrique

- Le polymorphisme en OCaml est paramétrique
- Une fonction polymorphe n'est pas capable d'observer le type qui sera choisi pour l'instancier
- Une fonction polymorphe se comporte de la même manière quelle que soit l'instance
- Polymorphisme non paramétrique = polymorphisme ad hoc

1.

Polymorphisme paramétrique

- Le polymorphisme en OCaml est paramétrique
- Une fonction polymorphe n'est pas capable d'observer le type qui sera choisi pour l'instancier
- Une fonction polymorphe se comporte de la même manière quelle que soit l'instance
- Polymorphisme non paramétrique = polymorphisme ad hoc
- Exemple : une fonction

  ```
  val to_string: 'a -> string
  ```

  ne peut pas faire d'analyse de cas sur le *type* de son argument `'a` pour le transformer en chaîne de caractère

1.

Polymorphisme paramétrique

- Le polymorphisme en OCaml est paramétrique
- Une fonction polymorphe n'est pas capable d'observer le type qui sera choisi pour l'instancier
- Une fonction polymorphe se comporte de la même manière quelle que soit l'instance
- Polymorphisme non paramétrique = polymorphisme ad hoc
- Exemple : une fonction
  `val to_string: 'a -> string`
  ne peut pas faire d'analyse de cas sur le *type* de son argument `'a` pour le transformer en chaîne de caractère
- On peut montrer qu'une fonction qui a le type $\forall \alpha. \alpha \to \text{string}$ renvoie toujours le même résultat [1] : c'est nécessairement une fonction constante (théorèmes de paramétricité)

1. Si elle termine, est déterministe, est pure, n'utilise pas d'égalité polymorphe...

Polymorphisme et programmes qui ne retournent jamais

```
let rec loop () = loop ()
(* val loop : unit -> 'a *)

let () = loop (); print_endline "Code inaccessible"
(*       ^^^^^^^
   Warning 21 [nonreturning-statement]:
   this statement never returns (or has an unsound type.) *)
```

- `loop` a pour type $\forall \alpha.\ \text{unit} \to \alpha$
- La variable généralisée α n'est utilisée dans aucun argument
- Le résultat de `loop` aurait pour type $\forall \alpha.\ \alpha$, c-à-d tous les types possibles

Polymorphisme et programmes qui ne retournent jamais

```
let rec loop () = loop ()
(* val loop : unit -> 'a *)

let () = loop (); print_endline "Code inaccessible"
(*       ^^^^^^^
   Warning 21 [nonreturning-statement]:
   this statement never returns (or has an unsound type.) *)
```

- `loop` a pour type $\forall \alpha.\ \text{unit} \to \alpha$
- La variable généralisée α n'est utilisée dans aucun argument
- Le résultat de `loop` aurait pour type $\forall \alpha.\ \alpha$, c-à-d tous les types possibles
- Cela peut indiquer plusieurs choses :
  - Votre programme diverge
  - Votre programme renvoie une exception : `raise : exn -> 'a`
  - Votre programme décide de s'auto-détruire : `exit : int -> 'a`
  - Vous avez utilisé une primitive non sûre (exemple : `Obj.magic`)

Polymorphisme implicite / polymorphisme explicite

- En OCaml, le polymorphisme est implicite
  C'est le typeur qui a pour charge de :
  - Généraliser les types (c-à-d introduire des quantificateurs universels)
  - Instancier les types (c-à-d éliminer les quantificateurs universels)
- Dans d'autres langages (Coq, Agda, Java...), le polymorphisme est explicite
  C'est le programmeur qui a pour charge de généraliser et instancier

Polymorphisme implicite / polymorphisme explicite

- En OCaml, le polymorphisme est implicite
  C'est le typeur qui a pour charge de :
  - Généraliser les types (c-à-d introduire des quantificateurs universels)
  - Instancier les types (c-à-d éliminer les quantificateurs universels)
- Dans d'autres langages (Coq, Agda, Java...), le polymorphisme est explicite
  C'est le programmeur qui a pour charge de généraliser et instancier
- Exemple utilisant les « generics » de Java :

```
1     // `<A>' is a type parameter of the function `nth'
2     public static <A> A nth(List<A> l, int n) {
3         return l.get(n); // gets the `n'-th element of list `l'
4     }                    // or raises `IndexOutOfBoundsException'
5
6     public static void main(String[] args) {
7         List<Boolean> l = new LinkedList<Boolean>();
8         l.add(true); l.add(false); Boolean b = nth(l, 0);
9         System.out.println(b);
10    }
```

Compilation des fonctions polymorphes en OCaml

- Le comportement des fonctions polymorphes est le même quelle que soit l'instance choisie
- 👉 Le compilateur OCaml produit un seul et même code pour chaque fonction
- En OCaml, le code d'une fonction n'est pas spécialisé sur ses différentes instances
- Il n'y a pas de duplication de code lors de l'instance d'une fonction polymorphe sur un type particulier

Compilation des fonctions polymorphes en OCaml

- Le comportement des fonctions polymorphes est le même quelle que soit l'instance choisie
- 👉 Le compilateur OCaml produit un seul et même code pour chaque fonction
- En OCaml, le code d'une fonction n'est pas spécialisé sur ses différentes instances
- Il n'y a pas de duplication de code lors de l'instance d'une fonction polymorphe sur un type particulier
- ℹ️ Point culturel : monomorphisation
  - Transformation de programme qui spécialise les fonctions polymorphes (et les types paramétrés) sur leurs instances
  - Chaque instance peut ensuite être optimisée
  - Les gains en performance peuvent être importants
  - La monomorphisation duplique du code
  - Et est une optimisation globale (c-à-d pour un programme entier)
  - Le temps de compilation peut augmenter fortement
  - Non disponible en OCaml

Le sens des 'a en OCaml

```
module M : sig
  val f : 'a -> 'a
end = struct
  let f x = x + 1
end
(* Error: Signature mismatch:
   ...
   Values do not match:
     val f : int -> int
   is not included in
     val f : 'a -> 'a *)
```

- Erreur : `f` n'est pas polymorphe
- Dans les signatures de modules : les variables de types sont universellement quantifiées

Le sens des 'a en OCaml

```
module M : sig
  val f : 'a -> 'a
end = struct
  let f x = x + 1
end
(* Error: Signature mismatch:
   ...
   Values do not match:
     val f : int -> int
   is not included in
     val f : 'a -> 'a *)
```

- Erreur : `f` n'est pas polymorphe
- Dans les signatures de modules : les variables de types sont universellement quantifiées

```
let f : 'a -> 'a = fun x -> x + 1
(* val f : int -> int *)
```

- Cette fois : pas d'erreur de typage !
- ⚠ Dans les annotations de types : les variables de types sont existentiellement quantifiées
- Ces annotations sont utiles au programmeur pour trouver des incohérences

```
let g : 'a -> 'a = fun x -> x <= 0
(* Error:                      ^^^^^^
   This expression has type bool
   but an expression was expected
   of type int *)
```

Interlude : récursion polymorphe (1)

```
let rec length = function
| [] -> 0
| _ :: xs -> 1 + length xs
(* val length : 'a list -> int *)
```

- Dans cet exemple, la récursion est monomorphe
- C-à-d : les appels récursifs à `length` utilisent tous la même instance α, qui est celle prise en paramètre : « on type le corps de `length` pour α fixé »
- Autrement dit : si on suppose `length` : α list → int (⚠ non généralisé !), alors le corps de la fonction est typable et a le type `length` : α list → int
- Ensuite, on peut généraliser en $\forall\alpha.\,\alpha$ list → int

Interlude : récursion polymorphe (2)

```
let rec length2 = function
| [] -> 0
| [_] -> 1
| [_;_] -> 2
| _ :: _ :: xs ->
  length2 ["OCaml"] + length2 [true] + length2 xs
(*                                ^^^^
   Error:
   This expression has type bool
   but an expression was expected of type string *)
```

- Dans cet exemple, les appels récursifs ont besoin d'instancier α en des types différents (int et bool)
- Mais l'inférence de type reste récursive monomorphe : le typeur interdit de prendre des instances distinctes pour des appels récursifs
- On peut tout de même typer ce programmer en OCaml, en demandant à ce que la récursion soit polymorphe…

Interlude : récursion polymorphe (3)

```
let rec length2: 'a . 'a list -> int = function
| [] -> 0
| [_] -> 1
| [_;_] -> 2
| _ :: _ :: xs ->
  length2 ["OCaml"] + length2 [true] + length2 xs
(* val length2 : 'a list -> int *)
```

- On a rajouté l'annotation de type `'a . 'a list -> int`
  ce qui signifie : $\forall \alpha. \alpha$ list $\rightarrow$ int

Interlude : récursion polymorphe (3)

```
let rec length2: 'a . 'a list -> int = function
| [] -> 0
| [_] -> 1
| [_;_] -> 2
| _ :: _ :: xs ->
  length2 ["OCaml"] + length2 [true] + length2 xs
(* val length2 : 'a list -> int *)
```

- On a rajouté l'annotation de type `'a . 'a list -> int`
  ce qui signifie : $\forall \alpha. \alpha$ list $\rightarrow$ int
- Cette fois, on suppose que `length2` : $\forall \alpha. \alpha$ list $\rightarrow$ int
- Pour typer le corps de `length2`, on peut alors instancier le type de `length2` sur le type string, sur le type bool, et sur une variable de type β
- On peut donc donner au corps le type β list $\rightarrow$ int
- Puis généraliser en $\forall \beta. \beta$ list $\rightarrow$ int

Interlude : récursion polymorphe (4)

- L'inférence de type avec récursion polymorphe est indécidable
- Il est nécessaire d'aider le typeur, à l'aide d'une annotation de type pour indiquer quelles variables doivent être supposées généralisées
- On peut trouver que l'exemple de `length2` est artificiel
- Nous verrons un exemple plus convaincant dans ce cours (qui utilise un type de données récursif non régulier)

Types de données paramétrés

Types paramétrés

- Ce sont des fonctions des types vers les types
- Exemple : le type des listes en OCaml

```
type 'a list = Nil | Cons of 'a * 'a list
```

- Autres noms :
  - « constructeurs de types »
  - « types génériques »

⚠ Ce ne sont pas des types polymorphes

Mais : la valeur `Nil` a un type polymorphe : $\forall \alpha.\ \alpha$ list

Types paramétrés

- Ce sont des fonctions des types vers les types
- Exemple : le type des listes en OCaml

```
type 'a list = Nil | Cons of 'a * 'a list
```

- Autres noms :
  - « constructeurs de types »
  - « types génériques »

⚠ Ce ne sont pas des types polymorphes
Mais : la valeur `Nil` a un type polymorphe : $\forall \alpha.\ \alpha$ list

- Les types paramétrés peuvent être instanciés *a posteriori*
- En OCaml : toutes les instances ont la même représentation

⚠ Les deux types suivants n'ont pas la même représentation en OCaml

- `(int * int) list`
- `type int2_list = Nil | Cons of int * int * int2_list`

🛈 Une passe de monomorphisation *pourrait* transformer
les instances `(int * int) list` en `int2_list`

Fonctions usuelles sur les types paramétrés : `map`

```
type 'a tree =
| Empty
| Node2 of 'a tree * 'a * 'a tree
| Node3 of 'a tree * 'a * 'a tree * 'a * 'a tree
```

- Arbres 2-3 : un cas particulier de B-trees
- Invariants de la structure de données :
  1. C'est un arbre de recherche
  2. Les chemins vers les feuilles ont tous la même taille
- Exercice : étant donné une relation d'ordre totale sur les éléments d'un arbre, programmer les fonctions d'insertion et de suppression dans un arbre 2-3

Fonctions usuelles sur les types paramétrés : `map`

```
type 'a tree =
| Empty
| Node2 of 'a tree * 'a * 'a tree
| Node3 of 'a tree * 'a * 'a tree * 'a * 'a tree


let rec map f = function
| Empty ->
  Empty
| Node2 (l, x, r) ->
  Node2 (map f l, f x, map f r)
| Node3 (l, xl, m, xr, r) ->
  Node3 (map f l, f xl, map f m, f xr, map f r)
(* val map : ('a -> 'b) -> 'a tree -> 'b tree *)
```

Propriétés sur `map`

Étant donné les fonctions :

- `let id x = x`
- `let compose f g x = f (g x)`

La fonction `map` satisfait les propriétés suivantes :

- `map id t = t`
- `map (compose f g) t = compose (map f) (map g) t`

Pour des fonctions `f` et `g` qui sont pures et qui terminent

Fonctions usuelles sur les types paramétrés : `iter`

```
type 'a tree =
| Empty
| Node2 of 'a tree * 'a * 'a tree
| Node3 of 'a tree * 'a * 'a tree * 'a * 'a tree


let rec iter f = function
| Empty ->
  ()
| Node2 (l, x, r) ->
  iter f l; f x; iter f r
| Node3 (l, xl, m, xr, r) ->
  iter f l; f xl; iter f m; f xr; iter f r
(* val iter : ('a -> unit) -> 'a tree -> unit *)
```

Propriétés sur `iter`

Étant donné les fonctions :

- `let ignore x = ()`
- `let seq f g x = f x; g x`

La fonction `iter` satisfait la propriété suivante :

- `iter ignore t = ()` et ne produit pas d'effet de bord

Propriétés sur `iter`

Étant donné les fonctions :

- `let ignore x = ()`
- `let seq f g x = f x; g x`

La fonction `iter` satisfait la propriété suivante :

- `iter ignore t = ()` et ne produit pas d'effet de bord

Que pensez-vous de la propriété suivante ?

- ❓ `iter (seq f g) t = iter f t; iter g t`

Fonctions usuelles sur les types paramétrés : `fold`

```
type 'a tree =
| Empty
| Node2 of 'a tree * 'a * 'a tree
| Node3 of 'a tree * 'a * 'a tree * 'a * 'a tree

let rec fold e f2 f3 = function
| Empty ->
  e
| Node2 (l, x, r) ->
  f2 (fold e f2 f3 l) x (fold e f2 f3 r)
| Node3 (l, xl, m, xr, r) ->
  f3 (fold e f2 f3 l) xl (fold e f2 f3 m) xr (fold e f2 f3 r)
(* val fold :
   'a ->
   ('a -> 'b -> 'a -> 'a) ->
   ('a -> 'b -> 'a -> 'b -> 'a -> 'a) ->
   'b tree ->
   'a                                    *)
```

Propriétés sur `fold`

Étant donné les fonctions :

- `let node2 l x r = Node2 (l, x, r)`
- `let node3 l x m y r = Node3 (l, x, m, y, r)`

La fonction `fold` satisfait les propriétés suivantes :

- `fold Empty node2 node3 t = t`

Exercice :
Étant donné une fonction pure `f`, trouver des programmes `fempty`, `fnode2` et `fnode3` tels que :

- `fold fempty fnode2 fnode3 t = map f t`

Types paramétrés récursifs non réguliers

Un type récursif `'a t` paramétré par une variable `'a` est non régulier lorsque certaines de ses occurrences récursives utilisent une instance autre que `'a`.

Exemple : « random-access lists » (Chris Okasaki)
Ajout d'un élément en tête, accès au k-ième élément : en temps logarithmique

```
type 'a ra_list =
| Empty
| Cons0 of ('a * 'a) ra_list
| Cons1 of 'a * ('a * 'a) ra_list

let l = Cons1 (1, Cons0 (Cons1 (((2, 3), (4, 5)), Empty)))
(* val l : int ra_list *)
```

Types paramétrés récursifs non réguliers

Un type récursif `'a t` paramétré par une variable `'a` est non régulier lorsque certaines de ses occurrences récursives utilisent une instance autre que `'a`.

Exemple : « random-access lists » (Chris Okasaki)
Ajout d'un élément en tête, accès au k-ième élément : en temps logarithmique

```
type 'a ra_list =
| Empty
| Cons0 of ('a * 'a) ra_list
| Cons1 of 'a * ('a * 'a) ra_list

let l = Cons1 (1, Cons0 (Cons1 (((2, 3), (4, 5)), Empty)))
(* val l : int ra_list *)
```

Exercices :

- En utilisant la récursion polymorphe, écrire la fonction `map` pour `ra_list`
- Écrire une fonction `fold` ayant le même type que le `fold` des listes
- Écrire une fonction qui accède au k-ième élément

Récursion ouverte, points fixes

Définitions récursives et points fixes

- Considérons un programme bien connu :

```
let rec fact n =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

Définitions récursives et points fixes

- Considérons un programme bien connu :

```
let rec fact n =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

- La fonction `fact` est solution de l'équation `fact = f_fact fact` pour la fonctionnelle `f_fact` suivante :

```
let f_fact g n =
  if n <= 0
  then 1
  else n * g (n - 1)
(* val f_fact: (int -> int) -> int -> int *)
```

Définitions récursives et points fixes

- Considérons un programme bien connu :

```
let rec fact n =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

- La fonction `fact` est solution de l'équation `fact = f_fact fact` pour la fonctionnelle `f_fact` suivante :

```
let f_fact g n =
  if n <= 0
  then 1
  else n * g (n - 1)
(* val f_fact: (int -> int) -> int -> int *)
```

- On pourrait définir la fonction `fact` de la manière suivante :

```
let rec fact n = f_fact fact n
```

Définitions récursives et points fixes

- Considérons un programme bien connu :

```
let rec fact n =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

- La fonction `fact` est solution de l'équation `fact = f_fact fact` pour la fonctionnelle `f_fact` suivante :

```
let f_fact g n =
  if n <= 0
  then 1
  else n * g (n - 1)
(* val f_fact: (int -> int) -> int -> int *)
```

- On pourrait définir la fonction `fact` de la manière suivante :

```
let rec fact n = f_fact fact n
```

- La fonction `fact` est donc un point fixe de la fonctionnelle `f_fact`

Points fixes

Cherchons à définir une fonction `fix` telle que :

- `fix f` est une fonction de type `'a -> 'b`
- Et `fix f` est un point fixe de `f` :
  Pour tout argument `x` de type `'a`, `(fix f) x = f (fix f) x`

Points fixes

Cherchons à définir une fonction `fix` telle que :

- `fix f` est une fonction de type `'a -> 'b`
- Et `fix f` est un point fixe de `f` :
  Pour tout argument `x` de type `'a`, `(fix f) x = f (fix f) x`
- On a nécessairement : `f : ('a -> 'b) -> 'a -> 'b`

Points fixes

Cherchons à définir une fonction `fix` telle que :

- `fix f` est une fonction de type `'a -> 'b`
- Et `fix f` est un point fixe de `f` :
  Pour tout argument `x` de type `'a`, `(fix f) x = f (fix f) x`
- On a nécessairement : `f : ('a -> 'b) -> 'a -> 'b`
- D'où : `fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b`

Points fixes

Cherchons à définir une fonction `fix` telle que :

- `fix f` est une fonction de type `'a -> 'b`
- Et `fix f` est un point fixe de `f` :
  Pour tout argument `x` de type `'a`, `(fix f) x = f (fix f) x`
- On a nécessairement : `f : ('a -> 'b) -> 'a -> 'b`
- D'où : `fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b`
- C-à-d : `fix` est une fonction d'ordre 3

Points fixes

Cherchons à définir une fonction `fix` telle que :

- `fix f` est une fonction de type `'a -> 'b`
- Et `fix f` est un point fixe de `f` :
  Pour tout argument `x` de type `'a`, `(fix f) x = f (fix f) x`
- On a nécessairement : `f : ('a -> 'b) -> 'a -> 'b`
- D'où : `fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b`
- C-à-d : `fix` est une fonction d'ordre 3

Une solution possible, utilisant une définition récursive en OCaml :

- ```
  let fix f =
    let rec g x = f g x in
    g
  (* val fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b *)
  ```
  ```

Récursion ouverte

Tautologie : « la fonctionnelle `f_fact` prend son point fixe en paramètre »

```
let f_fact self n =
  if n <= 0
  then 1
  else n * self (n - 1)
(* val f_fact: (int -> int) -> int -> int *)
```

- Ce style est appelé « récursion ouverte »
- L'opération `fix f_fact` « noue » la récursion
- C-à-d : `fix f_fact` définit *a posteriori* la valeur pour `self`
- C'est un ingrédient clef de la programmation objet (voir cours 6) :
  - La résolution tardive de `self` est analogue à la notion de « liaison tardive »
  - La récursion ouverte permet de modifier *a posteriori* le comportement d'une fonction (de manière analogue à l'héritage)

Application de la récursion ouverte : tracer les appels récursifs

Une fonction qui modifie une fonctionnelle, pour tracer les appels à `self` :

```
let trace func self arg =
  Printf.printf "Called with argument %i\n" arg;
  let res = func self arg in
  Printf.printf "Returned value from argument %i: %i\n" arg res;
  res
(* val trace : ('a -> int -> int) -> 'a -> int -> int *)

let n = fix (trace f_fact) 5
(* n = 120           Quels messages sont imprimés ? *)
```

Application de la récursion ouverte : mémoïser les appels récursifs

Une fonction qui modifie une fonctionnelle, pour mémoïser les appels à `self` :

```
let memo_fix func =
  let h = Hashtbl.create 128 in
  fun self x ->
    try Hashtbl.find h x
    with Not_found -> begin
      let y = func self x in
      Hashtbl.add h x y;
      y
    end

let f_fib fib n =
  if n <= 0 then 0 else if n = 1 then 1
  else fib (n - 1) + fib (n - 2)

let n = fix (memo_fix (trace f_fib)) 10
(* n = 55            Quels messages sont imprimés ? *)
```

Combinateurs de point fixe

- On peut définir des fonctions récursives sans utiliser le mot-clef **`rec`**, en définissant des combinateurs de point fixe.
- Ces combinateurs `fix` ont pour type : $\forall \alpha\, \beta.\ ((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta$ et satisfont l'équation de point fixe : `fix f x = f (fix f) x`

Combinateurs de point fixe

- On peut définir des fonctions récursives sans utiliser le mot-clef **rec**, en définissant des combinateurs de point fixe.
- Ces combinateurs `fix` ont pour type : $\forall \alpha\, \beta.\ ((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta$ et satisfont l'équation de point fixe : `fix f x = f (fix f) x`
- Combinateur de point fixe de Turing :

```
let fix_turing f = (* nécessite l'option `-rectypes' d'OCaml *)
  let h g f = f (fun x -> g g f x) in
  h h f
```

Combinateurs de point fixe

- On peut définir des fonctions récursives sans utiliser le mot-clef **`rec`**, en définissant des combinateurs de point fixe.
- Ces combinateurs `fix` ont pour type : $\forall \alpha\, \beta.\ ((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta$ et satisfont l'équation de point fixe : `fix f x = f (fix f) x`
- Combinateur de point fixe de Turing :

```
let fix_turing f = (* nécessite l'option `-rectypes' d'OCaml *)
  let h g f = f (fun x -> g g f x) in
  h h f
```

- Exercice : modifier `fix_turing` pour qu'il soit accepté par OCaml sans utiliser l'option `-rectypes`

Combinateurs de point fixe

- On peut définir des fonctions récursives sans utiliser le mot-clef **rec**, en définissant des combinateurs de point fixe.
- Ces combinateurs `fix` ont pour type : $\forall \alpha\, \beta.\ ((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta$ et satisfont l'équation de point fixe : `fix f x = f (fix f) x`
- Combinateur de point fixe de Turing :

```
let fix_turing f = (* nécessite l'option `-rectypes' d'OCaml *)
  let h g f = f (fun x -> g g f x) in
  h h f
```

- Exercice : modifier `fix_turing` pour qu'il soit accepté par OCaml sans utiliser l'option `-rectypes`
- Autre exemple : le « nœud récursif » de Landin :

```
let fix_landin f =
  let r = ref (fun _ -> raise Not_found) in
  let g x = f !r x in
  r := g;
  !r
```

Conclusion

Conclusion

Bilan : Nous avons (*re*)vu dans ce cours :

- Quelques concepts fondamentaux de programmation fonctionnelle
- Des éléments pour votre culture générale en programmation fonctionnelle

Prochain cours : programmation fonctionnelle avancée

Notamment : continuations, suspensions, paresse...