# Programmation fonctionnelle, OCaml #3

Pierre Karpman

Lycée Champollion MP2I

https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

# Table des matières

# Rôle du typage (statique)

Un système de type sert à limiter l'ensemble des programmes valides (que l'on a « le droit d'écrire »), afin de limiter les erreurs à l'exécution

## Interprétation express en trois étapes

Pour pouvoir exécuter un programme d'un certain langage, il faut notamment :

1. [Syntaxe] Qu'il soit écrit en utilisant correctement les constructions du langage
2. [Typage (aussi de la « syntaxe » quand *statique*)] Qu'il soit *typable*
3. [Sémantique] Que l'on puisse donner un sens aux opérations effectuée

## Trois grands types d'erreur

```
1 + (let x = 3) (* Error: Syntax error *)
1 + true (* Error: The constructor true has type bool
            but an expression was expected of type int *)
List.hd [] (* Exception: Failure "hd" *)
```

# Utilité du typage (statique)

## Une aide précieuse à la programmation

- Limite les erreurs à l'exécution
  - Une erreur à l'exécution peut faire exploser une fusée, contrairement (normalement…) à une erreur à (disons) la compilation
- Aide la réflexion
  - On peut se laisser « guider par les types », par exemple pour savoir quels cas de base sont nécessaires dans une récursion

## Avec des inconvénients

- Le typage rajoute des contraintes sur les programmes acceptés
- On peut rejeter certains programmes (non typables) dont l'exécution ne poserait pas de problème

  Exemple :

  ```
  false && (1 + true)
  ```

  - Certains langages typent pas, peu, mal… notamment pour ces raisons

# Utilité du typage (statique) *bis*

## La correspondance preuve/programme 🙈

- Un type représente un théorème
- Un habitant du type (une expression possédant le type) en est une démonstration

Exemple (*cf.* TD) :

```
fun x y z -> x z (y z)
```

fournit une preuve de :

$$(P \to Q \to R) \to (P \to Q) \to P \to R$$

# En OCaml

## Comme on l'a constaté par la pratique

- Le typage est statique et (assez) **strict**
  - On ne peut pas ajouter un `int` à un `bool`, contrairement au C
- Il n'est pas nécessaire en OCaml d'indiquer le type des expressions
  - Les types sont inférés à l'interprétation/la compilation
- On *peut* préciser le type par une *annotation*
  - Syntaxe : `(<expression> : <type>)`
  - Exemples :

```
(3 : int)
(fun x -> x + 3 : int -> int)
([1] : int list)
(1, 2 : int * int)
```

- Certains types sont *polymorphes*
  - Typiquement des types fonctionnels ou construits
  - Exemples :

```
(fun x -> x : 'a -> 'a)
([] : 'a list)
(None : 'a option)
```

# Polymorphisme : vision naïve

Un type polymorphe est un type dont la définition fait intervenir un ou plusieurs *paramètres de type* (notés `'a`, `'b`, …)

Ces paramètres de types permettent d'exprimer des contraintes d'égalité qui devront être satisfaites lors de la *spécialisation*

## Spécialisation

Un type polymorphe peut être (partiellement) *spécialisé* en remplaçant chaque (certains) paramètre de type par un **unique** type (polymorphe ou non) ; on obtient un nouveau type

- Par ex. remplacer toutes les occurrences de `'a` par `int`

## Exemples

- `'a -> 'a` : type des fonctions d'un paramètre d'un type quelconque, s'évaluant en une valeur du même type
- `'a -> 'b -> 'a` : type des fonctions de deux paramètres quelconques, s'évaluant en une valeur du même type que le premier

# Polymorphisme : intérêt

Le polymorphisme permet de définir des types (et les objets correspond) d'une certaine *généricité*

## Types polymorphes

Définissent une structure générale sans se soucier du « contenu »
Par ex. :

- Une liste d'entiers (non vide) a une tête et une queue, comme une liste de booléens, etc.

## Fonctions polymorphes

Une fonction polymorphe n'a besoin d'être implémentée qu'une seule fois pour toutes les spécialisations possibles du type
Par ex. :

- Le calcul de la longueur d'une liste se fait de la même façon quelque soit le type des éléments : c'est mieux si on peut ne l'implémenter qu'une fois

# Polymorphisme : interprétation de types

## Exercice

Que « signifient » ces types ? Quels exemples (intéressants) d'habitants ? Quels éventuels cas d'échecs peuvent-ils causer à leur utilisation ?

```
'a * 'a * 'b
'a list -> bool
'a list -> int
'a list -> 'a
'a list -> 'a option
'a list -> int -> 'a
'a list -> 'b list -> ('a * 'b) list
('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
```

# Polymorphisme : interprétation de `'a -> 'b`

- Le type `'a -> 'b` est celui d'une fonction qui prend un argument de type `'a` et s'évalue en une valeur d'un type `'b` *arbitraire*
- Doit notamment pouvoir s'évaluer en une valeur du type :

  ```
  type bottom = |
  ```

- Or `bottom` est *vide* (comme `void` en C) : il n'a **aucun habitant**
- Une fonction de type `'a -> 'b` doit donc pouvoir faire quelque chose d'impossible ! (D'après la correspondance preuve/programme, elle permet de *tout prouver* (y compris les choses fausses))
- Seule possibilité : la fonction de doit **jamais** terminer son évaluation. Exemples :
  - `let rec f x = f x` : ne termine jamais son évaluation car récursion infinie
  - `fun x -> assert false` : — car interrompue par une *exception*
  - `fun x -> failwith "OHAI"` : pareil
- Autre possibilité : la fonction casse le système de type d'OCaml :(

# Polymorphisme en OCaml : limitations

## Paramètres de type : tout ou rien

- On ne peut pas restreindre les paramètres à un sous-ensemble des types
- Mais certaines valeurs de type polymorphe on en fait *quand même* des contraintes
- Dans ce cas, le typage ne protège pas complètement d'erreurs à l'exécution

Exemple :

```
utop # (=);;
- : 'a -> 'a -> bool = <fun>
utop # (=) (=) (=);;
Exception: Invalid_argument "compare: functional value".
```

## Paramètres de type *faibles*

On obtient parfois des paramètres de type « `_weak1` » etc.
Explications (partielles) (haha) à la prochaine partie…

# Inférence de types : aperçu

Le *système de type* utilisé en OCaml a une *inférence* qui est *décidable* : étant donnée une expression (ou phrase) OCaml syntaxiquement valide, il existe un algorithme qui peut toujours :

- la rejeter comme non typable si elle ne l'est pas (sans se tromper)
- lui attribuer un type (correct), qui sera le plus général possible (au sens du polymorphisme ; on ne spécialise pas inutilement)

## Principe express de l'algorithme 🙈

On introduit des inconnues pour chaque sous-expression intervenant dans l'expression (ou phrase) à typer, et un mécanisme de propagation de contrainte pour les déterminer. On utilise le polymorphisme paramétrique lorsqu'il n'y a pas assez de contraintes. L'une des difficultés de mise en œuvre est l'ordre dans lequel procéder en présence de `let` (éventuellement multiples).

# Table des matières

# Rappels

## Fonctions *anonymes*

Expressions de type fonction :

`fun <params> -> ef`

Avec `<params>` un ou plusieurs noms de paramètres utilisables dans `ef`, et liés aux arguments lors de l'appel (par ex. : `(fun x -> x * x) 3` s'évalue à `x * x` où l'on a substitué `3` à `x`, donc `3 * 3`, soit la même chose que `let x = 3 in x * x`)

- Passage par valeur

## Fonctions globalement liées à un identifiant

```
let f = fun <params> -> ef
let f <params> = ef (* alternative *)
```

Si récursives :

```
let rec f = fun <params> -> ef
let rec f <params> = ef (* alternative *)
```

# Syntaxe de définition de fonction : annotation de types

## Re : annotation d'une expression

- On peut *annoter* toute expression `e` OCaml avec son type `t` avec la syntaxe `(e:t)`
- Fonctionne aussi pour un identifiant (pareil)

## Annotation des paramètres d'une fonction

- La syntaxe ci-dessus peut aussi être utilisée pour annoter les paramètres d'une fonction lors de sa définition
- On peut également annoter le type de l'évaluation par un `: t` terminal

Exemples :

```
let add (x:int) (y:int) : int = x + y
let add (x:int) y : int = x + y (* pas nécessaire de tout annoter *)
let is_empty (x:'a list) : bool = x = []
```

# Annotation de type : utilité

## Documentation du code

Les annotations de type rendent le code source plus explicite ; elles peuvent aider à ne pas confondre l'ordre des arguments d'une fonction

## Sur-spécialisation

On peut spécialiser de force le type de n'importe quelle expression polymorphe par une annotation ; c'est éventuellement parfois utile pour des fonctions. Exemple :

```
utop # let is_empty x = x = [];;
utop # is_empty [false];;
- : bool = false
utop # let is_empty (x:int list) = x = [];;
utop # is_empty [false];;
Error: The constructor false has type bool but an expression was
expected of type int
```

# Définition de fonction : fonctions locales

## Observations

- Le corps d'une fonction est une expression
- On dispose d'expressions de type fonction (les fonctions anonymes)
- On dispose d'une expression `let in`

## Fonctions locales

On peut définir une fonction *localement* dans une autre comme par exemple :

```
let f x =
    (* début du corps de la fonction f *)
    let _f x y z = <corps de la fonction _f>
    in
    (* suite du corps de la fonction f,
       qui peut utiliser _f *)
;;
(* ici, _f n'est plus lié à la fonction locale de f *)
```

# Fonctions locales : utilité

## Avantages, inconvénients de la localité

- Évite de polluer l'espace des identifiants
- Pas directement accessible pour tester, débugger, tracer…

## Usage typique

Il est courant qu'une fonction récursive ait besoin d'un ou plusieurs paramètres à usage « interne » ; on peut alors faire (par ex.) :

```
let rev x =
    let rec _rev tl = function
        | [] -> tl
        | x::xs -> _rev (x::tl) xs
    in _rev [] x
```

# Le mensonge de la non-unarité des fonctions

Techniquement, toute fonction OCaml prend **exactement un** paramètre (est unaire)

## Type d'une fonction OCaml

Une fonction a *toujours* un type $t_1 \to t_2$ avec $t_1$ le type de son unique paramètre, et $t_2$ celui de son évaluation

- Mais $t_2$ peut *aussi* être un type fonction. Exemples :
  - `int -> int -> int` (identique à `int -> (int -> int)`) : fonction d'un paramètre `int` s'évaluant en (une fonction d'un paramètre `int` s'évaluant en un `int`)
  - `(int -> int) -> int` : fonction d'un paramètre (fonction d'un paramètre `int` s'évaluant en un `int`) s'évaluant en un `int`
- Ceci permet en pratique d'écrire des fonctions prenant « plusieurs » paramètres

## Curryfication

Le processus ci-dessus est (généralement) nommé *curryfication* (en anglais : *currying*), d'après Haskell B. Curry

# L'une des sources du mensonge : un sucre syntaxique

La syntaxe :

```
fun x y -> e
```

est en fait un *sucre* pour :

```
fun x -> (fun y -> e)
```

(de même pour encore plus de paramètres)

# Applications partielles

## Définition (informelle)

Soit $f$ une fonction de type $t_0 \to t_1 \to \cdots \to t_n$ avec $t_n$ un type non fonctionnel, toute application de $f$ à moins de $n$ arguments :

- s'évalue en un type fonctionnel
- est dite *partielle*

## Exemple

```
let add x y = x + y
let add3 = add 3 (* équivalent à fun x -> 3 + x *)
```

## Intérêt

Permet de « spécialiser » une fonction générale en en fixant certains paramètres

# Conséquences des applications partielles

## Ordre des paramètres

Une application partielle ne peut se faire qu'en fournissant les arguments dans l'ordre de définition des paramètres

- Certains ordres de définition peuvent être plus « pratiques » que d'autres
- (Mais l'on peut toujours redéfinir une nouvelle fonction qui appelle juste la première avec un ordre différent)

## Fermeture

Une application partielle *capture* les arguments fournis dans une fermeture, en attendant le reste de l'évaluation

- Équivalent à avoir ajouté autant de `let in`, avec les mêmes conséquences d'immuabilité. Par ex., `(fun x y -> ef) 3` équivaut à `let x = 3 in fun y -> ef`
- (De la même façon que `(fun x -> ef) ea` équivaut à `let x = ea in ef`)
- (Avec quelques subtilités sur le typage que l'on ignorera)

# Conséquence des fermetures : paramètres de types faibles

## Un exemple

```
utop # let f x y = x <> [] && (List.hd x) = y;;
val f : 'a list -> 'a -> bool = <fun>
utop # f [1];;
- : int -> bool = <fun>
utop # let fp = f [];;
val fp : '_weak1 -> bool = <fun>
utop # fp 1;;
- : bool = false
utop # fp;;
- : int -> bool = <fun>
```

## Que se passe-t'il ?

- `f [1]` est spécialisé en `int -> bool` comme attendu
- On pourrait s'attendre à ce que `fp` soit de type `'a -> bool`
  - À la place, on obtient un paramètre polymorphique *faible* `'_weak1`, qui dès sa *première* spécialisation (n'importe où) se dégrade en cette spécialisation

# Paramètres de types faibles : pourquoi ?

- Ne seraient pas nécessaire dans un langage *purement fonctionnel* (comme le fragment OCaml vu jusqu'à présent)
- Mais OCaml permet (aussi) de modifier des états (*via* des *références* ; *cf.* un peu plus tard)
- Le polymorphisme faible empêche d'utiliser celles-ci pour « casser » le système de type (par ex. en spécialisant sournoisement un type polymorphe)

Pour en savoir plus : https://ocamlverse.net/content/weak_type_variables.html

# Illustration en avant

## Prélude

```
utop # let fimp x y = !x <> [] && (List.hd !x) = y;;
val fimp : 'a list ref -> 'a -> bool = <fun>
utop # let trap = ref [];;
val trap : '_weak1 list ref = {contents = []}
utop # let fimpp = fimp trap;;
val fimpp : '_weak1 -> bool = <fun>
```

## Suite 1

```
utop # trap := [1];;
- : unit = ()
utop # trap;;
- : int list ref = {contents = [1]}
utop # fimpp;;
- : int -> bool = <fun>
```

## Suite 1'

```
utop # fimpp true;;
- : bool = false
utop # fimpp;;
- : bool -> bool = <fun>
utop # trap;;
- : bool list ref = {contents = []}
```