

# LIFPF – Programmation fonctionnelle

## CM1 – $\lambda$ -calcul

Licence informatique UCBL – Printemps 2022–2023

<https://forge.univ-lyon1.fr/programmation-fonctionnelle/lifpf/-/blob/main/README.md>

# λ-calcul

Comment noter différemment la même chose

en maths	$x \mapsto x$	$f \mapsto x \mapsto f(f(x))$
en λ-calcul	$\lambda x.x$	$\lambda f.\lambda x.f (f x)$
en OCaml	<code>fun x -&gt; x</code>	<code>fun f -&gt; fun x -&gt; f(f x)</code>

# λ-calcul

Comment noter différemment la même chose

en maths	$x \mapsto x$	$f \mapsto x \mapsto f(f(x))$
en <b>λ-calcul</b>	$\lambda x.x$	$\lambda f.\lambda x.f (f x)$
en OCaml	<code>fun x -&gt; x</code>	<code>fun f -&gt; fun x -&gt; f(f x)</code>

1936 : Alonzo Church invente le λ-calcul

...

1970 : le λ-calcul explose avec ses applications à l'informatique

...

2004 : paradigme map/reduce présenté par Google à OSDI

# Bases du $\lambda$ -calcul : syntaxe

## Syntaxe du $\lambda$ -calcul

L'ensemble  $\Lambda$  des *termes*<sup>a</sup> du  $\lambda$ -calcul est *le plus petit* ensemble qui contient :

- $x$  si  $x \in Var$ , avec  $Var$  un ensemble (donné) de variables
- $\lambda x.M$  si  $M \in \Lambda$  et  $x \in Var$
- $(M N)$  si  $M \in \Lambda$  et  $N \in \Lambda$

---

a. dits aussi *expressions*, *formules* ou même *phrases*

# Bases du $\lambda$ -calcul : syntaxe

## Syntaxe du $\lambda$ -calcul

L'ensemble  $\Lambda$  des *termes*<sup>a</sup> du  $\lambda$ -calcul est *le plus petit* ensemble qui contient :

- $x$  si  $x \in Var$ , avec  $Var$  un ensemble (donné) de variables
- $\lambda x.M$  si  $M \in \Lambda$  et  $x \in Var$
- $(M N)$  si  $M \in \Lambda$  et  $N \in \Lambda$

---

a. dits aussi *expressions*, *formules* ou même *phrases*

En fixant un ensemble fini de constructeurs (syntaxiques) et en définissant **le plus petit ensemble qui est clos par ces constructeurs**, on définit un ensemble **par induction**. On pourrait écrire plus concisément :

$$M, N ::= x \in Var \mid \lambda x.M \mid (M N)$$

# Bases du $\lambda$ -calcul : syntaxe

## Intuition des expressions

$x \in Var$  une expression atomique, une boîte noire,

$\lambda x.M$  une fonction <sup>a</sup> de paramètre formel  $x$  dont le corps est  $M$ ,

$(M N)$  l'application d'une fonction  $M$  avec  $N$  passé en paramètre.

---

a. une fonction *unaire*, en  $\lambda$ -calcul, tout est curryfié *par défaut* !

# Bases du $\lambda$ -calcul : syntaxe - suite

## Conventions

- élision des  $\lambda$  : on écrit  $\lambda x_1 \dots x_n. M$  pour  $\lambda x_1. (\dots (\lambda x_n. M) \dots)$
- application à gauche : on écrit  $(M \ N_1 \dots M_p)$  pour  $(\dots (M \ N_1) \dots M_p)$

## Bases du $\lambda$ -calcul : syntaxe - suite

### Conventions

- élision des  $\lambda$  : on écrit  $\lambda x_1 \dots x_n. M$  pour  $\lambda x_1. (\dots (\lambda x_n. M) \dots)$
- application à gauche : on écrit  $(M \ N_1 \dots M_p)$  pour  $(\dots (M \ N_1) \dots M_p)$

### Exemple

On écrit  $\lambda xyz. (x \ z \ (y \ z))$  pour  $\lambda x. (\lambda y. (\lambda z. ((x \ z)(y \ z))))$

## Bases du $\lambda$ -calcul : variables libres

Soit la fonction  $FV^a$  définie par **induction** sur la structure des termes :

- $FV(x) = \{x\}$
- $FV(\lambda x.M) = FV(M) \setminus \{x\}$
- $FV(M N) = FV(M) \cup FV(N)$

---

a.  $FV$  pour *free variables*

# Exemples

$$(\lambda x. \lambda y. (\lambda z. xz)u)v$$

- $FV((\lambda x. \lambda y. (\lambda z. xz)u)v) = \{u, v\}$

$$(\lambda x. \lambda y. xz)(\lambda z. z)$$

- $FV((\lambda x. \lambda y. xz)(\lambda z. z)) = \{z\}$

# Substitutions

On écrit  $M[x := P]$  pour le terme  $M$  dont toutes les occurrences de la variable  $x$  ont été remplacées par le terme  $P$ , on a substitué  $x$  par  $P$ .

# Substitutions

On écrit  $M[x := P]$  pour le terme  $M$  dont toutes les occurrences de la variable  $x$  ont été remplacées par le terme  $P$ , on a substitué  $x$  par  $P$ .

## Danger

La substitution doit faire attention à ne pas rendre libres des variables qui ne l'étaient pas ou vice-versa !

$$\lambda y. x[x := y] \neq \lambda y. y$$

$$\lambda y. x[y := x] \neq \lambda x. x$$

# Substitutions

## Substitution sans capture

- ①  $x[x := P] = P$
- ②  $y[x := P] = y$  si  $x \neq y$
- ③  $(MN)[x := P] = (M[x := P])(N[x := P])$
- ④  $(\lambda x.M)[x := P] = \lambda x.M$
- ⑤  $(\lambda y.M)[x := P] = \lambda y.(M[x := P])$  si  $x \neq y$  et  $y \notin FV(P)$

# Exemples

- $(\lambda x.(xy))[y := (\lambda z.z)] = \lambda x.x(\lambda z.z)$
- $(\lambda x.(xy) \ \lambda z.(zy))[y := (\lambda u.u)] = (\lambda x.(x \ \lambda u.u) \ \lambda z.(z \ \lambda u.u))$
- $(\lambda x.xz)[y := (\lambda u.u)] = (\lambda x.xz)$
- $(\lambda x.xy)[y := (\lambda x.x)] = (\lambda x.x(\lambda x.x))$
  
- $(\lambda x.x\ x)y[y := (\lambda z.z\ x)]$  non défini car capture de  $x$

# Substitutions : renommages

## $\alpha$ -conversion, ou $\alpha$ -renommage

La relation  $\equiv_\alpha$  est étendue à une congruence sur l'ensemble des termes à partir de la relation suivante :

$$(\lambda y.M) \equiv_\alpha \lambda z.(M[y := z]) \text{ si } z \notin FV(M)$$

L' $\alpha$ -conversion capture l'idée que les variables liées sont interchangeables, comme  $x$  dans  $\int f(x).dx$ , dans  $\forall x.P(x)$  ou dans  $(x) \Rightarrow f(x)$ .

# Substitutions : renommages

## $\alpha$ -conversion, ou $\alpha$ -renommage

La relation  $\equiv_\alpha$  est étendue à une congruence sur l'ensemble des termes à partir de la relation suivante :

$$(\lambda y.M) \equiv_\alpha \lambda z.(M[y := z]) \text{ si } z \notin FV(M)$$

L' $\alpha$ -conversion capture l'idée que les variables liées sont interchangeables, comme  $x$  dans  $\int f(x).dx$ , dans  $\forall x.P(x)$  ou dans  $(x) \Rightarrow f(x)$ .

On utilisera par la suite la convention de Barendregt : « **il n'existe aucun sous-terme dans lequel une variable apparaît à la fois libre et liée (c.-à-d. dans un  $\lambda$ ).** »

On peut pour cela utiliser l' $\alpha$ -conversion avec pour  $z$  une nouvelle variable, dite **fraîche**, c.-à-d. jamais utilisée jusqu'ici.

# Le calcul

Comment exprimer le « calcul », en  $\lambda$ -calcul ?

$\beta$ -réduction, ou  $\beta$ -contraction

$$(\lambda x.M)N \xrightarrow[\beta]{} M[x := N]$$

Calculer revient ainsi à *réduire* un terme en remplaçant les arguments des fonctions par les expressions passées en paramètres lors de l'appel.

# Le calcul

Comment exprimer le « calcul », en  $\lambda$ -calcul ?

$\beta$ -réduction, ou  $\beta$ -contraction

$$(\lambda x.M)N \xrightarrow[\beta]{} M[x := N]$$

Calculer revient ainsi à *réduire* un terme en remplaçant les arguments des fonctions par les expressions passées en paramètres lors de l'appel.

## Exemples

- $(\lambda x.(x\ x)\ \lambda y.y) \xrightarrow[\beta]{} (\lambda y.y\ \lambda y.y) \equiv_{\alpha} (\lambda z.z\ \lambda y.y) \xrightarrow[\beta]{} (\lambda y.y)$
- $((\lambda x.x\ \lambda x.x)\ y) \equiv_{\alpha} ((\lambda x.x\ \lambda z.z)\ y) \xrightarrow[\beta]{} (\lambda z.z\ y)$

# Stratégies d'évaluation

Stratégie : décider quel (sous) terme réduire ?

$$(\lambda x.(\lambda y.(yx) \ x) \ \lambda z.z) \xrightarrow{\beta} (\lambda x.(xx) \ \lambda z.z) \xrightarrow{\beta} (\lambda z.z \ \lambda z.z)$$

# Stratégies d'évaluation

Stratégie : décider quel (sous) terme réduire ?

$$\begin{aligned}
 (\lambda x.(\lambda y.(yx) \ x) \ \lambda z.z) &\xrightarrow{\beta} (\lambda x.(xx) \ \lambda z.z) \xrightarrow{\beta} (\lambda z.z \ \lambda z.z) \\
 \text{ou bien } &\xrightarrow{\beta} (\lambda y.(y \ \lambda z.z) \ \lambda z.z) \xrightarrow{\beta} (\lambda z.z \ \lambda z.z)
 \end{aligned}$$

Même résultat (ouf)

# Stratégie paresseuse (ou normale ou *left-most outer-most*)

- N'évalue que si nécessaire
- Peut conduire à des doublons dans l'évaluation
- Prudente : termine s'il est possible de terminer

$$\begin{aligned}
 & (\lambda x. \lambda y. x \ \lambda k. k) (\lambda u. uu \ \lambda v. vv) \\
 & \quad \xrightarrow[\beta]{} (\lambda y. \lambda k. k \ (\lambda u. uu \ \lambda v. vv)) \\
 & \quad \xrightarrow[\beta]{} \lambda k. k
 \end{aligned}$$

## Stratégie avide (ou *eager* ou *right-most inner-most*)

- Évalue les arguments avant de les passer à la fonction
- Certaines évaluations peuvent être inutiles et même faire boucler l'évaluation
- Plus efficace si toutes les expressions sont à réduire

$$\begin{aligned}
 & (\lambda x. \lambda y. x \ \lambda k. k)(\lambda u. (uu) \ \lambda v. (vv)) \\
 & \quad \xrightarrow[\beta]{} (\lambda x. \lambda y. x \ \lambda k. k)(\lambda v. (vv) \ \lambda v. (vv)) \\
 & \quad \equiv_{\alpha} (\lambda x. \lambda y. x \ \lambda k. k)(\lambda u. (uu) \ \lambda v. (vv))
 \end{aligned}$$

# Stratégie d'évaluation d'Ocaml

## Stratégie en Ocaml

- utilise toujours la stratégie **avide**
  - comme la majorité des langages
- plus efficace
  - évite maintenir en mémoire les structures à évaluer

## Pourquoi le $\lambda$ -calcul ?

Le  $\lambda$ -calcul est la *base théorique* de l'évaluation des programmes fonctionnels.

Le  $\lambda$ -calcul est *fondamental* pour **raisonner sur les programmes** et pour **écrire des compilateurs/interpréteurs**.

## $\lambda$ -calcul + arithmétique

On ajoute au  $\lambda$ -calcul

- les constantes représentant les entiers 1, 2, *etc.*
- les opérateurs  $+$  et  $*$

$\beta$ -réduction

étendue à  $+$  et  $*$  de manière naturelle

- ne fonctionne pas si leurs arguments ne sont pas des nombres

# λ-calcul + arithmétique

On ajoute au λ-calcul

- les constantes représentant les entiers 1, 2, *etc.*
- les opérateurs + et \*

β-réduction

étendue à + et \* de manière naturelle

- ne fonctionne pas si leurs arguments ne sont pas des nombres

Exemples

$$\bullet ((\lambda x.(2 + x))\ 4) * 3 \xrightarrow{\beta} (2 + 4) * 3 \xrightarrow{\beta} 6 * 3 \xrightarrow{\beta} 18$$

# $\lambda$ -calcul + arithmétique

On ajoute au  $\lambda$ -calcul

- les constantes représentant les entiers 1, 2, *etc.*
- les opérateurs + et \*

$\beta$ -réduction

étendue à + et \* de manière naturelle

- ne fonctionne pas si leurs arguments ne sont pas des nombres

Exemples

- $((\lambda x.(2 + x))\ 4) * 3 \xrightarrow{\beta} (2 + 4) * 3 \xrightarrow{\beta} 6 * 3 \xrightarrow{\beta} 18$
- $(\lambda x.x) + 5 \not\xrightarrow{\beta}$

# $\lambda$ -calcul + arithmétique

On ajoute au  $\lambda$ -calcul

- les constantes représentant les entiers 1, 2, *etc.*
- les opérateurs + et \*

$\beta$ -réduction

étendue à + et \* de manière naturelle

- ne fonctionne pas si leurs arguments ne sont pas des nombres

Exemples

- $((\lambda x.(2 + x))\ 4) * 3 \xrightarrow{\beta} (2 + 4) * 3 \xrightarrow{\beta} 6 * 3 \xrightarrow{\beta} 18$
- $(\lambda x.x) + 5 \not\xrightarrow{\beta}$

Typage : vérifier que les expressions sont “bien construites”

# Types

## Ensemble $T$ des types

défini inductivement par :

- `number`  $\in T$  est un type
- si  $\tau_1$  et  $\tau_2$  sont des types ( $\tau_1 \in T$  et  $\tau_2 \in T$ ), alors  
 $(\tau_1 \rightarrow \tau_2) \in T$

Notation : parenthèses optionnelles, par défaut autour de la flèche la plus à droite

# Types

## Ensemble $T$ des types

défini inductivement par :

- $\text{number} \in T$  est un type
- si  $\tau_1$  et  $\tau_2$  sont des types ( $\tau_1 \in T$  et  $\tau_2 \in T$ ), alors  $(\tau_1 \rightarrow \tau_2) \in T$

Notation : parenthèses optionnelles, par défaut autour de la flèche la plus à droite

## Exemples de types

$\text{number}, (\text{number} \rightarrow \text{number}), (\text{number} \rightarrow (\text{number} \rightarrow \text{number}))$   
 $((\text{number} \rightarrow \text{number}) \rightarrow \text{number})$

# Types

## Exemples de termes typés

- $2 : \text{number}$
- $x + 2 : \text{number}$
- $\lambda x. (x + 2) : \text{number} \rightarrow \text{number}$

# Types

## Exemples de termes typés

- $2 : \text{number}$
- $x + 2 : \text{number}$
- $\lambda x. (x + 2) : \text{number} \rightarrow \text{number}$

(si  $x : \text{number}$ )

# Règles de typage

## Jugement de typage

$$\Gamma \vdash M : \tau$$

où  $\Gamma = \{x_1 : \tau_1, \dots, x_n : \tau_n\}$

## Exemples

- $\{x : \text{number}\} \vdash x + 3 : \text{number}$
- $\emptyset \vdash 2 + 3 : \text{number}$
- $\{y : \text{number} \rightarrow \text{number}\} \vdash (y \ 3) : \text{number}$
- $\emptyset \vdash (\lambda x. x + 3) : \text{number} \rightarrow \text{number}$

# Règles de typage - 1

## Règles de typage générales du $\lambda$ -calcul

$$(Var) \frac{}{\Gamma \vdash x : \tau} \text{ si } x : \tau \in \Gamma$$

$$(Fun) \frac{\Gamma, x : \tau \vdash M : \tau'}{\Gamma \vdash \lambda x.M : \tau \rightarrow \tau'}$$

$$(App) \frac{\Gamma \vdash M : \tau \rightarrow \tau' \quad \Gamma \vdash N : \tau}{\Gamma \vdash (MN) : \tau'}$$

## Règles de typage - 2

### Règles spécifiques à $+$ , $*$ et aux constantes

$$(Plus) \quad \frac{\Gamma \vdash M : \text{number} \quad \Gamma \vdash N : \text{number}}{\Gamma \vdash M + N : \text{number}}$$

$$(Mult) \quad \frac{\Gamma \vdash M : \text{number} \quad \Gamma \vdash N : \text{number}}{\Gamma \vdash M * N : \text{number}}$$

$$\Gamma \vdash \frac{}{n : \text{number}} \text{ si } n \text{ est une constante numérique}$$

# Exemple de typage

## Intuitivement

$$\begin{array}{c}
 (\lambda x. \underbrace{x}_{\text{number}} * \underbrace{3}_{\text{number}}) \underbrace{4}_{\text{number}} \\
 \underbrace{\hspace{10em}}_{\text{number}} \\
 \underbrace{\hspace{10em}}_{\text{number} \rightarrow \text{number}} \\
 \underbrace{\hspace{10em}}_{\text{number}}
 \end{array}$$

## Plus formellement

$$\begin{array}{c}
 \frac{}{x : \text{number} \vdash x : \text{number}} \quad \frac{}{x : \text{number} \vdash 3 : \text{number}} \\
 \hline
 x : \text{number} \vdash x * 3 : \text{number} \\
 \hline
 \vdash (\lambda x : x * 3) : \text{number} \rightarrow \text{number} \qquad \vdash 4 : \text{number} \\
 \hline
 \vdash (\lambda x : x * 3) 4 : \text{number}
 \end{array}$$

# Références

Pour ce cours, les ressources suivantes ont été utilisées (cliquer pour suivre) :

- Cours de Pierre Lescanne : [Programmation 2](#) en L3 ENS Lyon
- Cours de Didier Rémy : [Type systems: Simply typed lambda-calculus](#) au MPRI
- Henk Barendregt & Erik Barendsen : [Introduction to Lambda Calculus](#)