LIFAP5 – Programmation fonctionnelle pour le WEB

CM2 – λ-calcul

Licence informatique UCBL – Printemps 2020–2021

http://emmanuel.coquery.pages.univ-lyon1.fr/enseignement/lifap5/

Plan

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 Haskell	• `\x -> x`	• `\f -> \x -> f(f x)`
• en JavaScript (ES6)	• `x => x`	• `f => x => f(f(x))`

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

2015 : JavaScript introduit la notation λ avec ES6

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 Haskell	• `\x -> x`	• `\f -> \x -> f(f x)`
• en JavaScript (ES6)	• `x => x`	• `f => x => f(f(x))`

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

2015 : JavaScript introduit la notation λ avec ES6

Bases du λ-calcul : syntaxe

Syntaxe du λ-calcul

L'ensemble Λ des *termes*[a] du λ-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 $x \in Var$
- (MN) si $M \in \Lambda$ et $N \in \Lambda$

a. dit 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 :

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

Bases du λ-calcul : syntaxe

Syntaxe du λ-calcul

L'ensemble Λ des *termes*[a] du λ-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 $x \in Var$
- (MN) si $M \in \Lambda$ et $N \in \Lambda$

a. dit 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 :

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

Bases du λ-calcul : syntaxe

Intuition des expressions

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

$\lambda x.M$ une fonction [a] de paramètre formel x dont le corps est M,

MN l'application d'une fonction M avec N passé en paramètre.

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

Conventions

- élision des λ : on écrit $\lambda x_1 \dots x_n.M$ pour $\lambda x_1.(\dots(\lambda x_n.M)\dots)$
- application à gauche : on écrit $(MN_1 \dots M_p)$ pour $(\dots(MN_1)\dots M_p)$

Exemple

on écrit $\lambda xyz.xz(yz)$ pour $\lambda x.(\lambda y.(\lambda z.((xz)(yz))))$

Bases du λ-calcul : syntaxe

Intuition des expressions

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

$\lambda x.M$ une fonction [a] de paramètre formel x dont le corps est M,

MN l'application d'une fonction M avec N passé en paramètre.

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

Conventions

- élision des λ : on écrit $\lambda x_1 \dots x_n.M$ pour $\lambda x_1.(\dots(\lambda x_n.M)\dots)$
- application à gauche : on écrit $(MN_1 \dots M_p)$ pour $(\dots(MN_1)\dots M_p)$

Exemple

on écrit $\lambda xyz.xz(yz)$ pour $\lambda x.(\lambda y.(\lambda z.((xz)(yz))))$

Bases du λ-calcul : variables libres et liées

Soit la fonction BV [a] définie par induction sur la structure des termes :

- $BV(x) = \emptyset$
- $BV(\lambda x.M) = BV(M) \cup \{x\}$
- $BV(MN) = BV(M) \cup BV(N)$

a. BV pour *bound variables*

Soit la fonction FV [a] définie par induction sur la structure des termes :

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

a. FV pour *free variables*

Bases du λ-calcul : variables libres et liées

Soit la fonction BV [a] définie par induction sur la structure des termes :

- $BV(x) = \emptyset$
- $BV(\lambda x.M) = BV(M) \cup \{x\}$
- $BV(MN) = BV(M) \cup BV(N)$

a. BV pour *bound variables*

Soit la fonction FV [a] définie par induction sur la structure des termes :

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

a. FV pour *free variables*

Exemples

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

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

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

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

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.

Danger

La substitution doit faire attention à ne pas rendre libres des variables liées ou vice-versa !

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

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

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 liées 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

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

Substitutions : renommages

α-conversion, ou α-renommage

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

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

La α-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) => 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. »

On peut pour cela utiliser l'α-conversion avec pour z une nouvelle variable, dite fraîche, *i.e.* jamais utilisée jusqu'ici.

Substitutions : renommages

α-conversion, ou α-renommage

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

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

La α-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) => 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. »

On peut pour cela utiliser l'α-conversion avec pour z une nouvelle variable, dite fraîche, *i.e.* jamais utilisée jusqu'ici.

Le calcul

Comment exprimer le « calcul », en λ-calcul ?

β-réduction, ou β-contraction

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

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

Exemples

- $\lambda x.(xx)(\lambda y.y) \underset{\beta}{\rightarrow} (\lambda y.y)(\lambda y.y) \equiv_\alpha (\lambda z.z)(\lambda y.y) \underset{\beta}{\rightarrow} (\lambda y.y)$
- $(\lambda x.x(\lambda x.x))y \equiv_\alpha (\lambda x.x(\lambda z.z))y \underset{\beta}{\rightarrow} y(\lambda z.z)$

Le calcul

Comment exprimer le « calcul », en λ-calcul ?

β-réduction, ou β-contraction

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

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

Exemples

- $\lambda x.(xx)(\lambda y.y) \underset{\beta}{\rightarrow} (\lambda y.y)(\lambda y.y) \equiv_\alpha (\lambda z.z)(\lambda y.y) \underset{\beta}{\rightarrow} (\lambda y.y)$
- $(\lambda x.x(\lambda x.x))y \equiv_\alpha (\lambda x.x(\lambda z.z))y \underset{\beta}{\rightarrow} y(\lambda z.z)$

Stratégies d'évaluation

Stratégie : décider quel (sous) terme réduire ?

$$(\ \lambda x.((\lambda y.yx)x)\ )\ (\lambda z.z) \quad \underset{\beta}{\rightarrow} (\ \lambda x.(xx)\ )\ (\lambda z.z) \quad \underset{\beta}{\rightarrow} (\lambda z.z)\ (\lambda z.z)$$

ou bien $\underset{\beta}{\rightarrow} ((\lambda y.y(\lambda z.z))(\lambda z.z)) \quad \underset{\beta}{\rightarrow} (\lambda z.z)\ (\lambda z.z)$

Même résultat (ouf)

Stratégies d'évaluation

Stratégie : décider quel (sous) terme réduire ?

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

$$\text{ou bien} \quad \underset{\beta}{\rightarrow} ((\lambda y.y(\lambda z.z))(\lambda z.z)) \underset{\beta}{\rightarrow} (\lambda z.z)\ (\lambda z.z)$$

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 si il est possible de terminer

$$(\lambda x.\lambda y.x)(\lambda k.k)((\lambda u.uu)(\lambda v.vv))$$
$$\underset{\beta}{\rightarrow} (\lambda y.(\lambda k.k))((\lambda u.uu)(\lambda v.vv))$$
$$\underset{\beta}{\rightarrow} (\lambda k.k)$$

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

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

λ-calcul et JavaScript

Traduction λ-calcul $\rightsquigarrow$ JavaScript

- $x \rightsquigarrow x$
- si $M_\lambda \rightsquigarrow M_{js}$ alors $\lambda x.M_\lambda \rightsquigarrow x \texttt{ => } M_{js}$
- si $M_\lambda \rightsquigarrow M_{js}$ et $N_\lambda \rightsquigarrow N_{js}$ alors $(M_\lambda N_\lambda) \rightsquigarrow M_{js}(N_{js})$

Exemples

- $\lambda x.x \rightsquigarrow$ `x => x` $\equiv$ `function(x){ return x; }`
- $\lambda x.\lambda y.(xy) \rightsquigarrow$ `x => y => x(y)`
  $\equiv$ `function(x){ return function(y){ return x(y); }; }`
- $(\lambda x.\lambda y.(xy))(\lambda z.z) \rightsquigarrow$ `(x => y => x(y))(z => z)`

Stratégie d'évaluation de JavaScript

Stratégie en JavaScript

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

Du λ-calcul au JavaScript, ou le mariage de la carpe et du lapin

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

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

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

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

Typage : vérifier que les expressions sont "bien construites"

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

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

Typage : vérifier que les expressions sont "bien construites"

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

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

Typage : vérifier que les expressions sont "bien construites"

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

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

Typage : vérifier que les expressions sont “bien construites”

Types

Ensemble T des types

défini inductivement par :

- $\texttt{number} \in T$ est un type
- si τ_1 et τ_2 sont des types ($\tau_1 \in T$ et $\tau_2 \in T$), alors $(\tau_1 \to \tau_2) \in T$

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

Exemples de types

$\texttt{number}, (\texttt{number} \to \texttt{number}), (\texttt{number} \to (\texttt{number} \to \texttt{number}))$
$((\texttt{number} \to \texttt{number}) \to \texttt{number})$

Exemples de termes typés

- $2 : \texttt{number}$
- $x + 2 : \texttt{number}$ (si $x : \textit{number}$)
- $\lambda x.(x + 2) : \texttt{number} \to \texttt{number}$

Types

Ensemble T des types

défini inductivement par :

- $\texttt{number} \in T$ est un type
- si τ_1 et τ_2 sont des types ($\tau_1 \in T$ et $\tau_2 \in T$), alors $(\tau_1 \to \tau_2) \in T$

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

Exemples de types

$\texttt{number}$, $(\texttt{number} \to \texttt{number})$, $(\texttt{number} \to (\texttt{number} \to \texttt{number}))$
$((\texttt{number} \to \texttt{number}) \to \texttt{number})$

Exemples de termes typés

- $2 : \texttt{number}$
- $x + 2 : \texttt{number}$ (si $x : \textit{number}$)
- $\lambda x.(x + 2) : \texttt{number} \to \texttt{number}$

Types

Ensemble T des types

défini inductivement par :

- $\texttt{number} \in T$ est un type
- si τ_1 et τ_2 sont des types ($\tau_1 \in T$ et $\tau_2 \in T$), alors $(\tau_1 \to \tau_2) \in T$

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

Exemples de types

$\texttt{number}$, $(\texttt{number} \to \texttt{number})$, $(\texttt{number} \to (\texttt{number} \to \texttt{number}))$
$((\texttt{number} \to \texttt{number}) \to \texttt{number})$

Exemples de termes typés

- $2 : \texttt{number}$
- $x + 2 : \texttt{number}$ (si $x : \textit{number}$)
- $\lambda x.(x + 2) : \texttt{number} \to \texttt{number}$

Types

Ensemble T des types

défini inductivement par :

- $\texttt{number} \in T$ est un type
- si τ_1 et τ_2 sont des types ($\tau_1 \in T$ et $\tau_2 \in T$), alors $(\tau_1 \to \tau_2) \in T$

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

Exemples de types

$\texttt{number}$, $(\texttt{number} \to \texttt{number})$, $(\texttt{number} \to (\texttt{number} \to \texttt{number}))$
$((\texttt{number} \to \texttt{number}) \to \texttt{number})$

Exemples de termes typés

- $2 : \texttt{number}$
- $x + 2 : \texttt{number}$ (si x : *number*)
- $\lambda x.(x + 2) : \texttt{number} \to \texttt{number}$

Règles de typage

Jugement de typage

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

où $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$

Exemples

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

Règles de typage générales du λ-calcul

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

$$\frac{\Gamma, x : \tau \vdash M : \tau'}{\Gamma \vdash \lambda x.M : \tau \to \tau'}$$

$$\frac{\Gamma \vdash M : \tau \to \tau' \qquad \Gamma \vdash N : \tau}{\Gamma \vdash (MN) : \tau'}$$

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

$$\frac{\Gamma \vdash M : \texttt{number} \qquad \Gamma \vdash N : \texttt{number}}{\Gamma \vdash M + N : \texttt{number}}$$

$$\frac{\Gamma \vdash M : \texttt{number} \qquad \Gamma \vdash N : \texttt{number}}{\Gamma \vdash M * N : \texttt{number}}$$

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

Exemple de typage

Intuitivement

$$\underbrace{\underbrace{(\lambda x.\ \underbrace{x}_{\texttt{number}}\ *\ \underbrace{3}_{\texttt{number}}\ )}_{\texttt{number}\to\texttt{number}}\ \underbrace{4}_{\texttt{number}}}_{\texttt{number}}$$

Plus formellement

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \texttt{number} \vdash x : \texttt{number}} \quad \dfrac{}{x : \texttt{number} \vdash 3 : \texttt{number}}}{x : \texttt{number} \vdash x * 3 : \texttt{number}}}{\vdash (\lambda x : x * 3) : \texttt{number} \to \texttt{number}} \quad \dfrac{}{\vdash 4 : \texttt{number}}}{\vdash (\lambda x : x * 3)\ 4 : \texttt{number}}$$

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