LIFAP5 – Aide-mémoire

Licence informatique UCBL – Printemps 2018–2019

1 Syntaxe du λ-calcul

1.1 Syntaxe de base

Soit Var un ensemble *infini* de variables. L'ensemble Λ des *termes* du λ-calcul est défini *inductivement*, c'est *le plus petit* ensemble qui contient :

- x si $x \in \text{Var}$
- $\lambda x.M$ si $M \in \Lambda$ et $x \in \text{Var}$
- $(M\ N)$ si $M \in \Lambda$ et $N \in \Lambda$

Exemples

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

1.2 Extensions

Arithmétique on ajoute

- $M + N$ si $M \in \Lambda$ et $N \in \Lambda$
- $M * N$ si $M \in \Lambda$ et $N \in \Lambda$
- n si $n \in \mathbb{N}$ qu'on appelle constantes

Constantes on ajoute

- $\texttt{const}\ x = M\ \texttt{in}\ N$, cette expression se réécrit en $(\lambda x.N)M$

Paires on ajoute

- $\langle M, N\rangle$ si $M \in \Lambda$ et $N \in \Lambda$
- π_1 et π_2

Type unité on ajoute la constante

- $\square$

1.3 Variables libres et liées

BV[1] est définie par *induction* :

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

FV[2] est définie par *induction* :

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

1. *BV* pour *bound variables*
2. *FV* pour *free variables*

Exemples

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

2 Réduction du λ-calcul

On écrit $M[x := P]$ pour le terme M dont toutes les occurrences de la variable x ont été remplacées par le terme P. ⚠ Attention à ne pas rendre libres des variables liées ou vice-versa ⚠

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

2.1 Substitution sans capture

Définition formelle

$$x[x := P] = P$$
$$y[x := P] = y \text{ si } x \neq y$$
$$(MN)[x := P] = (M[x := P])(N[x := P])$$
$$(\lambda x.M)[x := P] = \lambda x.M$$
$$(\lambda y.M)[x := P] = \lambda y.(M[x := P]) \text{ si } x \neq y \text{ et } y \notin FV(P)$$

Exemples

$$(\lambda x.xy)[y := (\lambda z.z)] = \lambda x.x(\lambda z.z)$$
$$(\lambda x.xz)[y := (\lambda u.u)] = (\lambda x.xz)$$
$$(\lambda x.xy)[y := (\lambda x.x)] = (\lambda x.x(\lambda x.x)))$$
$$(\lambda x.xy)[y := (\lambda z.zx)] = \text{⚠ non défini ⚠}$$

2.2 α-conversion

Soit $\equiv_\alpha$ la relation de congruence dite « α-*conversion* » obtenue à partir de la relation :

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

Convention de Barendregt « *il n'existe aucun sous-terme dans lequel une variable apparaît à la fois libre et liée.* »
On peut toujours réécrire un terme pour qu'il respecte la convention de Barendregt en utilisant l'α-conversion avec une variable *fraîche*.

2.3 β-réduction

Le coeur du λ-calcul est la *réduction* des termes de la forme $(\lambda x.M)N$ qu'on appelle *redex*[3].

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

Exemples

$$\begin{aligned} & \lambda x.(x\ x)(\lambda y.y) \\ \underset{\beta}{\rightarrow} & (\lambda y.y)(\lambda y.y) \\ \equiv_\alpha & (\lambda z.z)(\lambda y.y) \\ \underset{\beta}{\rightarrow} & (\lambda y.y) \\ & (\lambda x.x(\lambda x.x))y \\ \equiv_\alpha & (\lambda x.x(\lambda z.z))y \\ \underset{\beta}{\rightarrow} & y(\lambda z.z) \end{aligned}$$

β-réduction de l'arithmétique On ajoute implicitement les β-réductions du calcul de l'arithmétique, sans les définir formellement. Par exemple $7 + 3 \underset{\beta}{\rightarrow} 10$.

β-réduction des constantes La réduction ne fait que réécrire[4] :

$$\texttt{const}\ x = M\ \texttt{in}\ N \underset{\beta}{\rightarrow} (\lambda x.N)M$$

β-réduction des projections de paires

$$\pi_1\langle M, N\rangle \underset{\beta}{\rightarrow} M$$
$$\pi_2\langle M, N\rangle \underset{\beta}{\rightarrow} N$$

2.4 Stratégie d'évaluation

Stratégie : décider de l'ordre dans lequel réduire les sous-termes de la forme $(\lambda x.M)N$

left-most outer-most Dite aussi stratégie *paresseuse* (*lazy*) ou *normale* : n'évalue que si nécessaire ; peut conduire à des doublons dans l'évaluation ; termine si il est possible de terminer.

Exemples

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

right-most inner-most Dite aussi stratégie *avide* (*eager*) ou *applicative* : elle 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.

3. pour *REDucible EXpression*.
4. On dit que c'est *du sucre syntaxique*

Exemples

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

3 Typage du λ-calcul

3.1 Syntaxe

Soit VarType un ensemble infini de variables de type. L'ensemble T des types est défini inductivement :

- $\texttt{number} \in T$
- $\alpha \in T$ si $\alpha \in$ VarType
- $(\tau_1 \rightarrow \tau_2) \in T$ si $\tau_1 \in T$ et $\tau_2 \in T$
- $(\tau_1 \times \tau_2) \in T$ si $\tau_1 \in T$ et $\tau_2 \in T$
- $\texttt{unit} \in T$

3.2 Règles de typage

Un *jugement de typage* est de la forme $\Gamma \vdash M : \tau$ avec $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ un ensemble de paires $x_i : \tau_i$ faites d'une variable x_i et d'un type τ_i.

Règles de typage générales

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

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

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

Règles de typage de l'arithmétique

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

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

$$\frac{}{\Gamma \vdash \texttt{n} : \texttt{number}}\ \text{si } n \in \mathbb{N}$$

Règles de typage des paires

$$\frac{\Gamma \vdash M : \tau_M \quad \Gamma \vdash N : \tau_N}{\Gamma \vdash \langle M, N\rangle : \tau_M \times \tau_N}$$

$$\frac{\Gamma \vdash p : \tau_M \times \tau_N}{\Gamma \vdash \pi_1\ p : \tau_M}$$

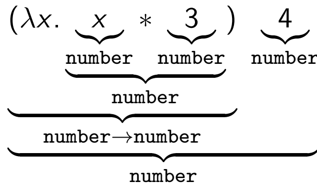
$$\frac{\Gamma \vdash p : \tau_M \times \tau_N}{\Gamma \vdash \pi_2\ p : \tau_N}$$

Typage de l'unité

$$\frac{}{\Gamma \vdash \square : \texttt{unit}}$$

3.3 Exemples

Intuitivement, on part des sous-termes *faciles* puis on étend à l'expression complète



Formellement, on utilise les règles de typage pour produire un arbre (on abrège « `number` » en « `n` »)

$$\dfrac{\dfrac{\dfrac{\overline{x:\mathtt{n}\vdash x:\mathtt{n}} \quad \overline{x:\mathtt{n}\vdash 3:\mathtt{n}}}{x:\mathtt{n}\vdash x*3:\mathtt{n}}}{\vdash(\lambda x:x*3):\mathtt{n}\to\mathtt{n}} \qquad \overline{\vdash 4:\mathtt{n}}}{\vdash(\lambda x:x*3)\ 4:\mathtt{n}}$$

Voir la figure 1 pour une dérivation du type de $\lambda x.\lambda y.x$ et la figure 2 celle de $\lambda x.\lambda y.\lambda z.(x\ z)(y\ z)$.

3.4 Propriétés du typage

L'adage « *well-typed expressions do not go wrong* » encapsule les deux résultats suivants.

Theorem 1 (Les réductions préservent les types). *Si $M \underset{\beta}{\to} N$, alors pour tout type τ tel que $\emptyset \vdash M : \tau$, on a également $\emptyset \vdash N : \tau$.*

Theorem 2 (Les réductions progressent). *Si M est une expression bien typée et sans variable libre, alors M est soit réductible soit une valeur*[5] *: si $\emptyset \vdash M : \tau$ avec $\mathrm{BV}(M) = \emptyset$, alors soit il existe N tel que $M \underset{\beta}{\to} N$ soit M est une valeur.*

4 λ-calcul vers `javascript`

On désigne par $\leadsto$ la transformation (syntaxique) qui permet de passer représenter un terme du λ-calcul en `javascript` avec les *fat arrows*.

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

Exemples

$$\lambda x.x \leadsto \texttt{x => x}$$
$$\lambda x.\lambda y.(xy) \leadsto \texttt{x => y => x(y)}$$
$$(\lambda x.\lambda y.(xy))(\lambda z.z) \leadsto \texttt{(x => y => x(y))(z => z)}$$

5 Curryfication

Curryfier c'est transformer une fonction avec un argument de type *paire* en une fonction à un argument qui renvoie une fonction à un argument. *Décurryfier* est l'opération inverse.

Définitions

$$\text{curry} : (\tau_1 \times \tau_2 \to \tau_3) \to (\tau_1 \to (\tau_2 \to \tau_3))$$
$$\text{curry} = \lambda f.\lambda x.\lambda y.f\ \langle x, y\rangle$$
$$\text{uncurry} : (\tau_1 \to (\tau_2 \to \tau_3)) \to (\tau_1 \times \tau_2 \to \tau_3)$$
$$\text{uncurry} = \lambda f.\lambda p.f\ (\pi_1\ p)\ (\pi_2\ p)$$

6 JavaScript

6.1 Déclaration

`var x` portée = *fonction englobante*

`let y` portée = *bloc*

`const c` constante, portée = *bloc*

6.2 Fermetures

> *Une fermeture, ou* closure *en anglais, est une fonction qui fait utiliser des variables indépendantes (utilisées localement mais définies dans la portée englobante). Autrement dit, ces fonctions se « souviennent » de l'environnement dans lequel elles ont été créées (on dit aussi que la fonction capture son « environnement »). Source MDN.*

```
function build(x) {
  return function(y){
    console.log(x + ": " + y);
  };
}
let f = build("A");
let g = build("B");
f("1");
f("2");
g("2");

// Affiche :
// A: 1
// A: 2
// B: 2
```

```
// avec IIFE (Immediately Invoked
   Function Expression)

let add = (function () {
    let counter = 0;
    return () => (counter++);
})();

add();  //0
add();  //1
add();  //2
```

6.3 API `Array`

```
const l = [1 , 5 , 10 , 15];

const d = l.map(x  => x * 2);
// d vaut [2 , 10 , 20 , 30]
const f = d.filter(x => x > 15) ;
```

5. C'est-à-dire soit une expression qui commence par un λ, soit une constante (mais pas une application ou une variable).

$$\cfrac{\cfrac{\overline{x:\tau_1, y:\tau_2 \vdash x:\tau_1}}{x:\tau_1 \vdash \lambda y.x : \tau_2 \to \tau_1}}{\vdash \lambda x.\lambda y.x : \tau_1 \to (\tau_2 \to \tau_1)}$$

FIGURE 1 – Typage de $\lambda x.\lambda y.x$

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\overline{\Gamma \vdash x:\tau_1 \to (\tau_2 \to \tau_3)} \quad \overline{\Gamma \vdash z:\tau_1}}{\Gamma \vdash (x\ z):(\tau_2 \to \tau_3)} \qquad \cfrac{\overline{\Gamma \vdash y:\tau_1 \to \tau_2} \quad \overline{\Gamma \vdash z:\tau_1}}{\Gamma \vdash (y\ z):\tau_2}}{\Gamma = x:\tau_1 \to (\tau_2 \to \tau_3), y:\tau_1 \to \tau_2, z:\tau_1 \vdash (x\ z)(y\ z):\tau_3}}{x:\tau_1 \to (\tau_2 \to \tau_3), y:\tau_1 \to \tau_2 \vdash \lambda z.(x\ z)(y\ z):(\tau_1 \to \tau_3)}}{\lambda x:\tau_1 \to (\tau_2 \to \tau_3) \vdash \lambda y.\lambda z.(x\ z)(y\ z):(\tau_1 \to \tau_2) \to (\tau_1 \to \tau_3)}}{\vdash \lambda x.\lambda y.\lambda z.(x\ z)(y\ z):(\tau_1 \to (\tau_2 \to \tau_3)) \to ((\tau_1 \to \tau_2) \to (\tau_1 \to \tau_3))}$$

FIGURE 2 – Typage de $\lambda x.\lambda y.\lambda z.(x\ z)(y\ z)$

```
// f vaut [20 , 30]
const s = f.reduce((a, c) => a + c, 0) ;
// s vaut 50

const r = l
  .map (x  => x * 2)
  .filter(x => x > 15)
  .reduce ((acc, cur) => acc + cur, 0) ;

s === r; // true

l.every(x => x % 5 === 0);
// renvoie false
l.some(x => x % 5 === 0);
// renvoie  true
```

6.4 API Promise

Une promesse est un objet (Promise) qui représente la complétion ou l'échec d'une opération asynchrone. La plupart du temps, on « consomme » des promesses et c'est donc ce que nous verrons dans la première partie de ce guide pour ensuite expliquer comment les créer.

En résumé, une promesse est un objet qui est renvoyé et auquel on attache des callbacks plutôt que de passer des callbacks à une fonction. *Source MDN*.

```
let PA = new Promise(
  (resolve, reject) => setTimeout(() =>
     resolve("Error 1"), 0));

PA
  .then(v => Promise.reject(v))
  // rejet
  .then(console.log)
  // pas exécuté
  .catch(e => console.error("Catch " + e
     ))
  // catch
  .then(() => console.log("OK"));
  // executé
```

```
// affiche
// Catch Error
// OK
```

```
const inc = x => Promise.resolve(x + 1)
const dec = x => Promise.resolve(x - 1)

// les deux sont considérés équivalents
let P1 = Promise.resolve(1)
  .then(inc)
  .then(dec);

let P2 = Promise.resolve(1)
  .then(x => inc(x)
    .then(y => dec(y))
  );

P1.then(console.log);
// affiche 1
P2.then(console.log);
// affiche 1
```

⚠ *Attention* ⚠

```
let P = Promise.resolve(1);

console.log(P);
// affiche
// Promise { <state>: "fulfilled", <
   value>: 1 }

P.then(console.log);
// affiche
// 1

console.log(P.then(console.log));
// affiche
// Promise { <state>: "pending" }
// 1
```