

# LIFAP5 – Programmation fonctionnelle pour le WEB

## TD2 – programmation fonctionnelle en JavaScript

Licence informatique UCBL – Printemps 2021–2022

### Exercice 1 : Curryfication et application partielle

Dans cet exercice on considère le  $\lambda$ -calcul étendu aux paires. La paire des termes  $M$  et  $N$  est notée  $\langle M, N \rangle$ . Si  $M : \tau_M$ , lu «  $M$  est de type  $\tau_M$  » et  $N : \tau_N$ , alors  $\langle M, N \rangle : \tau_M \times \tau_N$ . Les fonctions  $\pi_1$  et  $\pi_2$  sont les projections associées aux paires. Les projections se  $\beta$ -réduisant respectivement en  $\pi_1 \langle M, N \rangle \xrightarrow{\beta} M$  et  $\pi_2 \langle M, N \rangle \xrightarrow{\beta} N$ .

1. Donner le type puis définir une fonction `c_add` qui prend un nombre en argument et renvoie une fonction qui à son tour prend un autre nombre en argument et renvoie la somme des deux.
2. Donner les types des projections  $\pi_1$  et  $\pi_2$ .
3. Donner le type puis définir une fonction `curry2` qui prend en argument une fonction `f` à deux arguments et renvoie sa version curryfiée, c'est-à-dire la fonction qui prend le premier argument et renvoie une fonction qui prend le deuxième argument et renvoie le résultat de `f` appliquée aux deux arguments.
4. On définit `let plus = (x,y) => x + y`; . Utiliser `curry2` pour proposer une nouvelle définition de `c_add`.
5. Soit la définition de `let combine = f => f(2)*f(3)`; . Calculer `combine(c_add(5))`.
6. Simplifier en la  $\beta$ -réduisant l'expression `let f = x => combine(c_add(x))`;
7. Donner le type puis écrire une fonction `uncurry2` en JavaScript qui soit l'inverse de `curry2`, c'est-à-dire qui vérifie `uncurry2(curry2(f)) = f` et `curry2(uncurry2(f)) = f`.
8. Traduire les définitions de `uncurry2` et de `curry2` en  $\lambda$ -calcul (étendu aux paires). Ces termes seront nommés **uncurry** et **curry**.
9. On rappelle la définition de  $\mathbf{K} = \lambda x. \lambda y. x$ . Prouver que le curryfié de  $\pi_1$  est  $\mathbf{K}$ . Quelle expression du  $\lambda$ -calcul correspond au curryfié de  $\pi_2$ .
10. Prouver les propriétés qui unissent `curry2` et `uncurry2` de la question 7 en  $\beta$ -réduisant `uncurry(curry(F))\langle X, Y \rangle` et `curry(uncurry(F)) X Y` pour des termes  $F, X$  et  $Y$  donnés en paramètres.
11. Quel est l'intérêt, du point de vue de la performance, du résultat précédent ?

## Exercice 2 : Curryfication appliquée à `map()`

On rappelle que la méthode `Array.prototype.map(f)` crée un nouveau tableau composé des images des éléments du tableau appelant par la fonction `f` donnée en argument. Pour un tableau `A = [v_0, v_1, ..., v_n]`, `A.map(f)` renvoi `[f(v_0), f(v_1), ..., f(v_n)]` et laisse `A` inchangé.

1. Donner le type puis écrire une fonction `c_map` qui prend en argument une fonction `f` et renvoie une fonction qui prend un tableau `t` et applique `f` à chaque élément de `t`, c'est-à-dire une version curryfiée de `Array.prototype.map(f)`.
2. Utiliser `c_add` et `c_map` pour définir une fonction qui ajoute 3 à tous les éléments d'un tableau sans utiliser `function` et `=>`.
3. Soit un tableau `t`. Expliquer ce que calcule la fonction `c_map(f => f(3))(t)`
4. Soit `A` un tableau qui contienne des objets représentant des personnes avec les champs `nom`, `prénom` et `age`. Définir une fonction qui transforme `A` en un tableau ne comportant que les ages. Le faire également en utilisant la fonction `let prj = c => o => o[c];`.

## Corrections

### Solution de l'exercice 1

Objectif : comprendre ce qu'est une fonction curryfiée et pourquoi on peut vouloir écrire de telles fonctions ; montrer que le  $\lambda$ -calcul permet de raisonner sur les programmes purs.

1. Le type est  $c\_add : \text{number} \rightarrow \text{number} \rightarrow \text{number}$ .

```
1 let c_add = (n1 => n2 => n1 + n2);
2
3 // ou équivalent (à la gestion du this près)
4 function c_add(x){
5   return function(y){
6     return (x+y);
7   }
8 }
```

2.  $\pi_1 : \tau_M \times \tau_N \rightarrow \tau_M$  et  $\pi_2 : \tau_M \times \tau_N \rightarrow \tau_N$
3. Autrement dit, on souhaite que  $\text{curry2}(f)(x)(y) = f(x,y)$  et tout est dit. Le type est  $\text{curry2} : (\alpha \times \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta \rightarrow \gamma)$ .

```
1 let curry2 = f => x => y => f(x, y);
2
3 // ou équivalent (à la gestion du this près)
4 function curry2(f){
5   return function(x){
6     return function(y){
7       return f(x,y);
8     }
9   }
10 }
```

4. Simplement, on curryfie **plus**

```
1 let c_add2 = curry2(plus);
```

5. On va montre qu'on peut facilement faire des applications partielles une fois une fonction curryfiée.

```
1   combine(c_add(5))
2 = (f => f(2)* f(3))(c_add(5))
3 = c_add(5)(2)* c_add(5)(3)
4 = (5 + 2) * (5 + 3)
5 = 7 * 8
6 = 56
```

6. On va faire les réductions suivantes en  $\lambda$ -calcul :

$$\begin{aligned} f &\stackrel{\text{def}}{=} \lambda x. \text{combine}(c\_add\ x) \\ &\stackrel{\text{def}}{=} \lambda x. \text{combine}((\lambda y. \lambda y. x + y)\ x) \\ &\stackrel{\text{def}}{=} \lambda x. (\lambda f. (f\ 2) \times (f\ 3))((\lambda y. \lambda y. x + y)\ x) \\ &\rightsquigarrow^x \lambda x. (\lambda f. (f\ 2) \times (f\ 3))(\lambda y. x + y) \\ &\rightsquigarrow^f \lambda x. ((\lambda y. x + y)\ 2) \times ((\lambda y. x + y)\ 3) \\ &\rightsquigarrow^y \lambda x. (x + 2) \times (x + 3) \end{aligned}$$

On pourrait donc de façon équivalente définir `let f = x => (x + 2) * (x + 3);`

7. Le type est `uncurry2 : (α → β → γ) → (α × β → γ)`.

1 `let uncurry2 = f => (x, y) => f(x)(y);`

8. On définit `curry = λf.λx.λy.f⟨x, y⟩` et `uncurry = λf.λp.f(π1p)(π2p)` avec π<sub>i</sub> les projections de la paire.

9.

$$\begin{aligned} (\text{curry } \pi_1) &\stackrel{\text{def}}{=} (\lambda f. \lambda x. \lambda y. f \langle x, y \rangle) \pi_1 \\ &\stackrel{f}{\rightsquigarrow} \lambda x. \lambda y. \pi_1 \langle x, y \rangle \\ &\stackrel{\pi_1}{\rightsquigarrow} \lambda x. \lambda y. x \\ &\stackrel{\text{def}}{=} \mathbf{K} \end{aligned}$$

Le curryfié de π<sub>2</sub> est λx.λy.y

10. Pour la preuve on β-réduit simplement les expressions et on utilise les règles π<sub>1</sub>⟨X, Y⟩ = X et π<sub>2</sub>⟨X, Y⟩ = Y. On conclut par η-équivalence que `uncurry(curry(F)) = F`.

$$\begin{aligned} \text{uncurry}(\text{curry}(F)) \langle X, Y \rangle &\stackrel{\text{def}}{=} (\lambda f. \lambda p. f(\pi_1 p)(\pi_2 p)) ((\lambda f. \lambda x. \lambda y. f \langle x, y \rangle) F) \langle X, Y \rangle \\ &\stackrel{f}{\rightsquigarrow} (\lambda f. \lambda p. f(\pi_1 p)(\pi_2 p)) (\lambda x. \lambda y. F \langle x, y \rangle) \langle X, Y \rangle \\ &\stackrel{f}{\rightsquigarrow} (\lambda p. (\lambda x. \lambda y. F \langle x, y \rangle) (\pi_1 p)(\pi_2 p)) \langle X, Y \rangle \\ &\stackrel{p}{\rightsquigarrow} (\lambda x. \lambda y. F \langle x, y \rangle) (\pi_1 \langle X, Y \rangle) (\pi_2 \langle X, Y \rangle) \\ &\stackrel{x,y}{\rightsquigarrow} F \langle (\pi_1 \langle X, Y \rangle), (\pi_2 \langle X, Y \rangle) \rangle \\ &\stackrel{\pi_1}{\rightsquigarrow} F \langle X, (\pi_2 \langle X, Y \rangle) \rangle \\ &\stackrel{\pi_2}{\rightsquigarrow} F \langle X, Y \rangle \end{aligned}$$

Pour l'autre sens, on conclut par deux η-équivalences que `curry(uncurry(F)) = F`.

$$\begin{aligned} \text{curry}(\text{uncurry}(F)) X Y &\stackrel{\text{def}}{=} (\lambda f. \lambda x. \lambda y. f \langle x, y \rangle) ((\lambda f. \lambda p. f(\pi_1 p)(\pi_2 p)) F) X Y \\ &\stackrel{f}{\rightsquigarrow} (\lambda f. \lambda x. \lambda y. f \langle x, y \rangle) (\lambda p. F(\pi_1 p)(\pi_2 p)) X Y \\ &\stackrel{f}{\rightsquigarrow} (\lambda x. \lambda y. (\lambda p. F(\pi_1 p)(\pi_2 p)) \langle x, y \rangle) X Y \\ &\stackrel{x,y}{\rightsquigarrow} (\lambda p. F(\pi_1 p)(\pi_2 p)) \langle X, Y \rangle \\ &\stackrel{p}{\rightsquigarrow} F(\pi_1 \langle X, Y \rangle) (\pi_2 \langle X, Y \rangle) \\ &\stackrel{\pi_1, \pi_2}{\rightsquigarrow} F(X)(Y) \end{aligned}$$

On peut ainsi conclure sur l'isomorphisme entre  $X \times Y \rightarrow Z$  et  $X \rightarrow (Y \rightarrow Z)$

11. On a montré (à la η-équivalence près que l'on passe un peu sous silence) que `uncurry ∘ curry` et `curry ∘ uncurry` sont équivalentes à l'identité λx.x.

On peut donc se demander s'il y a intérêt à privilégier l'une ou l'autre de ces formes de ces équivalences quand on programme. Cela va dépendre du compilateur/interpréteur :

— pour ceux, comme Haskell, qui simplifient les expressions une bonne fois pour toute cela ne change quasiment rien ;

— par contre, pour ceux qui réévaluent systématiquement toute l'expression, il y a plusieurs créations de fonctions intermédiaires et d'appels qui sont éliminés avec l'expression β-réduite.

Comparons empiriquement les performances des fonctions évaluées par NodeJS comme ci-dessous, on va observer une différence significative d'un facteur  $\approx 45$ .

```
1 // npm install microtime benchmark
2 // see https://github.com/bestiejs/benchmark.js
3
4 let Benchmark = require('benchmark');
5 let suite1 = new Benchmark.Suite;
6 let suite2 = new Benchmark.Suite;
7 let suite3 = new Benchmark.Suite;
8
9 let curry2 = f => x => y => f(x,y);
10 let uncurry2 = g => (x,y) => g(x)(y);
11 let c_add = x => y => x + y;
12 let plus = (x, y) => x + y;
13 let a = 1023;
14 let b = 65535;
15
16 suite1
17 .add('id1#full', () => uncurry2(curry2(plus))(a,b))
18 .add('id1#red ', () => plus(a,b))
19 .on('cycle', (event) => console.log(String(event.target)))
20 .run({ 'async': true });
21 suite2
22 .add('id2#full', () => curry2(uncurry2(c_add))(a)(b))
23 .add('id2#red ', () => c_add(a)(b))
24 .on('cycle', (event) => console.log(String(event.target)))
25 .run({ 'async': true });
26
27 // Intel(R) Core(TM) i7-5600U CPU @ 2.60GHz (2 cores * 2 threads)
28 // NodeJS v12.15.0
29 // id1#full x 17,815,241 ops/sec +/-2.40% (82 runs sampled)
30 // id2#full x 18,229,881 ops/sec +/-2.37% (85 runs sampled)
31 // id1#red x 828,225,214 ops/sec +/-1.50% (82 runs sampled)
32 // id2#red x 811,819,712 ops/sec +/-1.63% (75 runs sampled)
```

## Solution de l'exercice 2

*Objectif : pratiquer la currification sur des cas à peu près concret en JavaScript. Apprécier l'intérêt des  $\lambda$  pour le passage de paramètres (quand le nom n'est pas important).*

1. Le type est  $c\_map : (\alpha \rightarrow \beta) \rightarrow \text{Array}(\alpha) \rightarrow \text{Array}(\beta)$ . Il suffit d'utiliser la méthode `map` des tableaux pour l'implémentation :

```
1 let c_map = (f => t => t.map(f)) ;
```

2. Deux application partielles de fonctions :

```
1 let add3 = c_map(c_add(3));
```

3. Cette fonction prend un tableau qui contient des fonctions et applique chacune à la valeur 3. Par exemple, avec `t = [x => x+1, x => x*2, x => x*x]` on obtient `[ 4, 6, 9 ]`.
4. On définit directement `A.map( o => o.age)` ou `A.map(prj("age"))`.