LIFAP5 – Programmation fonctionnelle pour le WEB
TD2 – programmation fonctionnelle en JavaScript

Licence informatique UCBL – Printemps 2020–2021

Exercice 1 : Curryfication et application partielle

Dans cet exercice on considère le λ-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 τ_N » et $N : \tau_N$, alors $\langle M, N\rangle : \tau_M \times \tau_N$. Les fonctions π_1 et π_2 sont les projections associées aux paires. Les projections se β-réduisant respectivement en $\pi_1 \langle M, N\rangle \underset{\beta}{\rightarrow} M$ et $\pi_2 \langle M, N\rangle \underset{\beta}{\rightarrow} N$.

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 π_1 et π_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 β-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 λ-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 π_1 est $\mathbf{K}$. Quelle expression du λ-calcul correspond au curryfié de π_2.
10. Prouver les propriétés qui unissent `curry2` et `uncurry2` de la question 7 en β-réduisant $\mathbf{uncurry}(\mathbf{curry}(F))\langle X, Y\rangle$ et $\mathbf{curry}(\mathbf{uncurry}(F))\ X\ Y$ 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 λ-calcul permet de raisonner sur les programmes purs.

1. Le type est `c_add` : `number` $\to$ `number` $\to$ `number`.

```
let c_add = (n1 => n2 => n1 + n2);

// ou équivalent (à la gestion du this près)
function c_add(x){
  return function(y){
    return (x+y);
  }
}
```

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

```
let curry2 = f => x => y => f(x, y);

// ou équivalent (à la gestion du this près)
function curry2(f){
  return function(x){
    return function(y){
      return f(x,y);
    }
  }
}
```

4. Simplement, on curryfie `plus`

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

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

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

6. On va faire les réductions suivantes en λ-calcul :

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

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

7. Le type est `uncurry2` : $(\alpha \to \beta \to \gamma) \to (\alpha \times \beta \to \gamma)$.

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

8. On définit **curry** $= \lambda f.\lambda x.\lambda y.f\langle x, y\rangle$ et **uncurry** $= \lambda f.\lambda p.f(\pi_1 p)(\pi_2 p)$ avec π_i les projections de la paire.
9. $$\begin{aligned}(\textbf{curry}\ \pi_1) &\overset{def}{=} (\lambda f.\lambda x.\lambda y.f\langle x, y\rangle)\pi_1\\ &\overset{f}{\rightsquigarrow} \lambda x.\lambda y.\pi_1\langle x, y\rangle\\ &\overset{\pi_1}{\rightsquigarrow} \lambda x.\lambda y.x\\ &\overset{def}{=} \mathbf{K}\end{aligned}$$

   Le curryfié de π_2 est $\lambda x.\lambda y.y$
10. Pour la preuve on β-réduit simplement les expressions et on utiliser les règles $\pi_1\langle X, Y\rangle = X$ et $\pi_2\langle X, Y\rangle = Y$ On conclut par η-équivalence que **uncurry**(**curry**$(F)) = F$.

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

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

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

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

11. On a montré (à la η-équivalence près que l'on passe un peu sous silence) que **uncurry** $\circ$ **curry** et **curry** $\circ$ **uncurry** sont équivalentes à l'identité $\lambda x.x$.
    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 ≈ 45.

```
// npm install microtime benchmark
// see https://github.com/bestiejs/benchmark.js

let Benchmark = require('benchmark');
let suite1 = new Benchmark.Suite;
let suite2 = new Benchmark.Suite;
let suite3 = new Benchmark.Suite;

let curry2 = f => x => y => f(x,y);
let uncurry2 = g => (x,y) => g(x)(y);
let c_add = x => y => x + y;
let plus = (x, y) => x + y;
let a = 1023;
let b = 65535;

suite1
.add('id1#full', () => uncurry2(curry2(plus))(a,b))
.add('id1#red ', () => plus(a,b))
.on('cycle', (event) => console.log(String(event.target)))
.run({ 'async': true });
suite2
.add('id2#full', () => curry2(uncurry2(c_add))(a)(b))
.add('id2#red ', () => c_add(a)(b))
.on('cycle', (event) => console.log(String(event.target)))
.run({ 'async': true });

// Intel(R) Core(TM) i7-5600U CPU @ 2.60GHz (2 cores * 2 threads)
// NodeJS v12.15.0
// id1#full x  17,815,241 ops/sec +/-2.40% (82 runs sampled)
// id2#full x  18,229,881 ops/sec +/-2.37% (85 runs sampled)
// id1#red  x 828,225,214 ops/sec +/-1.50% (82 runs sampled)
// 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 λ pour le passage de paramètres (quand le nom n'est pas important).

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

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

2. Deux application partielles de fonctions :

```
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"))`.