Curryfication

En programmation fonctionnelle, la **curryfication** désigne la transformation d'une fonction à plusieurs arguments en une fonction à un argument qui retourne une fonction sur le reste des arguments. La **curryfication** permet de créer des fonctions pures. L'opération inverse est possible et s'appelle la **décurryfication**.

Le terme vient du nom du mathématicien américain Haskell Curry, bien que cette opération ait été introduite pour la première fois par Moses Schönfinkel[1].

Sommaire

Définition

Considérons une fonction `add(x, y)` qui prend deux arguments et en renvoie la somme. Une fois curryfiée, on aurait une fonction `add(x)` qui prend un argument et renvoie une fonction qui prend le deuxième argument. En pseudo-langage :

```
curry (add (x,y)) → add x → lambda (y → x + y)
```

La **curryfication** permet l'**application partielle** : si on appelle la fonction curryfiée avec l'argument 3, on récupère une fonction qui ajoute 3 à son argument.

Exemples

Haskell

En Haskell voici une fonction non-curryfiée :

```
uncurried_add (x, y) = x + y
```

et la même après **curryfication**:

```
curried_add x y = x + y
```

ou (la barre \ signifie lambda et sert pour définir des fonctions anonymes) :

```
curried_add = \x -> \y -> x + y
```

Et voici une application partielle :

```
add_three = curried_add 3
```

Remarques

La fonction *add_three* est dite pure.

La **curryfication** peut se faire à la main ou bien par un programme. Pour les fonctions à deux arguments, voici ces programmes en Haskell :

```
curry f x y = f(x,y)
uncurry f (x, y) = f x y
```

Python

Même fonction en Python :

```
def uncurried_add (x, y):
    return x + y
def curried_add (x):
    return lambda y: x + y
print uncurried_add(1, 4)
print curried_add(1)(4)
```

Caml

Même fonction en Caml :

```
let uncurried_add (x,y) = x + y;; (* Type de la fonction : (entier * entier) -> entier *)
let curried_add x y = x + y;; (* Type de la fonction : entier -> entier -> entier *)

uncurried_add(3,4);; (* Retourne 7. Type de la valeur retournée : entier. *)

curried_add 3;; (* Retourne la fonction qui ajoute 3. Type de la valeur retournée : entier -> entier. *)
(curried_add 3) 4;; (* Crée la fonction qui ajoute 3 et l'applique à l'entier 4. Retourne 7. Type de la valeur retournée : entier. *)

let add_three = curried_add 3;; (* Définit la fonction add_three comme la fonction qui ajoute 3 à son argument. *)
add_three 4;; (* Applique add_three à 4. Retourne 7. *)
```

Ruby

Dans la version 1.9 de Ruby, la classe Proc fournit à ses instances la méthode curry. Proc#curry retourne un objet proc curryfié. La méthode prend en argument un argument facultatif : l'arité qui permet des processus plus fins que nous ne détaillerons pas ici.

```
p = Proc.new { |a,b| a+b } => #<Proc:0x00000000ff5208>

#Appel classique
p.call(3) => "TypeError: nil can't be coerced into Fixnum"
p.call(3,4) => 7

#Utilisation de la méthode curry
p.curry[3][4] => 7
#renvoie un Fixnum, car il y a eu le bon nombre d'arguments fournis.

a = p.curry[3] => #<Proc:0x00000000f90d68>
a.call(5) => 8
a.call(10) => 13
```

Scheme

```
(define uncurried-add (lambda (x y)
                        (+ x y)))
(define curried_add (lambda (x) (lambda (y) (+ x y))))
(define add-three (curried_add 3))
```

Scala

```
  def uncurried_add(x : Int, y : Int) = x + y         // |-> Int
  // Après currying :
  def curried_add0(x : Int)(y : Int) = x + y          // |-> Int
  // Ce qui équivaut (si on décompose) à :
  def curried_add1 (x : Int) = (y : Int) =>  x + y    // |-> Int => Int
  // Ou encore à :
  def curried_add2 = (x : Int) => (y : Int) =>  x + y // |-> Int => (Int => Int)

// applications partielles :

  def add_three0 = curried_add0(3)_ // retourne une fonction de type : Int => Int
  curried_add0(3)(5) // Retourne 8
  add_three0(5) // Retourne 8

  def add_three1 = curried_add1(3) // retourne une fonction de type : Int => Int
  curried_add1(3)(5) // Retourne 8
  add_three1(5) // Retourne 8

  def add_three2 = curried_add2(3) // retourne une fonction de type : Int => Int
  curried_add2(3)(5) // Retourne 8
  add_three2(5) // Retourne 8
```

Tcl

```
proc uncurried_add {x y} {expr {$x + $y}}
proc curried_add x {list apply {list y {format {expr {%d + $y}} $x}}}
```

Perl 6

Un JAPH qui est un exemple de **curryfication** en Perl 6 :

```
sub japh ($str $slang) { say "just another $slang hacker"; }
my &perl6japh := &japh.assuming('Perl6');
perl6japh();
```

JavaScript

```
function uncurriedAdd(x, y) {
  return x + y;
}
function curriedAdd(x) {
  return function(y) {
    return x + y;
  }
}

uncurriedAdd(2, 5); // 7
curriedAdd(2)(5); // 7
```

JavaScript ES2015

```
const uncurriedAdd = (x, y) => x + y;
const curriedAdd = x => y => x + y;

uncurriedAdd(2, 5); // 7
curriedAdd(2)(5); // 7
```

Clojure

```
(defn cur_add [x]
  (fn [y] (+ x y)) ;; fonction anonyme renvoyée par cur_add
)

((curr_add 2) 4) ;; 6
```

Modern C++

Le C++11 a introduit les lambdas dans le langage, puis le C++14 a introduit la possibilité de définir un type de retour **auto** pour faciliter l'écriture de certaines fonctions.

Ces deux améliorations permettent d'implémenter la curryfication :

```
#include <iostream>

auto uncurried_add(int a, int b){
  return a + b;
}

auto curried_add(int x){
  return [x](int y){return x + y;};
}

int main(int argc, char *argv[]){
  std::cout << uncurried_add(1, 4) << "\n";
  std::cout << curried_add(3)(4) << "\n";
}
```

À noter que depuis le C++17, la mention du type **int** en argument de l'expression lambda peut également être remplacée par **auto**, généralisant l'implémentation (au prix de faire de la méthode **curried_add** un template implicite)

Notes et références

1. (**en**) John C. Reynolds, « Definitional Interpreters for Higher-Order Programming Languages », *Higher-Order and Symbolic Computation*, vol. 11, n° 4, 1998, p. 374 (DOI 10.1023/A:1010027404223 (http://dx.doi.org/10.1023%2FA%3A1010027404223)) :

> « In the last line we have used a trick called Currying (after the logician H. Curry) to solve the problem of introducing a binary operation into a language where all functions must accept a single argument. (The referee comments that although "Currying" is tastier, "Schönfinkeling" might be more accurate.) »

Annexes

Articles connexes

- Haskell Curry
- Moses Schönfinkel

Sur les autres projets Wikimedia :

- *curryfication*, sur le Wiktionnaire
- *opérateur de diffusion*, sur le Wiktionnaire