À propos des continuations

Bernard Tatin

2017

Résumé. Ici, on s'occupe des continuations tout d'abord avec *Scheme* puis, si possible, avec d'autres langages dont *Standard ML* ou *F#*. Le fil conducteur provient, sauf indication contraire, des articles de *Wikipedia* en anglais ou en français qui concernent ces continuations de la programmation fonctionnelle.

Le choix de noweb provient du simple fait que documentation et sources sont conçus en même temps.[a]

[a] Document crée le 30/11/2017 à 00:10.

Contents

I

introduction

Figure 1: *la Loire, métaphore de la continuation?* (r)
Source: photo de l'auteur

Note. Ce qui suit provient pour l'essentiel de l'article de Wikipedia (en anglais): Continuation-passing style.

I.1

I.1 un premier test et quelques définitions

Commençons donc par les définitions essentielles:

Définition 1 - CPS. Le *continuation-passing style* ou **CPS** est un style de programmation où le contrôle est passé explicitement sous forme de continuation.

C'est ce style que nous allons présenter dans les pages qui suivent. En attendant, voyons ce qu'est une continuation :

Définition 2 - continuation. Une continuation d'un programme est *la suite des instructions qu'il lui reste à exécuter à un moment précis*[a]

[a] *Cf.* Continuation.

Voici un exemple montrant la différence entre le style direct:

4 ⟨*distance direct style* 4⟩≡

```
(define (distance x y)
 (sqrt (+ (* x x) (* y y))))
```

This code is used in chunk 6a.
Defines:
`distance`, used in chunks 5a and 6a.

Et le CPS:

5a ⟨*distance CPS* 5a⟩≡

```
(define (distance-cps x y k)
 (*-cps x x (lambda (x2)
          (*-cps y y (lambda (y2)
                   (+-cps x2 y2 (lambda (x2py2)
                              (sqrt-cps x2py2 k))))))))
```

This code is used in chunk 6a.
Defines:
`distance-cps`, used in chunk 6a.
Uses `distance` 4 and `sqrt-cps` 5b.

Pour la définition des fonctions utilisées en CPS :

5b ⟨*cps function definition* 5b⟩≡

```
(define (cps-prim f)
 (lambda args
  (let ((r (reverse args)))
   ((car r) (apply f
              (reverse (cdr r)))))))
(define *-cps (cps-prim *))
(define +-cps (cps-prim +))
(define sqrt-cps (cps-prim sqrt))
```

This code is used in chunk 6a.
Defines:
`*-cps,`, never used.
`+-cps,`, never used.
`cps-prim,`, never used.
`sqrt-cps`, used in chunk 5a.

Testons:

6a ⟨*first-cps-test.scm* 6a⟩≡

```
;; first-cps-test

⟨tools for scheme 10⟩
⟨cps function definition 5b⟩
⟨distance CPS 5a⟩
⟨distance direct style 4⟩

(define test
  (lambda(x y)
    (myprint "x=" x " y=" y)
    (myprint " direct=" (distance x y))
    (distance-cps x y (lambda(e) (myprint " cps=" e "\n")))))

(test 3 4)
(test 0 3)
(test 3 0)
```

Root chunk (not used in this document).
Uses `distance` 4, `distance-cps` 5a, and `myprint` 10.

Ce code doit nous renvoyer ce résultat:

6b ⟨*resultat first cps test* 6b⟩≡

```
$ gosh first-cps-test.scm
x=3 y=4 direct=5 cps=5
x=0 y=3 direct=3 cps=3
x=3 y=0 direct=3 cps=3
```

Root chunk (not used in this document).

I.2 du code plus intéressant

Notre continuation un peu simpliste permet de mieux appréhender l'essentiel du problème. Voyons ce qui peut-être plus constructif et utile comme les échappements.

Voici un exemple calculant le produit des éléments d'une liste en style direct:

7a ⟨*list product* 7a⟩≡

```
(define direct-product-1
  (lambda(L)
    (if (null? L)
      1
      (* (car L) (direct-product-1 (cdr L))))))
```

This definition is continued in chunks 7 and 8.
Root chunk (not used in this document).
Defines:
`direct-product-1`, never used.

On peut raccourcir les calculs si 0 est présent dans la liste:

7b ⟨*list product* 7a⟩+≡

```
(define direct-product-2
  (lambda(L)
    (cond
      ((null? L) 1)
      ((zero? (car L)) 0)
      (else (* (car L) (direct-product-2 (cdr L)))))))
```

Defines:
`direct-product-2`, never used.

Maintenant, passons en CPS[1]:

8a ⟨*list product* 7a⟩+≡
```
(define cps-product
  (lambda(L k)
    (cond
      ((null? L) (k 1))
      ((zero? (car L)) (k 0))
      (else (cps-product (cdr L) (lambda(p-cdr) (k (* p-cdr (car L))))))
      )
  ))
```
Defines:
`cps-product`, used in chunk 8.

L'utilisation de `cps-product` est en fait simple, il suffit d'utiliser la fonction identité:

8b ⟨*list product test* 8b⟩≡
```
(cps-product '(12 6 0 35 42) (lambda(n) n))
```
Root chunk (not used in this document).
Uses `cps-product` 8a.

Pour les continuations vérifiant:

$$k(0) = 0 \tag{1}$$

on peut définir un `cps-product-0`:

8c ⟨*list product* 7a⟩+≡
```
(define cps-product-0
  (lambda(L k)
    (cond
      ((null? L) (k 1))
      ((zero? (car L)) 0)
      (else (cps-product-0 (cdr L) (lambda(p-cdr) (k (* p-cdr (car L))))))
      )
  ))
```
Defines:
`cps-product-0`, never used.
Uses `cps-product` 8a.

[1] La démonstration dans le Chazarain

II annexes

II.1 fonctions utiles

II.1.1 affichage console

Voici un `display` plus *fun*:

10 ⟨*tools for scheme* 10⟩≡
```
(define (myprint . l)
  (for-each (lambda(e) (display e)) l))
```
This code is used in chunk 6a.
Defines:
`myprint`, used in chunk 6a.

III tables et index

III.1 table des extraits de code

III.2 index des symboles

III.3

III.3 Définitions