

## Cours No 7 : Optimisation de fonctions récursives

Notes de cours - 2007-2012

### 14 Optimisation de fonctions récursives

#### 14.1 Rappels sur itérations

Processus de calcul itératif : processus basé sur une suite de transformation de l'état de l'ordinateur spécifié par au moins une boucle et des affectations.

##### Affectation

Définition : instruction permettant de modifier la valeur stockée à l'emplacement mémoire associé au nom de la variable.

```
1 x := 12; ;; en Algol, Simula, Pascal (syntaxe classique)
2 x = 12; ;; en C, Java, C++
3 (set! x 12) ;; en scheme
4 (set! x (+ x 1))
```

##### Une structure de contrôle pour écrire les boucles en Scheme

```
1 (do (initialisation des variables)
2   (condition-d'arrêt expression-si-condition-d'arrêt-vérifiée)
3   (instruction 1)
4   ...
5   (instruction n))
```

#### 14.2 Equivalence itération - récursions terminales

Du point de vue de la quantité d'espace pile nécessaire pour leur interprétation, les deux fonctions suivantes sont équivalentes.

```
1 ;; fonction explicitement itérative
2 (define (ipgcd a b)
3   (do ((m (modulo a b)))
4     ((= m 0) b)
5     (set! a b)
6     (set! b m)
7     (set! m (modulo a b))))
```

```

8
9 ;; fonction récursive terminale
10 (define (rpgcd a b)
11   (let ((m (modulo a b)))
12     (if (= m 0)
13         b
14         (rpgcd b m))))

```

### 14.3 Transformations

Note : Il n'est pas utile de vouloir à tout prix dérécursiver. La taille des mémoires et l'efficacité des processeurs rendent de nombreuses fonctions récursives opérantes.

Cependant certaines transformations sont simples et il n'est pas coûteux de les mettre en oeuvre. Les récursions à complexité exponentielles doivent être simplifiées quand c'est possible.

Pour dérécursiver, il y a deux grandes solutions : soit trouver une autre façon de poser le problème soit, dans le cas général, passer le calcul d'enveloppe.

### 14.4 Transformation des récursions enveloppées simples

Exemple de factorielle.

#### Une version itérative (en C)

```

1 int fact(n){
2   int cpt = 0;
3   int acc = 1;
4   while (cpt < n){
5     // acc = fact(cpt) -- invariant de boucle
6     cpt = cpt + 1;
7     acc = acc * cpt;
8     // acc = fact(cpt) -- invariant de boucle
9   }
10  // en sortie de boucle, cpt = n, acc = fact(n)
11  return(acc)
12 }

```

#### Une version itérative (en scheme)

```

1 (define (ifact n)
2   (do ((cpt 0) (acc 1))
3     ;; test d'arrêt et valeur rendue si vrai
4     ((= cpt n) acc)
5     ;; acc = fact(cpt) -- invariant de boucle
6     (set! cpt (+ cpt 1))
7     (set! acc (* acc cpt))
8     ;; acc = fact(cpt) -- invariant

```

```
9 ))
```

## Calcul d'enveloppe via les arguments

Solution générique à la dérécursivation : passer, lorsque c'est possible, le calcul d'enveloppe (ce qu'il y a à faire une fois que l'appel récursif est achevé) en argument de l'appel récursif.

Deux paramètres vont prendre au fil des appels récursifs les valeurs successives de `cpt` et `acc` de la version itérative.

```
1 (define (ifact n cpt acc)
2   ;; acc = fact(cpt)
3   (if (= cpt n)
4       acc
5       (ifact n (+ cpt 1) (* (+ cpt 1) acc))))
```

Il est nécessaire de réaliser l'appel initial correctement : `(ifact 7 0 1)`

Version plus élégante, qui évite le passage du paramètre `n` à chaque appel récursif et évite l'appel initial complexe.

```
1 (define (rtfact n)
2   (let boucle ((cpt 0) (acc 1))
3     ;; acc = fact(cpt)
4     (if (= cpt n)
5         acc
6         (boucle (+ cpt 1) (* (+ cpt 1) acc))))))
```

La multiplication étant associative, on peut effectuer les produits de `n` à 1. Ce qui donne une version encore plus simple.

```
1 (define (rtfact2 n)
2   (let boucle ((cpt n) (acc 1))
3     ;; acc = fact(n-cpt)
4     (if (= cpt 0)
5         acc
6         (boucle (- cpt 1) (* cpt acc))))))
```

Trouver ce que calcule la fonction interne `boucle` revient à trouver l'invariant de boucle en programmation impérative. A chaque tour de boucle, on vérifie pour cette version que  $acc = fact(n - cpt)$ .

### 14.4.1 Dérécursivation par “passage de continuation”

Lorsqu'un calcul partiel du résultat final à chaque appel récursif est impossible,

Le CPS (Continuation passing style) est une généralisation de la résorption des enveloppes. Avec le CPS, on passe en argument à l'appel récursif la continuation du calcul.

**Continuation** : pour un calcul donné, nom donné à ce qui doit être fait avec son résultat.

Une mise en pratique avec “factorielle” :

```
1 (define (kfact n)
2   (let boucle ((n n) (k (lambda(x) x)))
3     (if (= n 0)
```

```

4      (k 1)
5      (boucle (- n 1) (lambda(x) (k (* n x))))))

```

Pour comprendre comment le CPS fonctionne, explicitation de la continuation sous forme lisible via une liste.

```

1  (define (kfactExplication n)
2    (let boucle ((n n) (k (list 'lambda (list 'x) 'x)))
3      (if (= n 0)
4          (list k 1)
5          (boucle (- n 1) (list 'lambda (list 'x) (list k (list '* n 'x)))))))

```

```

1  (kfactExplication 0)
2  = ((lambda (x) x) 1)
3  (kfactExplication 1)
4  = ((lambda (x) ((lambda (x) x) (* 1 x))) 1)
5  (kfactExplication 2)
6  = ((lambda (x) ((lambda (x) ((lambda (x) x) (* 2 x))) (* 1 x))) 1)
7
8  (eval (kfactExplication 3))
9  = 6

```

## 14.5 Quelques mesures comparatives

Mesures de la différence de temps d'exécution entre versions avec la fonction `time`.

Utilisons une fonction `mult` qui ne construit ni très grand nombre ni grande liste, pour laquelle on mesure donc essentiellement le coût des appels récursifs.

```

1  (define (mult-it x n)
2    (do ((n n) (acc 0))
3      ((= n 0) acc)
4      (set! n (- n 1))
5      (set! acc (+ acc x))))
6
7  (define (mult-rt x n)
8    (let boucle ((n n) (acc 0))
9      (if (= n 0)
10         acc
11         (boucle (- n 1) (+ acc x))))))
12
13 (define (mult-k x n)
14   (let boucle ((n n) (k (lambda(x) x)))
15     (if (= n 0)
16         (k 0)
17         (boucle (- n 1) (lambda (y) (k (+ x y)))))))

```

```

1  n = 4000000
2  (time (mult-r 0 n))
3  cpu time: 31054 real time: 31216 gc time: 23742
4
5  (time (mult-k 0 n))
6  cpu time: 1411 real time: 1437 gc time: 541

```

```

7
8 (time (mult-rt 0 n))
9 cpu time: 756 real time: 773 gc time: 0
10
11 (time (mult-it 0 n))
12 cpu time: 1123 real time: 1157 gc time: 58

```

---

```

1 n = 50000000
2 (time (mult-k 0 n))
3 cpu time: 65938 real time: 66278 gc time: 55422
4
5 (time (mult-rt 0 n))
6 cpu time: 9187 real time: 9276 gc time: 0
7
8 (time (mult-it 0 n))
9 cpu time: 14036 real time: 14182 gc time: 1190

```

---

## 14.6 Recursion vers iteration, le cas des listes

Mêmes principes que précédemment, seuls les opérateurs changent.

Longueur d'une liste.

```

1 (define (longueur l)
2   (let boucle ((l2 l) (acc 0))
3     ;; acc = taille(l) - taille(l2)
4     (if (null? l2)
5         acc
6         (boucle (cdr l2) (+ 1 acc))))))

```

---

## 14.7 Autres exemples

### Somme des éléments d'une liste

Version explicitement itérative :

```

1 (define (do-somme-liste l)
2   (do ((l1 l) (acc 0))
3       ((null l1) acc)
4       (set! acc (+ acc (car l)))
5       (set! l1 (cdr l))))

```

---

Version récursive terminale :

```

1 (define (i-somme-liste l)
2   (let boucle ((l l) (acc 0))
3     (if (null? l)
4         acc
5         (boucle (cdr l) (+ (car l) acc))))))

```

---

## Renversement d'une liste

Version explicitement itérative.

```
1 (define (do-reverse l)
2   (do ((current l) (result ()))
3     ((null? l) result)
4     (set! result (cons (car l) result))
5     (set! l (cdr l))
6     ;; invariant : result == reverse (initial(l) - current(l))
7   ))
```

Version récursive terminale. L'accumulateur est une liste puisque le résultat doit en être une. A surveiller le sens dans lequel la liste se construit par rapport à la version récursive non terminale.

```
1 (define (ireverse l)
2   (let boucle ((l l) (acc ()))
3     (if (null? l)
4         acc
5         (boucle (cdr l) (cons (car l) acc))))))
```

## 14.8 Autres améliorations de fonctions récursives

Etude mathématique ou informatique du problème.

Exemple avec la fonction puissance : amélioration des formules de récurrence conjuguée à une dérécursivation.

- Version récursive standard. Cette version réalise  $n$  multiplications par  $x$  et consomme  $n$  blocs de pile.

```
1 (define (puissance x n)
2   (if (= n 0)
3       1
4       (* x (puissance x (- n 1)))))
```

- Une version récursive terminale conforme au schéma de dérécursivation précédent nécessite toujours  $n$  appels de fonction,  $n$  multiplications par  $x$  mais ne consomme plus de pile.

```
1 (define (puissance-v2 x n)
2   (let boucle ((cpt n) (acc 1))
3     ;; puis(x,n-cpt) = acc
4     (if (= cpt 0)
5         acc
6         (boucle (- cpt 1) (* acc x)))))
```

- Une version<sup>1</sup> récursive “dichotomique” :

si  $n$  est pair,  $\exists m = n/2$  et  $x^n = x^{2m} = (x^2)^m$   
si  $n$  est impair,  $x^n = x^{2m+1} = x \cdot x^{2m} = x \cdot (x^2)^m$

```
1 (define (puissance-v3 x n)
```

<sup>1</sup>Attention, utiliser “quotient” plutôt que “/” car quotient rend un entier compatible avec les fonctions “even” et “odd”.

```

2 (cond ((= n 0) 1)
3       ((even? n) (puissance-v3 (* x x) (quotient n 2)))
4       ((odd? n) (* x (puissance-v3 (* x x) (quotient n 2)))))

```

Fait passer de  $O(n)$  à  $O(\log_2(n))$  appels récursifs et multiplications.

- Couplage des deux améliorations

Une version itérative de la fonction puissance dichotomique suppose à nouveau le passage par un accumulateur

```

1 (define (puissance-v4 x n)
2   (let boucle ((x x) (n n) (acc 1))
3     (cond ((= n 0) acc)
4           ((even? n) (boucle (* x x) (quotient n 2) acc))
5           ((odd? n) (boucle (* x x) (quotient n 2) (* x acc)))))

```

## 14.9 Transformation des récursions arborescentes, l'exemple de “fib”

### 14.9.1 Inefficacité du CPS

“Continuation passing style” appliqué à la fonction fib.

```

1 (define (k-fib n k) ; la continuation est appelee k
2   (cond
3     ((= n 0) (k 0)) ; application de la continuation à 0
4     ((= n 1) (k 1)) ; application de la continuation à 1
5     (#t (k-fib (- n 1)
6              (lambda (acc1) ; construction de la fonction de continuation
7                (k-fib (- n 2)
8                      (lambda (acc2)
9                        (k (+ acc1 acc2))))))))))

```

Cette transformation est moins efficace que pour factorielle :

- il reste un appel récursif dans la continuation,
- la consommation en pile est remplacée par une consommation mémoire via le code des continuations (voir ci-après).

Elle peut être utile si elle bénéficie d'une prise en charge spéciale par le compilateur.

### 14.9.2 Une solution en $O(n)$ : la “mémoization”

Cette solution s'applique globalement au calcul de toutes les suites.

**Mémoization** : Du latin “memorandum”. “In computing, memoization is an optimization technique used primarily to speed up computer programs by storing the results of function calls for later reuse, rather than recomputing them at each invocation of the function” - Wikipedia

Il est amusant d'écrire une solution purement applicative à ce problème :

```

1 (define (memo-fib n)
2
3   (define cherche-dans-alist assq)

```

```

4  (define (val n memoire) (cadr (cherche-dans-alist n memoire)))
5
6
7  (define (memo n memoire)
8    ;; rend une liste dans laquelle la valeur de fib(n) et toutes les
9    ;; précédentes sont stockées
10   (let ((dejaCalcule (cherche-dans-alist n memoire)))
11     (if dejaCalcule
12         memoire
13         ;; il suffit de calculer fib(n-1), cela impliquera le calcul
14         ;; de fib(n-2) et son stockage dans "memoire".
15         (let ((memoire (memo (- n 1) memoire)))
16           (let ((fibn (+ (val (- n 1) memoire) (val (- n 2) memoire))))
17             ;; on ajoute à la liste memoire la dernière valeur calculée et on la rend
18             (cons (list n fibn) memoire))))))
19
20 (val n (memo n '((1 1) (0 0))))

```

### 14.9.3 Une version itérative en $O(n)$

Transformation du problème : utiliser des variables ou la pile d'exécution pour conserver en mémoire les deux dernières valeurs qui sont suffisantes pour calculer la suivante.

Observons les valeurs successives de deux suites :

$$a_n = a_{n-1} + b_{n-1} \text{ avec } a_0 = 1$$

et

$$b_n = a_{n-1} \text{ avec } b_0 = 0$$

On note que pour tout  $n$ ,  $b_n = fib(n)$ .

On en déduit la fonction récursive terminale suivante :

```

1  (define (ifib n)
2    (let boucle ((cpt 0) (a 1) (b 0))
3      (if (= cpt n)
4          b
5          (boucle (+ cpt 1) (+ a b) a))))

```