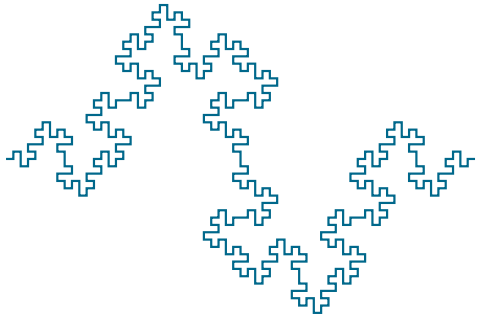
HLIN403 – Programmation Applicative

Optimisation des fonctions récursives

Christophe Dony – Annie Chateau
Université Montpellier – Faculté des Sciences



Introduction

Simplification de certaines fonctions récursives

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

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

Une structure de contrôle pour écrire les boucles en Scheme

```
(do (initialisation des variables)
    (condition-d'arret
       expression-si-condition-d'arret-vérifiée)
  (instruction 1)
  ...
  (instruction n))
```

Equivalence itération - récursions terminales

Du point de vue de la quantité d'espace pile consommé par l'interprétation, les deux fonctions suivantes sont équivalentes.

```
(define ipgcd
  (lambda (a b)
     (do ((m (modulo a b)))
       ((= m 0) b)
       (set! a b)
       (set! b m)
       (set! m (modulo a b)))))

(define pgcd
  (lambda (a b)
    (let ((m (modulo a b)))
      (if (= m 0)
          b
          (pgcd b m)))))
```

Transformation Récursion vers itération

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. Par ailleurs 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 (ce qu'il y a à faire une fois que l'appel récursif est achevé) en paramètre de l'appel récursif.

Récursions enveloppées simples : l'exemple de factorielle

Une version itérative (en C)

```
int fact(n){
  int cpt = 0;
  int acc = 1;
  while (cpt < n){
    // acc = fact(cpt) -- invariant de boucle
    cpt = cpt + 1;
    acc = acc * cpt;
    // acc = fact(cpt) -- invariant de boucle
  }
  // en sortie de boucle, cpt = n, acc = fact(n)
  return(acc)
}
```

Récursions enveloppées simples : l'exemple de factorielle

Une version itérative (en scheme)

```
(define ifact
  (lambda (n)
    (do ((cpt 0) (acc 1))
      ;; test d'arret et valeur rendue si le test est #t
      ((= cpt n) acc)
      ;; acc = fact(cpt) -- invariant de boucle
      (set! cpt (+ cpt 1))
      (set! acc (* acc cpt))
      ;; acc = fact(cpt) -- invariant de boucle
)))
```

Version récursive terminale (en scheme)

Solution générale à la dérécursivation : passer le calcul d'enveloppe en argument de l'appel récursif.

Exemple de factorielle : deux paramètres supplémentaires qui vont prendre au fil des appels récursifs les valeurs successives de `cpt` et `acc` dans la version itérative.

Version récursive terminale (en scheme)

```
(define ifact
  (lambda (n cpt acc)
    ;; acc = fact(cpt)
    (if (= cpt n)
       acc
       (ifact n (+ cpt 1) (* (+ cpt 1) acc)))))
```

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

Version récursive terminale (en scheme)

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

```
(define ifact
  (lambda (n)
    (letrec ((boucle (lambda (cpt acc)
                ;; acc = fact(cpt)
                (if (= cpt n)
                   acc
                   (boucle (+ cpt 1) (* (+ cpt 1) acc))))))
      (boucle 0 1))))
```

Version récursive terminale (en scheme)

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

```
(define ifact
  (lambda (n)
    (letrec ((boucle (lambda (cpt acc)
                ;; acc = fact(n-cpt)
                (if (= cpt 0)
                    acc
                    (boucle (- cpt 1) (* cpt acc))))))
      (boucle n 1)))
```

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 $acc = fact(n - cpt)$.

Version récursive terminale (en scheme)

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

```
(define f (lambda (x y) (+ x y)))

(define g
  (lambda (x n)
    (if (= n 0) 1
      (f x (g x (- n 1))))))

(define g2
  (lambda (x n)
    (letrec ((boucle (lambda (cpt acc)
      (if (= cpt 0) acc
          (boucle (- cpt 1) (f acc x))))))
    (boucle n 1))))
```

Version récursive terminale (en scheme)

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

```
(time (g 2  700000))
(time (g 2  700000))
(time (g 2  700000))
(time (g 2  700000))
```

```
(time (g2 2  700000))
(time (g2 2  700000))
(time (g2 2  700000))
(time (g2 2  700000))
```

Récursion vers itération, le cas des listes

Mêmes principes que précédemment, les opérateurs changent.

```
(define longueur
  (lambda (l)
    (letrec ((boucle (lambda (l1 acc)
        ;; acc = taille(l) - taille(l1)
        (if (null l1)
            acc
            (boucle (cdr l1) (+ 1 acc))))))
      (boucle l 0))))
```

Autres exemples de transformation

Somme des éléments d'une liste

Version explicitement itérative :

```
(define do-somme-liste
  (lambda (l)
    (do ((l1 l) (acc 0))
        ((null l1) acc)
        (set! accc (+ acc (car l)))
        (set! l1 (cdr l1)))))
```

Autres exemples de transformation

Version récursive terminale :

```
(define isomme-liste
  (lambda (l)
    (letrec ((boucle (lambda (l acc)
                       (if (null l) acc
                (boucle (cdr l) (+ (car l) acc))))))
      (boucle l 0))))
```

Autres exemples de transformation

Renversement d'une liste

Version explicitement itérative.

```
(define (do-reverse l)
  (do ((current l) (result ()))
    ((null? l) result)
    (set! result (cons (car l) result))
    (set! l (cdr l))
    ;; invariant :
    ;; result == reverse (initial(l) - current(l))
    ))
```

Autres exemples de transformation

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.

```
(define ireverse
  (lambda (l)
    (letrec ((boucle (lambda (l acc)
                       (if (null? l)
                           acc
                           (boucle (cdr l) (cons (car l) acc))))))
      (boucle l ()))))
```

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.

```
(define puissance
  (lambda (x n)
    (if (= n 0)
      1
      (* x (puissance x (- n 1))))))
```

Autres améliorations de fonctions récursives

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

```
(define puissance-v2
  (lambda (x n)
    (letrec ((boucle (lambda (cpt acc)
                       ;; puis(x,n-cpt) = acc
                       (if (= cpt 0)
                           acc
                           (boucle (- cpt 1) (* acc x))))))
      (boucle n 1))))
```

Autres améliorations de fonctions récursives

- Une version récursive dite “dichotomique” nécessitant moins de multiplications. Elle est basée sur la propriété suivante de la fonction *puissance* :
  si n est pair, $\exists m = n/2$ et $x^n = x^{2m} = (x^2)^m$,
  si n est impair, $x^n = x^{2m+1} = x.x^{2m} = x.(x^2)^m$,

```
(define puissance-v3
  (lambda (x n)
    (cond ((= n 0) 1)
      ((even? n) (puissance-v3 (* x x) (quotient n 2)))
      ((odd? n) (* x (puissance-v3 (* x x) (quotient n 2)))))))
```

Cette version [1] n’est pas récursive terminale quand n est impair, par contre elle n’effectue plus que de l’ordre de $log(n)$ multiplications.

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

Autres améliorations de fonctions récursives

- Couplage des deux améliorations. Une version itérative de la fonction puissance dichotomique suppose à nouveau le passage par un accumulateur passé en paramètre.

```
(define puissance-v4
  (lambda (x n)
    (letrec ((boucle (lambda (x n acc)
        (cond ((= n 0) acc)
          ((even? n)
              (boucle (* x x) (quotient n 2) acc))
          ((odd? n)
              (boucle (* x x) (quotient n 2) (* x acc)))))))
     (boucle x n 1))))
```

Transformation des récursions arborescentes - L'exemple de Fibonacci

Solution générale

D'une façon générale, les récursions arborescentes peuvent être transformées en itérations en passant en paramètre la continuation.

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

Exemple : avec la fonction factorielle, la continuation de l'appel récursif est une multiplication par n.

Pour Fibonacci, après le premier appel récursif qui donne un résultat `acc1`, il y a un autre appel récursif à réaliser qui donnera un résultat `r2` et après le second appel récursif il y a une addition des deux.

Transformation des récursions arborescentes - L'exemple de Fibonacci

Le CPS (Continuation passing style) est une généralisation de la résorbtion des enveloppes.

Avec le CPS, on passe en argument à l'appel récursif, la continuation du calcul. Pour Fibonacci, l'appel récursif est terminal et on passe en argument la fonction qui devra être éxécutée une fois la valeur trouvée. L'utilisation de ce style en toute généralité suppose de pouvoir passer des fonctions en arguments.

Transformation des récursions arborescentes - L'exemple de Fibonacci

```
(define k-fib
  (lambda (n k) ; la continuation est appelee k
    (cond
        ; application de la continuation à 0
        ((= n 0) (k 0))
        ; application de la continuation à 1
        ((= n 1) (k 1))
        (#t (k-fib (- n 1)
             ; construction de la fonction de continuation
             (lambda (acc1)
                (k-fib (- n 2)
                       (lambda (acc2)
                         (k (+ acc1 acc2)))))))))
```

Transformation des récursions arborescentes - L'exemple de Fibonacci

Cette version est uniquement intéressante en théorie. Elle ne consomme pas de pile mais son exécution est pourtant encore plus longue que celle de fib standard parce que :

- on effectue le même nombre d'additions que dans la version standard
- la consommation mémoire en pile est remplacée par une consommation mémoire encore plus grande en code des fonctions intermédiaires. Pour s'en convaincre on peut imprimer k en début de fonction k-fib.

Une solution en $0(n)$: la "mémoization"

Memoization :

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

Une solution en $0(n)$: la "mémoization"

```
(define memo-fib
  (lambda (n)
    (define val (lambda (n memoire)
        (cadr (assq n memoire))))
    (define memo (lambda (n memoire)
      ;; rend une liste où la valeur de fib(n) est stockée
      (let ((dejaCalcule (assq n memoire)))
        (if dejaCalcule
          memoire
          (let ((memoire (memo (- n 1) memoire)))
            (let ((fibn (+ (val (- n 1) memoire)
                           (val (- n 2) memoire))))
              ;; on ajoute à la liste memoire la dernière
              ;; valeur calculée et on la rend
              (cons (list n fibn) memoire))))))))

    (val n (memo n '((1 1) (0 0))))))
```

Une version itérative de `fib` 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 :

$a_n = a_{n-1} + b_{n-1}$ avec $a_0 = 1$
et
$b_n = a_{n-1}$ avec $b_0 = 0$

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

Une version itérative de `fib` en $O(n)$

On en déduit la fonction récursive terminale suivante :

```
(define ifib
  (lambda (n a b)
    (if (= n 0)
      b
      (ifib (- n 1) (+ a b) a))))
```

ou

```
(define ifib
  (lambda (n)
    (letrec ((boucle (lambda (cpt a b)
                       (if (= cpt n)
                         b
                         (boucle (+ cpt 1) (+ a b) a)))))
      (boucle 0 1 0))))
```