Optimisation des fonctions récursives

Université Montpellier-II - UFR - ULIN201 - Langages Applicatifs
- C. Dony -
Cours No 7

14 Simplification de certaines 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

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

```
x := 12;        ;; en Algol, Simula, Pascal (syntaxe classique)
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))
```

14.2 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 a b)
   (do ((m (modulo a b)))
       ((= m 0) b)
       (set! a b)
       (set! b m)
       (set! m (modulo a b))))
```

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

14.3 Récursions enveloppées simples : l'exemple de factorielle

14.3.1 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)
}
```

14.3.2 Une version itérative (en scheme)

```
(define (ifact n)
  (do ((cpt 0) (acc 1))
    ;; test d'arrèt 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
   ))
```

14.3.3 Version récursive terminale (en scheme)

Solution générale à la dérécursivation : passer le calcul d'enveloppe (ce qu'il y a à faire une fois que l'appel récursif est achévé) en argument de l'appel récursif.

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

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

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

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

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

Version encore plus simple où le compteur est initialisé à `n`. (multiplication associative donc ordre des multiplications est indifférent).

```
(define (ifact 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 $acc = fact(n - cpt)$.

14.4 Recursion vers iteration, le cas des listes

Mêmes principes, les opérateurs changent.

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

14.5 Autres exemples de transformation

Somme des éléments d'une liste

Version explicitement itérative :

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

Version récursive terminale :

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

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))
    ))
```

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

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

– 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 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)))
```

14.7 Transformation des récursions arborescentes - L'exemple de fibonacci.

14.7.1 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.

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 fib, 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.

```
(define (k-fib n k) ; la continuation est appelee k
  (cond ((= n 0) (k 0)) ; application de la continuation au resultat
        ((= n 1) (k 1)) ; application de la continuation au resultat
        (#t (k-fib (- n 1)
             (lambda (acc1)     ; explicitation de la continuation
                (k-fib (- n 2)  ; sous forme d'une fonction
                       (lambda (acc2)
                         (k (+ acc1 acc2)))))))))
```

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'addition 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.

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

14.7.2 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

```
(define (memo-fib n)

  (define (val n memoire)
    (cadr (assq n memoire)))

  (define (memo n memoire)
    ;; rend une liste dans laquelle la valeur de fib(n) est stockée
    (let ((dejaCalcule (assq n memoire)))
      (if dejaCalcule
          memoire
          (let ((memoire (memo (- n 1) memoire)))
            (let ((memoire (memo (- n 2) 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)))))
```

14.7.3 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, $b_n = fib(n)$.

On en déduit la fonction récursive terminale suivante :

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

ou

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