LYCÉE LECONTE DE LISLE

Programmation par continuation

Vincent Picard

1
Récursivité terminale

Fonctions récursives terminales

Une fonction récursive est **récursive terminale** (*tail-recursive*) si aucune opération n'est effectuée sur la valeur de retour des appels récursifs : autrement dit la valeur de retour d'un appel récursif est toujours la valeur de retour de la fonction initiale.

- Cette fonction n'est pas récursive terminale :

```
let rec fact n =
    if n = 0 then 1
    else n * (fact (n-1))
;;
```

- Mais cette fonction aux l'est :

```
let fact n =
    let rec aux k accu =
        if k = 0 then accu
        else aux (k-1) (n * accu)
    in aux n 1;;
```

Avantage et inconvénient

- **Avantage :** Les fonctions récursives terminales sont plus faciles à optimiser pour le compilateur car elles ne nécessitent pas d'empiler un contexte sur la pile d'exécution à chaque appel récursif : plus d'efficacité et absence de *stack overflow*.

- **Inconvénient :** Les fonctions récursives terminales sont plus difficiles à écrire, à lire et à comprendre... Il ne faut surtout pas se "forcer" à écrire des versions récursives terminales

- En utilisant la méthode qu'on va étudier, certains compilateurs transforment automatiquement les fonctions récursives pour qu'elles soient récursives terminales. On obtient alors du code compilé efficace même si le programme a été codé récursivement.

Exercices

Proposer deux versions de fonction (récursive terminale ou non) pour les fonctions suivantes :

1. `somme_entiers n` : qui calcule la somme des entiers de 0 à n inclus.
2. `max_val l` : qui retourne la valeur maximale d'une liste non vide d'entiers
3. `taille a` : qui retourne la taille d'un arbre binaire a dont le type est :

```
type arbre = Vide | Noeud of (arbre * arbre);;
```

- On remarque qu'on utilise souvent la méthode de l'**accumulateur** pour obtenir une version récursive terminale.
- L'exemple de `taille` montre que la méthode de l'accumulateur n'est pas toujours simple à mettre en œuvre.

Un cas vraiment trop difficile...

On travaille toujours sur les arbres binaires :

```
type arbre = Vide | Noeud of (arbre * arbre);;
```

et on veut maintenant coder la fonction **hauteur** de l'arbre :

```
let rec hauteur a = match a with
    | Vide -> (-1)
    | Noeud (g, d) -> 1 + max (hauteur g) (hauteur d)
;;
```

Il est très difficile d'écrire une version récursive terminale de cette fonction (même en utilisant la méthode de l'accumulateur)

2
Programmation par continuation

Programmation par continuation

La **programmation par continuation** (*continuation-passing style*) consiste à ajouter à chaque fonction récursive un paramètre `k` qui est une fonction qui prend en argument le résultat de l'appel de fonction et qui indique quoi en faire. Elle s'oppose à la **programmation directe**.

- On dit souvent que le paramètre `k` représente le **futur** de la fonction : que va-t-on faire du résultat ?
- **La programmation par continuation rend toutes les fonctions récursives terminales !**

Un exemple

```
let rec fact_cont n k =
    if n = 0 then (k 1)
    else fact_cont (n-1) (fun x -> k (n * x))
;;
```

- **Cas de base** : si $n = 0$, alors $n! = 1$ donc on applique le futur k sur le résultat 1.
- **Cas récursif** : sinon on calcule $(n-1)!$ et on donne le futur suivant :
  - commence par multiplier la valeur obtenue par n...
  - ... puis applique le futur k passé en argument
- **Exemples d'utilisation**

```
fact_cont 5 (fun x -> x);; (* Pour obtenir la valeur de 5! *)
fact_cont 5 print_int;; (* Pour afficher 5! à l'écran *)
```

Résolution du cas difficile

Revenons au problème de la hauteur d'un arbre binaire :

```
let rec hauteur_cont a k = match a with
    | Vide -> k (-1)
    | Noeud(g, d) ->
        hauteur_cont g (fun x ->
        hauteur_cont d (fun y ->
        k (1 + max x y)
        ))
;;
```

Cette fonction doit se lire ainsi :

- Commence par calculer la hauteur de g ensuite avec le résultat x tu feras :
- Calcule la hauteur de d ensuite avec le résultat y tu feras :
- Calcule $1 + \max(x, y)$ et donne cela au futur k de la fonction hauteur.

Exercices

Proposer des versions *par continuation* des fonctions suivantes :

1. `somme_entiers_cont n k` : qui calcule la somme des entiers de 0 à n inclus.
2. `max_val_cont l k` : qui renvoie la valeur maximale d'une liste non vide d'entiers
3. `taille_cont a k` : qui renvoie la taille d'un arbre binaire a dont le type est :

```
type arbre = Vide | Noeud of (arbre * arbre);;
```

4. `concat l1 l2 k` : qui renvoie la concaténation des listes l_1 et l_2.

Exemples : parcours d'arbres binaires

On travaille avec les arbres binaires :

```
type 'a arbre = Vide | Noeud of ('a * 'a arbre * 'a arbre);;
```

En utilisant la fonction `concat` définie précédemment, écrire des versions par continuation des fonctions suivantes de parcours d'arbre :

1. `parcours_prefixe a k`
2. `parcours_infixe a k`
3. `parcours_suffixe a k`

Ici, `k` désigne donc une fonction qui prend en entrée la liste des étiquettes du parcours. Pour tester son code, on peut par exemple écrire :

```
parcours_prefixe a_test (List.iter print_int)
```

Solutions

1. **Parcours préfixe**

```
let rec parcours_prefixe a k =
    match a with
    | Vide -> k []
    | Noeud (x, g, d) ->
            parcours_prefixe g (fun parg ->
            parcours_prefixe d (fun pard ->
            concat parg pard (fun y -> k (x::y))));;
```

2. **Parcours infixe**

```
let rec parcours_infixe a k =
    match a with
    | Vide -> k []
    | Noeud (x, g, d) ->
            parcours_infixe g (fun parg ->
            parcours_infixe d (fun pard ->
            concat parg (x::pard) k));;
```

3. **Parcours suffixe**

```
let rec parcours_suffixe a k =
    match a with
    | Vide -> k []
    | Noeud (x, g, d) ->
            parcours_suffixe g (fun parg ->
            parcours_suffixe d (fun pard ->
            concat pard [x] (fun z ->
                concat parg z k)
            ))
;;
```

Remarque : cela ne règle pas le problème de la mauvaise complexité liée à l'utilisation de `concat`...

Bon parcours préfixe

- Pour obtenir une complexité linéaire du parcours, on peut écrire en programmation **directe** :

```
let bon_pprefixe a =
    let rec aux a suite = match a with
    | Vide -> suite
    | Noeud (x, g, d) -> x::(aux g (aux d suite))
    in aux a []
;;
```

- Écrire une version **par continuation** de cette même fonction.

Bon parcours préfixe : solution

- Pour obtenir une complexité linéaire du parcours, on peut écrire en programmation **directe** :

```
let bon_pprefixe a =
    let rec aux a suite = match a with
    | Vide -> suite
    | Noeud (x, g, d) -> x::(aux g (aux d suite))
    in aux a []
;;
```

- Écrire une version **par continuation** de cette même fonction.

```
let bon_pprefixe_cont a k =
    let rec aux a suite k = match a with
    | Vide -> k suite
    | Noeud (x, g, d) ->
            aux d suite (fun y ->
            aux g y (fun z -> k (x::z)))
    in aux a [] k
;;
```

Conclusion

- La programmation par continuation est une construction intéressante et amusante...
- ...de plus elle permet d'obtenir des fonctions récursives terminales même dans des cas où ce n'est pas évident.
- **Cependant elle rend l'écriture et la lecture des fonctions bien plus difficiles. Ne pas utiliser ce mode de programmation au concours (sauf si c'est explicitement demandé par le sujet)**