Continuation Passing Style (CPS) et Exemple en OCaml

Définition du CPS

- Le **Continuation Passing Style (CPS)** est un paradigme de programmation fonctionnelle dans lequel, plutôt que de retourner directement un résultat, une fonction reçoit en argument une *continuation*.

Définition du CPS

- Le **Continuation Passing Style (CPS)** est un paradigme de programmation fonctionnelle dans lequel, plutôt que de retourner directement un résultat, une fonction reçoit en argument une *continuation*.
- Cette continuation est une autre fonction spécifie ce qu'il faut faire avec le résultat calculé. Autrement dit, chaque fonction ne "renvoie" pas son résultat, mais le transmet à une fonction de continuation qui représente la suite du calcul.

Avantages du CPS

Ce style présente plusieurs avantages :

- **Contrôle explicite du flux d'exécution :** Le programme est décomposé en continuations, ce qui facilite l'implémentation de mécanismes comme les exceptions, la gestion de l'asynchronisme ou la transformation de la récursion en récursion terminale (ce qui nous intéresse ici).
- **Optimisation de la récursion :** Les appels récursifs étant en position terminale, le compilateur peut mieux les optimiser.

Fonction CPS pour la hauteur

```
1  type 'a tree = Nil | Node of 'a tree * 'a * 'a tree;;
2
3  let hauteur (a: 'a tree): int =
4    let rec height_cps a cont = match a with
5      | Nil -> cont @@ -1
6      | Node(g,_,d) ->
7        height_cps g (fun resg ->
8            height_cps d (fun resd ->
9                    cont (1 + max resg resd)))
10   in height_cps a (fun x -> x);;
```

Listing 1 – Exemple de code OCaml en CPS

Explication détaillée : Cas de base et récursif

- **Cas de base :** Pour un arbre vide (`Nil`), la hauteur est -1. On appelle alors la continuation `cont` avec -1 (`cont (-1)`).

Explication détaillée : Cas de base et récursif

- **Cas de base :** Pour un arbre vide (`Nil`), la hauteur est -1. On appelle alors la continuation `cont` avec -1 (`cont (-1)`).
- **Cas récursif :** Pour un nœud (`Node`), l'arbre possède deux sous-arbres, `g` et `g`.

Explication détaillée : Cas de base et récursif

- **Cas de base :** Pour un arbre vide (`Nil`), la hauteur est -1. On appelle alors la continuation `cont` avec -1 (`cont (-1)`).
- **Cas récursif :** Pour un nœud (`Node`), l'arbre possède deux sous-arbres, `g` et `g`.
  - On appelle récursivement `height_cps g` en lui passant une continuation qui récupère la hauteur de `g` (`resg`).

Explication détaillée : Cas de base et récursif

- **Cas de base :** Pour un arbre vide (`Nil`), la hauteur est -1. On appelle alors la continuation `cont` avec -1 (`cont (-1)`).
- **Cas récursif :** Pour un nœud (`Node`), l'arbre possède deux sous-arbres, `g` et `g`.
  - On appelle récursivement `height_cps g` en lui passant une continuation qui récupère la hauteur de `g` (`resg`).
  - Ensuite, dans cette continuation, on appelle `height_cps d` avec une continuation qui reçoit la hauteur de `d` (`resd`).

Explication détaillée : Cas de base et récursif

- **Cas de base :** Pour un arbre vide (`Nil`), la hauteur est -1. On appelle alors la continuation `cont` avec -1 (`cont (-1)`).
- **Cas récursif :** Pour un nœud (`Node`), l'arbre possède deux sous-arbres, `g` et `g`.
  - On appelle récursivement `height_cps g` en lui passant une continuation qui récupère la hauteur de `g` (`resg`).
  - Ensuite, dans cette continuation, on appelle `height_cps d` avec une continuation qui reçoit la hauteur de `d` (`resd`).
  - Finalement, on calcule la hauteur du nœud courant comme `1 + max resg resd` et on transmet ce résultat à la continuation initiale.

Conclusion

L'exemple montre comment le style CPS permet de gérer le passage de résultat de manière explicite et modulable. Ce paradigme est particulièrement utile pour obtenir un contrôle fin du flux d'exécution et optimiser la récursion, même si, pour des calculs simples comme la hauteur d'un arbre, une approche standard peut suffire.

Exercice

Écrire une fonction qui calcule la taille d'un arbre binaire avec un CPS.

Principe

- On simule manuellement la pile d'appels en utilisant une structure de données (comme une liste) pour stocker les nœuds restants à traiter.

Principe

- On simule manuellement la pile d'appels en utilisant une structure de données (comme une liste) pour stocker les nœuds restants à traiter.
- Chaque étape du parcours consomme la pile et effectue un appel récursif terminal.

Calcul de la hauteur avec gestion de pile explicite

```
1  let hauteur arbre =
2    let rec loop stack max_height =
3      match stack with
4      | [] -> max_height (* Si la pile est vide, on retourne la
           hauteur maximale trouvée *)
5      | (n, h) :: rest -> (* n : noeud courant, h : hauteur
           courante *)
6        match n with
7        | Nil -> loop rest (max max_height h)
8        (* On met à jour la hauteur maximale *)
9        | Node(g, _, d) ->
10         (* On empile les sous-arbres avec la hauteur incrémenté
               e *)
11         loop ((g, h + 1) :: (d, h + 1) :: rest) max_height
12   in
13   loop [(arbre, -1)] (-1)
14   (* On commence avec l'arbre et une hauteur de 0 *)
```

Explications

- La pile explicite remplace la pile d'appels récursifs, ce qui permet d'éviter les limitations de la récursion non terminale.

Explications

- La pile explicite remplace la pile d'appels récursifs, ce qui permet d'éviter les limitations de la récursion non terminale.
- On maintient une hauteur courante (`h`) pour chaque nœud, et on met à jour la hauteur maximale (`max_height`) à chaque fois qu'on atteint une feuille (`Nil`).