OCaml

Récursion terminale (MPSI)

Crédits

- ce cours

Crédits

- ce cours
- Developpez.com

Un exemple ♡

Considérons le programme suivant. Il calcule la somme des entiers de 0 à 1 000 000 :

```
let rec f n =
  if n=0 then 0 else n + f (n-1)
let v = f 1_000_000
```

Compilation, exécution :

```
$ ocamlopt somme.ml -o somme
$ ./somme
Fatal error: exception Stack_overflow
```

Il y a débordement de pile : le nombre de stack frame est trop grand.

Analyse ♡

- On empile puis dépile les stack frame

Analyse ♡

- On empile puis dépile les stack frame
- Chaque stack frame contient une sauvegarde des registres du processeur ; un espace pour stocker la valeur de retour ; le paramètre ; l'adresse de retour

Analyse ♡

- On empile puis dépile les stack frame
- Chaque stack frame contient une sauvegarde des registres du processeur ; un espace pour stocker la valeur de retour ; le paramètre ; l'adresse de retour
- Entrons la commande suivante

```
$ ulimit -s
8192
```

On obtient donc que la taille de la pile d'appel est de 8 Mo.

Analyse ♡

- On empile puis dépile les stack frame
- Chaque stack frame contient une sauvegarde des registres du processeur ; un espace pour stocker la valeur de retour ; le paramètre ; l'adresse de retour
- Entrons la commande suivante

```
$ ulimit -s
8192
```

On obtient donc que la taille de la pile d'appel est de 8 Mo.

- Avec `f 1_000_000` on empile donc $1\,000\,000 + 1$ stack frame d'au moins 8 bytes. On comprend que la taille allouée à la pile soit dépassée.

Pile d'exécution

Si la récursion est écrite sans prudence, à chaque appel récursif de la fonction, le programme allouera un nouvel environnement d'exécution dans la pile, avec un risque de dépassement de la capacité de la pile. Exemple

```
let rec f x = if x > maxStackSize then x else 1 + f (x+1)
```

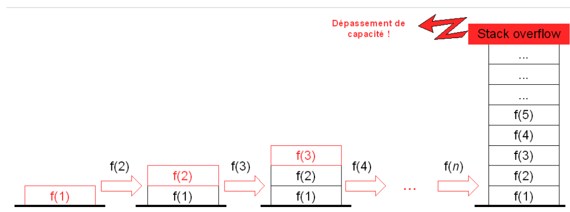

Figure – La pile d'exécution de la fonction `f` (image Developpez.com)

L'environnement de l'appel `f(1)` est gardé en mémoire en même temps que ceux des appels ultérieurs puisque, pour être calculé, il doit attendre que tous les `f(i)` ultérieurs soient évalués.

Appel terminal

- La *récursivité terminale* est une forme particulière de récursivité pouvant être optimisée afin de ne pas consommer de mémoire dans la pile.
- Dans le corps d'une fonction, un appel est *terminal* s'il est la dernière opération effectuée par la fonction.
- Les fonctions suivantes font un appel terminal à `g`

```
let f1 x = g x
let f2 x = if ... then g x else ...
let f3 x = let y = ... in g y
let f4 x = match x with
      | ... -> ...
      | ... -> g x
      | _   -> ...
```

- Appels à `g` non terminaux :

```
let f5 x = x + g x
let f6 x = let y = g x in y+1
```

Récursion terminale

- Une fonction est dite *récursive terminale* si tous les appels récursifs dans sa définition sont en position terminale.

Récursion terminale

- Une fonction est dite *récursive terminale* si tous les appels récursifs dans sa définition sont en position terminale.
- Intérêt : il n'est plus nécessaire d'empiler les stack frame lors des appels récursifs. La stack frame de départ suffit.

Pile d'exécution

Avec une fonction récursive terminale, l'environnement de l'appel `f(1)` n'est plus gardé en mémoire. Il est écrasé par celui des appels `f(i)` successifs.

Exemple

```
let rec f x = if x > maxStackSize then x else f (x+1+1)
```

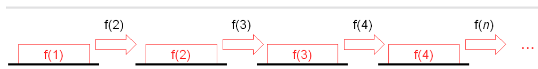

FIGURE – Avec une récursion terminale

La *stack frame* de l'appel `f(i)` est écrasée par celle de `f(i+1)` puisque le retour de `f(n)` est égal à celui de ces prédecesseurs.

Fonction somme : version en récursion terminale

- On utilise une fonction auxiliaire `sum` à deux paramètres : l'entier courant `x` et un *accumulateur* `acc` .

Fonction somme : version en récursion terminale

- On utilise une fonction auxiliaire `sum` à deux paramètres : l'entier courant `x` et un *accumulateur* `acc`.
- L'accumulateur grossit au fil des appels récursifs internes et il contient la valeur voulue lorsque `x` devient nul. On renvoie donc `acc`.

Fonction somme : version en récursion terminale

- On utilise une fonction auxiliaire `sum` à deux paramètres : l'entier courant `x` et un *accumulateur* `acc`.
- L'accumulateur grossit au fil des appels récursifs internes et il contient la valeur voulue lorsque `x` devient nul. On renvoie donc `acc`.
- Code :

```
1 let f x =
2   let rec sum x acc = (*fonction auxiliaire*)
3     if x = 0 then acc else sum (x-1) (acc + x)
4   in sum x 0
5
6 f 1_000_000 in Printf.printf "%d" (f 1000000)
```

Fonction somme : version en récursion terminale

- On utilise une fonction auxiliaire `sum` à deux paramètres : l'entier courant `x` et un *accumulateur* `acc`.
- L'accumulateur grossit au fil des appels récursifs internes et il contient la valeur voulue lorsque `x` devient nul. On renvoie donc `acc`.
- Code :

```
let f x =
  let rec sum x acc = (*fonction auxiliaire*)
    if x = 0 then acc else sum (x-1) (acc + x)
  in sum x 0

f 1_000_000 in Printf.printf "%d" (f 1000000)
```

- D'une façon générale, une fonction auxiliaire est utile lorsqu'on a besoin de plus de paramètres que ceux initialement prévus.