RÉCURSIVITÉ TERMINALE

1 Pile d'appels récursifs

1.1 Schéma d'organisation interne

Le système d'exploitation des ordinateurs organise la mémoire en plusieurs zones : entre autres, une pour le code du programme exécuté, une pour les données de ce programme et une zone particulière appelée pile d'appels (en anglais, stack).

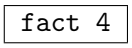
Pour simplifier, lorsqu'un programme fait appel à une fonction, le système d'exploitation enregistre au sommet de la pile d'appels les informations nécessaires pour que le résultat calculé par la fonction soit utilisée au bon moment : qui est l'appelant de la fonction, quelles sont les actions qui attendent le résultat de la fonction pour continuer, etc.

Illustrons cela avec la fonction récursive factorielle en simplifiant à l'extrême le contenu de la pile d'appels (en pratique, c'est bien plus compliqué !).

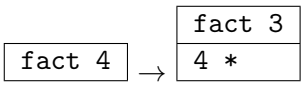
```
let rec fact n =
  if n = 0 then 1
  else n * fact (n-1);;
```

Supposons qu'on veuille calculer `fact 4`. Le programme appelant (par exemple, l'interface graphique de CAML) demande au système d'exploitation d'appeler la fonction `fact` avec l'argument 4.

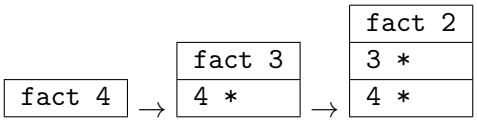
Il empile l'appel à `fact 4` et analyse la pile : en son sommet se trouve un appel de fonction donc il exécute cet appel et remplace le sommet de la pile par son résultat : c'est là que le programme principal retrouvera le résultat qu'il a demandé.



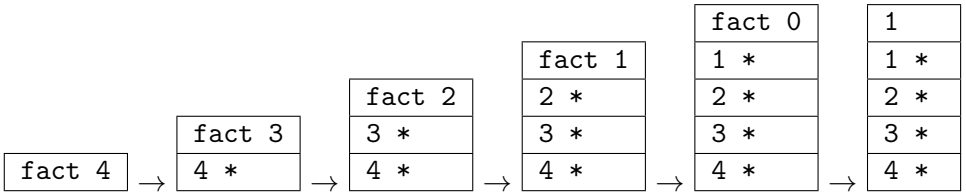
Pour calculer `fact 4`, le s.e. doit calculer $4\times$ `fact 3`, qui nécessite un appel à la fonction `fact`, avec en plus un calcul à effectuer sur le résultat de `fact 3`. Il empile donc l'opération à effectuer qui est momentanément suspendue (la multiplication par 4) et l'appel `fact 3`.



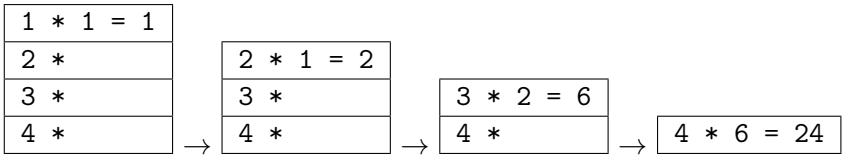
Puis on réitère : l'appel au sommet est remplacé par son résultat.



Et ainsi de suite



À cet instant, le s.e. rencontre une valeur concrète au sommet de la pile. Il est programmé pour alors dépiler cette valeur, dépiler la fonction en dessous et appliquer la fonction à la valeur : cela donne une valeur qui est remise au sommet de la pile. Puisque c'est encore une valeur concrète, on réitère, jusqu'à ce que la pile soit de hauteur 1 : un seul objet, qui sera une valeur concrète et le résultat du processus complet.



Lorsque la pile d'appel est de hauteur 1, le s.e. redonne la main au programme principal qui trouve au sommet de la pile la valeur qu'il attendait pour continuer son déroulement.

1.2 Complexité théorique et réalité

Lorsqu'on calcule la complexité d'un algorithme quelconque, on néglige le coût de la gestion de la pile d'appels par le s.e. : de toute façon, ce coût est présent quelque soit la machine sur laquelle le code est exécuté. Il ne représente rien de l'algorithme lui-même.

Mais en pratique, les actions sur la pile prennent du temps et de l'espace ! Un programme organisé pour faire le moins souvent usage de cette pile grapille des micro-secondes qui s'accumulent...

Autre problème : la pile d'appels a une capacité limitée. Si le nombre d'appels de fonctions est trop important, elle sera remplie avant que les calculs soient terminés (débordement de pile – stack overflow).

Un algorithme récursif classique utilise par essence cette pile, puisqu'il fait appel à lui-même : on ne peut donc empêcher cet usage. Mais on peut le limiter en évitant que la pile ne grandisse trop, car plus elle est haute, plus le temps pour revenir au programme principal sera long. On essaye donc d'éviter les empilements de calculs suspendus...

2 Récursivité terminale

2.1 Définition

Pour éviter que la pile d'appels ne grandisse, il suffit que tous les calculs soient faits *avant* l'appel récursif...

Définition. On dit qu'un algorithme est récursif terminal lorsqu'il ne comporte qu'un seul appel récursif et que cet appel est la dernière action de l'algorithme.

Dans la mise en œuvre d'un algorithme récursif terminal en CAML, la pile d'appels ne grandit jamais, puisque le compilateur a détecté une récursivité terminale et dès lors, chaque appel est directement remplacé par un autre appel.

Exemple d'algorithme récursif terminal :

```
let somme liste =
  let rec somme_aux li s =
    match li with
    | [] -> s
    | a :: q -> somme_aux q (s+a)
  in
  somme_aux liste 0;;
```

2.2 Transformation en récursif terminal

Supposons qu'on ait un algorithme itératif (en pseudo-code) schématiquement décrit

```
FI(...)=
  (* initialisation *)
  v := v0;
  (* boucle *)
  tant que C(v) faire
    v := f(v)
  finfaire;
  retourner v;;
```

On dit que la variable v est un accumulateur.

Le schéma d'un algorithme récursif terminal calculant la même valeur est

```
FRT(...)=
  (* alg. récursif encapsulé *)
  F_aux(v)=
    si C(v) fausse alors retourner v
    sinon retourner F_aux( f(v) );
  (* appel à cet algo. *)
  retourner F_aux(v0);;
```

Si les conditions du théorème fondamental de la récursivité sont satisfaites, alors on peut montrer par induction que ces deux algorithmes calculent la même valeur.

En pratique, il y a souvent plusieurs variables qui sont modifiées à chaque itération de la boucle : il faut alors considérer que v est le n-uplet de ces variables et que f est une fonction globale qui transforme le n-uplet au début de l'itération i en le n-uplet final. Dans le cas des boucles "for", le compteur est à prendre en compte.

3 Mise en œuvre dans CAML

Une fonction auxiliaire très utile : la concaténation en sens inverse.

```
let rec renverse li1 li2 =
  match li1 with
  | [] -> li2
  | a :: q1 -> renverse q1 (a :: li2);;
```

Cette fonction est récursive terminale et de complexité linéaire par rapport à la longueur de la première liste (indépendante de la longueur de la deuxième).

3.1 Fonction rev

```
let rev li =
  renverse li [];;
```

La complexité est linéaire par rapport à la longueur de la liste.

3.2 Opérateur @

```
let rec concat li1 li2 =
  let il1 = rev li1 in
  renverse il1 li2;;
```

La complexité est linéaire par rapport à la longueur de la première liste. CAML offre l'opérateur @ : concat p q = p @ q.