Informatique en MP(2)I

Marc de Falco

Récursivité

Source de l'image d'en-tête : Paul Noth. **À changer par une image CC**

■ **Note 1 Roadmap :**
- finir l'écriture.

■

I La récursivité

I.1 Principe et exemples

On dit qu'une fonction est récursive lorsqu'elle va s'appeler elle-même lors de son exécution. Le plus souvent, cet appel sera directement visible depuis le corps de la fonction.

Par exemple, la fonction `fact` suivante permettant de calculer $n!$ est récursive :

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

C
```
int fact(int n)
{
    if (n == 0)
    {
        return 1;
    }
    else
    {
        return n * fact(n-1);
    }
}
```

On utilise ici un exemple mathématique pour la simplicité des fonctions écrites. En effet, on ne voit rien d'autre dans `fact` que ces appels récursifs et ils correspondent directement à la définition mathématique :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon.} \end{cases}$$

Remarque En fait, la définition mathématique se fait plutôt en posant :

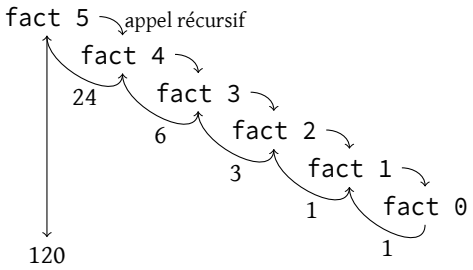
$$0! = 1 \qquad (n+1)! = (n+1) \times n!$$

Donc, pour retraduire une telle définition en tant que fonction, il est souvent nécessaire de décaler les rangs.

On remarque tout de suite deux choses :
- il y a un valeur pour laquelle on ne fait aucun appel
- dans les autres cas, quand on fait un appel, l'argument $n-1$ diminue strictement.

Cela permet d'être sûr qu'on ne va pas s'arrêter indéfiniment. On verra dans la suite comment on peut s'en assurer pour des fonctions plus complexes.

Quand on évalue `fact 5` on va effectuer une série d'appels :



Pour calculer la valeur il faut descendre le long des appels récursifs jusqu'à tomber sur un cas de base puis remonter.

I.2 Implémentation pratique

I.2.i Que se passe-t-il quand on appelle une fonction ?

Le principe de définition et d'appels de fonctions est à la base de la programmation structurée : au lieu d'avoir une longue succession d'instructions ou d'expressions, on les regroupe au sein de fonction pour s'organiser. Cela permet également de *factoriser* du code : au lieu de dupliquer, on regroupe le même code au sein d'une fonction.

La question qu'on se pose ici est celle du mécanisme permettant une telle exécution du code. On pourrait naïvement penser qu'appeler une fonction serait absolument équivalent à copier son corps :

Python
```
def f(x) :
    y = x * x
    return x + y

a = 3
b = f(a)
print(b)
```

Python
```
a = 3
# Début de f
x = a
y = x * x
# Fin de f et valeur de retour
b = x + y
```

Au début de l'informatique, c'était effectivement la manière dont on appelait les fonctions. Mais on peut se rendre compte facilement du problème en changeant noms de variables :

Python
```
def f(x) :
    a = x * x
    return x + a

a = 3
b = f(a)
print(b)
```

Python
```
a = 3
# Début de f
x = a
a = x * x
# Fin de f et valeur de retour
b = x + a
```

Ici, l'appel de la fonction a changé la valeur de `a` car `a` était aussi une variable définie dans `f`. Il est nécessaire d'avoir un mécanisme sophistiqué permettant de sauvegarder le contexte d'exécution d'une fonction et de le restaurer quand l'appel à une fonction est terminé.

Comme les fonctions vont elles-mêmes appelés d'autres fonctions, on ne peut pas se contenter de ne garder qu'un seul contexte. La bonne solution est d'utiliser ce qu'on appelle une **pile d'exécution**. Avant d'évaluer une fonction, on empile son contexte sur la pile, quand elle a terminé, on le dépile. On reverra plus tard la structure de donnée **pile** mais, pour le moment, on peut se l'imaginer comme une pile d'assiettes à laver : au départ, elle

est vide. Chaque fois qu'on empile une assiette, elle grandit. Quand on veut laver une assiette, on prend nécessairement la dernière assiette empilée.

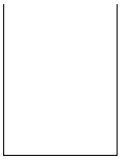
I.2.ii Un exemple

Si on considère le programme :

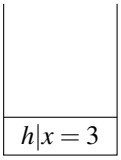
Python
```
def f() :
    return 4

def g(x) :
    v = x+f()
    return v

def h(x) :
    v = g(x+1) ** 2
    return v

h(3)
```

Il va s'exécuter ainsi :

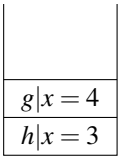
- Au départ le pile d'exécution est vide :



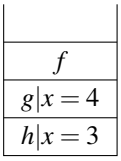
- Ligne 12 : on appelle h, on crée donc sur la pile une entrée pour h.



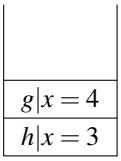
- Ligne 9 : on appelle g depuis h, on crée donc sur la pile une entrée pour g. On remarque notamment que x n'aura pas le même sens et la même valeur dans g que dans h.



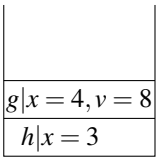
- Ligne 5 : on appelle f, on crée donc sur la pile une entrée pour f.



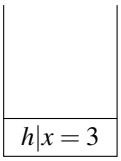
- Ligne 2 : on sort de f en renvoyant la valeur 4. On dépile ainsi l'environnement de f.



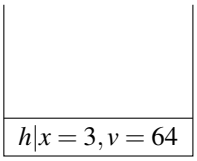
- Ligne 5 : l'environnement de g est modifié par la valeur de retour de f qui introduit une variable v.



- Ligne 6 : on sort de g en renvoyant la valeur 8. On dépile ainsi l'environnement de g.



- Ligne 9 : on définit une variable v locale à h avec la valeur de retour de g :



- Ligne 10 : l'appel de h est fini, on le dépile.
- Ligne 12 : on a recupéré la valeur de retour 64, on l'affiche avec un appel à `print` (qui sera aussi empilé).

I.2.iii Pour une fonction récursive

Le mécanisme mis en place est suffisament robuste pour permettre à une fonction de s'appeler elle-même. En effet, le fait d'avoir bien separé les environnements d'exécution assure qu'il n'y aura pas de problèmes.

Si on reprend l'exemple vu au dessus de l'évaluation de $5!$, on remarque que lorsqu'on descend le long des appels, on empile, et lorsqu'on remonte, on dépile.

I.3 Programmer en récursif

I.4 Récursivité croisée

II L'arbre d'appels

II.1 Définition

II.2 Complexité en nombre d'appels

II.3 Terminaison

III Récursivité terminale

III.1 Présentation

III.2 Optimisation

III.3 Techniques

IV Types inductifs et induction structurelle

IV.1 Définition naïve des types inductifs