

Principe de non-mutabilité

Principalement en programmation fonctionnelle (mais pas uniquement), on travaille avec des données **non mutables** : une fois une donnée construite, il est **impossible de la modifier**.

Corollaire à cela : étant donné un objet **x**, pour obtenir un objet **x'** calculé à partir de **x**, il faut **reconstruire x'**.

La non-mutabilité des données présente beaucoup d'avantages :

- 1 écriture de programmes d'avantage facilitée et sécurisée ;
- 2 démonstration de correction de programmes facilitée ;
- 3 gain de place mémoire.

Ces trois avantages s'appuient sur le fait que plusieurs grosses données peuvent **partager** des sous-données en commun, **sans aucune interférence**.

Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

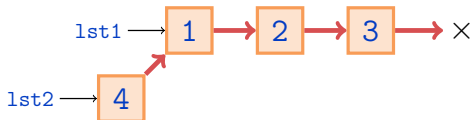
Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;  
# let lst2 = 4 :: lst1;;
```

La 1^{re} crée une liste (trois cellules) en mémoire :



La 2^e ne crée qu'une seule cellule et **partage** les trois précédentes :



Le cas des listes

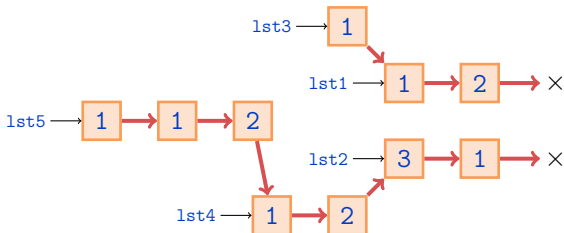
Considérons la fonction

```
let rec concatener lst1 lst2 =
  match lst1 with
  | [] -> lst2
  | e :: reste -> e :: (concatener reste lst2);;
```

Elle permet de concaténer deux listes (fonction `append` du module `List`).

```
Considérons l'effet des phrases suivantes : # let lst3 = 1 :: lst1;;
# let lst1 = [1; 2];; # let lst4 = (concatener lst1 lst2);;
# let lst2 = [3; 1];; # let lst5 = (concatener lst3 lst4);;
```

On obtient en mémoire la configuration de partage suivante :



On souhaite implanter les **files** (piles First In, First Out) dont les éléments sont d'un type quelconque. On doit pour cela

1 définir un type à un paramètre `file`;

2 définir une constante

```
vide : 'a file
```

égale à la file vide;

3 définir une fonction

```
ancien : 'a file -> 'a
```

qui renvoie le plus ancien élément de la file;

4 définir une fonction

```
supprimer : 'a file -> 'a file
```

qui renvoie la file obtenue en supprimant son plus ancien élément;

5 définir une fonction

```
ajouter : 'a file -> 'a -> 'a file
```

qui renvoie une nouvelle file prenant en compte de l'ajout d'un élément.

Implantation directe

L'implantation directe consiste à représenter une file par une liste. Les éléments sont rangés du plus récent au plus ancien.

```
type 'a file = 'a list;;

let vide = [];;

let rec ancien f =
  match f with
  | [] -> (failwith "file vide")
  | [e] -> e
  | e :: reste -> (ancien reste);;

let rec supprimer f =
  match f with
  | [] -> (failwith "file vide")
  | [e] -> []
  | e :: reste -> e :: (supprimer reste);;

let ajouter f x =
  x :: f;;
```

En notant par n le nombre d'éléments de la file, on obtient les complexités

Fonction	Complexité en temps
ancien	$\Theta(n)$
supprimer	$\Theta(n)$
ajouter	$\Theta(1)$

Implantation astucieuse

Il existe une implantation **respectant le principe de non-mutabilité** beaucoup plus performante.

Elle se base sur l'idée suivante. Une file est représentée par deux listes **in** et **out**.

- Les éléments prêts à sortir se situent dans **out**. Ils sont rangés du plus ancien au plus récent.
- Les éléments ajoutés sont « mis en mémoire tampon » dans **in**. Ils sont rangés du plus récent au plus ancien.

Les opérations sur cette structure de données se décrivent ainsi :

- l'ajout d'un élément **e** à la file consiste à positionner **e** en tête de **in** ;
- la suppression/renvoi du plus ancien élément consiste à renvoyer et supprimer la tête de **out** si elle est non vide.

Si **out** est vide, on remplace **out** par le **miroir** de **in**, on vide **in** et on renvoie/supprime la tête de **out**.

Implantation astucieuse — exemple

Opération	in	out
vide	[]	[]
ajout(1)	[1]	[]
ajout(2)	[2 ; 1]	[]
ajout(3)	[3 ; 2 ; 1]	[]
ancien()	[]	[1 ; 2 ; 3]
ajout(4)	[4]	[1 ; 2 ; 3]
ajout(5)	[5 ; 4]	[1 ; 2 ; 3]
supprimer()	[5 ; 4]	[2 ; 3]
ancien()	[5 ; 4]	[2 ; 3]
supprimer()	[5 ; 4]	[3]
ajouter(6)	[6 ; 5 ; 4]	[3]
supprimer()	[6 ; 5 ; 4]	[]
supprimer()	[]	[5 ; 6]

Implantation astucieuse

```
type 'a file =  
  {entree : 'a list;  
    sortie : 'a list};;  
  
let vide =  
  {entree = [] ; sortie = []};;  
  
let ancien f =  
  match f.sortie with  
  | [] -> begin  
    match (List.rev f.entree) with  
    | [] -> (failwith  
              "file vide")  
    | e :: _ -> e  
  end  
  | e :: _ -> e;;  
  
let supprimer f =  
  match f.sortie with  
  | [] -> begin  
    match (List.rev f.entree) with  
    | [] -> (failwith  
              "file vide")  
    | _ :: reste ->  
      {entree = [] ;  
        sortie = reste}  
  end  
  | _ :: reste ->  
    {f with sortie = reste};;  
  
let ajouter f x =  
  {f with entree = x :: f.entree};;
```

Cette implantation est plus efficace que la précédente.

Elle est même strictement plus efficace : les trois opérations ont une complexité en temps en $\Theta(1)$ sauf lorsque **out** est vide, ce qui demande pour **ancien** et **supprimer** une mise à jour en $\Theta(n)$.

Le λ -calcul a été introduit par Church en 1936.

L'objectif de Church était de fournir une formalisation alternative des mathématiques fondée sur la notion de fonction (et non plus celle d'ensemble).

Ceci a échoué car ce que Church parvint à découvrir, le λ -calcul, n'a pas un pouvoir d'expression suffisant.

En revanche, le λ -calcul a le même pouvoir d'expression que les **machines de Turing**.

Il offre ainsi un formalisme pour exprimer tout ce qui est calculable.

Le λ -calcul constitue le cœur de tous les **langages de programmation fonctionnels**.

λ -termes

Un λ -terme est

- 1 une **variable** $x, y, z, t, \text{etc.}$, ou bien
- 2 une **abstraction** $\lambda x.t$ où x est une variable et t est un λ -terme, ou bien
- 3 une **application** $s t$ où s et t sont des λ -termes.

Remarque : il s'agit d'une définition récursive.

P.ex.,

$$(\lambda x.((x y) x)) (\lambda y.\lambda x.y)$$

est un λ -terme.

Attention à l'emploi de parenthèses pour éviter les ambiguïtés. Il existe diverses conventions pour réduire leur nombre (que nous n'emploierons pas ici).

Arbres syntaxiques

Tout λ -terme t peut se représenter par son **arbre syntaxique** de la manière suivante :

- 1 si t est une variable x , l'arbre syntaxique de t est la feuille étiquetée par x ;
- 2 si t est une abstraction $\lambda x.t'$, l'arbre syntaxique de t est un nœud unaire étiqueté par λx qui possède comme fils l'arbre syntaxique de t' ;
- 3 si t est une application $t' t''$, l'arbre syntaxique de t est un nœud binaire étiqueté par $\circ$ qui possède comme fils gauche l'arbre syntaxique de t' et comme fils droit l'arbre syntaxique de t'' .

P.ex.,

- l'arbre syntaxique de $(\lambda x.((x y) x)) (\lambda y.\lambda x.y)$ est *dessiner*,
- l'arbre syntaxique de $\lambda x.((y x) (\lambda y.(x y)))$ est *dessiner*.

Variables libres / liées

Soit t un λ -terme et une occurrence d'une variable x y apparaissant.

Cette **occurrence** correspond à une **feuille** f de l'arbre syntaxique a de t .

Pour décider si cette occurrence de x dans t est libre ou liée, on considère le processus qui consiste à remonter depuis f vers la racine de a de sorte que

- si le nœud visité est étiqueté par λx , on s'arrête et on relie f à ce nœud ;
- sinon, et si ce nœud n'est pas la racine de a , on réitère ce processus ;
- sinon, on s'arrête.

Finalement, si f est liée à un nœud, on dit que l'occurrence de x est **liée** (ou encore **capturée** par le λx cible) ; sinon, on dit qu'elle est **libre**.

Variables libres / liées

Voici les propriétés libres / liées des occurrences des variables de

$$\lambda x. \lambda x. ((x (\lambda x. x)) (y (\lambda y. y))).$$

Dessiner l'arbre syntaxique.

Substitutions

Soit t un λ -terme, x une variable et s un autre λ -terme.

La **substitution** de s à x dans t consiste à remplacer chaque **occurrence libre** de x dans t par s . On note $t \star_x s$ le λ -terme ainsi obtenu.

P.ex.,

$$\blacksquare \lambda z.(x y) \star_x z z = \lambda z.((z z) y),$$

$$\blacksquare \lambda x.(x y) \star_x z z = \lambda x.(x y),$$

$$\blacksquare (z (\lambda z.(y z))) z \star_z \lambda x.(x x) = ((\lambda x.(x x)) (\lambda z.(y z))) (\lambda x.(x x)).$$

α -renommages et α -équivalence

Soient t un λ -terme, x une variable et y une variable qui n'admet aucune occurrence dans t .

L' α -renommage de x en y dans t consiste à remplacer dans t chaque λx par λy et chaque occurrence de x liée à un λx par y .

P.ex.,

- l' α -renommage de x en y du λ -terme $\lambda x.x$ est $\lambda y.y$;
- l' α -renommage de x en y du λ -terme $(x x) (\lambda x.(x z))$ est $(x x) (\lambda y.(y z))$.

Deux λ -termes t et t' sont α -équivalents s'il est possible de transformer t en t' par une suite d' α -renommages.

P.ex.,

- $\lambda x.x$ et $\lambda y.y$ sont α -équivalents ;
- $\lambda x.\lambda y.(x y)$ et $\lambda y.\lambda x.(y x)$ sont α -équivalents ;
- $\lambda x.(x y)$ et $\lambda x.(x z)$ ne sont pas α -équivalents.

β -réductions en racine

Soit t un λ -terme de la forme

$$t := (\lambda x. u) v.$$

La **β -réduction en racine** appliquée sur t consiste à transformer t en le λ -terme

$$u \star_x v.$$

Attention : aucune occurrence d'une variable libre de v ne doit être capturée dans $u \star_x v$.

De ce fait, dès que v admet une occurrence libre d'une variable x , on doit α -renommer x en une variable x' dans u , où x' est une variable qui n'admet pas d'occurrence libre dans v .

On note $t \rightarrow_{\beta'} t'$ le fait qu'un λ -terme t' puisse être obtenu à partir du λ -terme t par une β -réduction en racine.

β -réductions en racine — exemples

- $(\lambda y.y)(\lambda x.(x z)) \rightarrow_{\beta'} \lambda x.(x z)$
- $(\lambda x.(x y)) z \rightarrow_{\beta'} z y$
- $(\lambda x.\lambda y.((x x)(\lambda x.x))) (\lambda y.(y x)) \rightarrow_{\beta'} \lambda y.(((\lambda y.(y x)) (\lambda y(y x))) (\lambda x.x))$
- $(\lambda x.\lambda y.x) (z y) \not\rightarrow_{\beta'} \lambda y.(z y)$

Ici, il l'occurrence libre de y dans $(z y)$ serait capturée par le λy dans le « résultat ».

On α -renomme donc y en y' dans $(\lambda x.\lambda y.x)$ et on obtient

$$(\lambda x.\lambda y.x) (z y) = (\lambda x.\lambda y'.x) (z y) \rightarrow_{\beta'} \lambda y'.(z y).$$

β -réductions

Soit t un λ -terme.

On appelle **sous-terme** de t tout λ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t .

P.ex., le λ -terme $x (\lambda x. \lambda y. (y y))$ possède comme sous-termes

$$x, y, y y, \lambda y. (y y), \lambda x. \lambda y. (y y), x (\lambda x. \lambda y. (y y)).$$

La **β -réduction** appliquée sur t consiste à appliquer une β -réduction en racine sur l'un de ses sous-termes.

On note $t \rightarrow_{\beta} t'$ le fait qu'un λ -terme t' puisse être obtenu à partir de t par une β -réduction.

On note $t \rightarrow_{\beta^*} t'$ le fait qu'un λ -terme t' puisse être obtenu à partir de t par une suite de β -réductions.

β -réductions — exemples

- Voici une suite de β -réductions :

$$\begin{aligned} ((\lambda x.(x\ y))\ y)\ ((\lambda y.y)\ ((\lambda x.x)\ x)) &\rightarrow_{\beta} (y\ y)\ ((\lambda y.y)\ ((\lambda x.x)\ x)) \\ &\rightarrow_{\beta} (y\ y)\ ((\lambda y.y)\ x) \\ &\rightarrow_{\beta} (y\ y)\ x. \end{aligned}$$

- En partant d'un même λ -terme, on peut appliquer plusieurs suites de β -réductions différentes :

$$\begin{aligned} (\lambda y.y)\ ((\lambda x.x)\ x) &\rightarrow_{\beta} (\lambda y.y)\ x \\ &\rightarrow_{\beta} x, \end{aligned}$$

$$\begin{aligned} (\lambda y.y)\ ((\lambda x.x)\ x) &\rightarrow_{\beta} (\lambda x.x)\ x \\ &\rightarrow_{\beta} x. \end{aligned}$$

Ainsi dans ces deux cas, $(\lambda y.y)\ ((\lambda x.x)\ x) \rightarrow_{\beta^*} x$.

Règle de calcul

La relation de β -réduction $\rightarrow_\beta$ est une **règle de calcul**. On appelle **évaluation** le processus qui consiste à appliquer des β -réductions sur un λ -terme.

L'évaluation permet, dans la mesure du possible, à partir d'un λ -terme t complexe, d'obtenir un λ -terme t' plus simple que l'on ne peut plus β -réduire.

On appelle un tel terme un **réduit**. Intuitivement, un réduit est une **valeur**.

Nous avons observé qu'il existe, étant donné un λ -terme, plusieurs processus d'évaluation différents.

Question : est-ce que les différentes évaluations d'un même λ -terme t aboutissent à la même valeur ?

Réponse : oui, d'après le théorème de Church-Rosser.