# LANGAGES FONCTIONNELS: EXECUTION Compilation ou Interprétation?

- Pourquoi pas les mêmes méthodes que pour un langage impératif qui permet de définir des fonctions?

- Fonctions renvoyant des fonctions avec variables libres

- Evaluation paresseuse

- Evaluation partielle

- Peut-on utiliser l'absence d'effets de bord?

# Modifier les méthodes “Traditionnelles” Variables libres

Les valeurs de variables (paramètres) seront stockées sur le tas. (parce qu’elles peuvent survivre à la fonction qui les contient)
Possibilité d’optimiser dans le cas de variables simples...
La pile ne contient que des pointeurs à ces variables.

La fermeture qui représente une fonction contient des pointeurs à ces valeurs sur le tas.
(au lieu d’une liaison statique permettant de reconstruire un environnement qui n’existe plus!)

# Evaluation paresseuse

La “valeur” d’un paramètre (sur le tas) est soit une vraie valeur soit (un pointeur vers) la fermeture d’une fonction sans paramètres qui donnera(it) la vraie valeur.

La première fois que la valeur est demandée, la fonction est évaluée est son résultat remplace le pointeur vers la fermeture.

Si ce paramètre est passé à une (autre) fonction, c’est le pointeur vers cette location du tas qui est passé; donc la valeur est calculée une fois (au maximum).

Evaluation d’une “structure paresseuse” continue jusqu’au moment de connaître le constructeur utilisé et construit un enregistrement avec ce constructeur et autant de fermetures qu’il faut pour évaluer les composants.

# Evaluation partielle

Au lieu de traiter un appel tel `f x y z` comme un appel `f x` qui construit une fermeture qui est appliquée à `y` etc., on dit que `f` **est** une fonction de 3 paramètres et que si on a `f x`, c'est une évaluation partielle:

Les seules choses qu'on peut faire avec une évaluation partielle sont de la passer à une autre fonction ou de l'appliquer à un paramètre de plus.

Le résultat de l'évaluation partielle peut être représentée par une fermeture où on a déjà inséré la valeur du premier $(\cdots)$ paramètre.

# UNE METHODE SPECIFIQUE: REDUCTION D'ARBRES

- représentation d'un programme par un arbre

- Sommets sont: constants, fonctions prédéfinies, applications (@), fonctions définies par le programme ($\lambda$), constructeurs

- Réduction est le processus de remplacer un sous-arbre *application* par son "résultat": pour une fonction prédéfinie (par. ex. $+$), il faut évaluer ses paramètres (des sous-arbres évalués récursivement) et appliquer l'opération; pour une fonction définie par le programme, les paramètres ne sont pas

forcément évalués (évaluation paresseuse): remplacer les occurrences des paramètres dans le sous arbre ($\lambda$) par des copies des paramètres non-évalués
**if then else** prend aussi des paramètres non-évalués

- Choix du sommet à réduire? le premier sur le chemin le plus à gauche de la racine

# REDUCTION DE GRAPHES

Idée très simple qui rend la réduction efficace: remplacer un paramètre (identificateur) par un pointeur au paramètre (sous-arbre); ensuite ce sous-arbre va être évalué une fois (au maximum)

Conséquences:

- L'arbre devient un graphe
- L'application d'une fonction doit être **remplacée** par une **copie** de la fonction

# LE PROBLEME DES VARIABLES LIBRES

Dans la fonction $\lambda\ x.\ x + (\lambda\ y.\ y + x)(2)$ la fonction imbriquée a la variable libre $x$

Si on applique cette fonction à un paramètre comme $2 * y$, la réduction donne $2y + (\lambda\ y.\ y + 2y)(2)$

Ce problème de "capture de variables" est très sérieux pour l'implémentation du calcul-$\lambda$

Avec notre ordre de réduction, ceci peut arriver seulement si $y$ est une variable libre du programme; donc interdit!

# TRADUCTION EN (SUPER)COMBINATEURS

Réduction ?=? méthode raisonnable d'interpréter des programmes fonctionnels?

Parce que, pour chaque fonction définie par le programme, on doit chercher dans le corps pour trouver les occurrences des paramètres afin de savoir où mettre les pointeurs aux sous-arbres

(Pourquoi pas le faire une fois pour toujours? l'évaluation partielle peut donner de nouvelles fonctions avec de nouveaux corps!)

Une solution: traduire tout le programme en fonctions prédéfinies: ensuite il existe un nombre fixe de fonctions à exécuter et pour chacune on peut compiler du code pour faire les manipulations (de pointeurs dans le graphe)

Plusieurs façons de procéder: une méthode n'utilise que 4 combinateurs!

$$S\ f\ g\ x \quad = \quad f\ x\ (g\ x)$$

$$K\ x\ y \quad = \quad x$$

$$I\ x \quad = \quad x$$

$$Y \quad = \quad (\lambda\ h.\ (\lambda\ x.\ h(x\ x))\ (\lambda\ x.\ h(x\ x)))$$

On peut réduire n'importe quelle expression $\lambda$ (y compris celle pour $Y$) en une combinaison de $S$, $K$, $I$

$$\lambda\ x.\ x \quad \Rightarrow \quad I$$

$$\lambda\ x.\ y \quad \Rightarrow \quad K\ y \qquad (x\ \neq\ y)$$

$$\lambda\ x.\ E_1\ E_2 \quad \Rightarrow \quad S\ (\lambda\ x.\ E_1)(\lambda\ x.\ E_2)$$

# RECURSION!

Comment définir des fonctions récursives par des expressions $\lambda$ non-récursives??

Une fonction récursive (par ex. *factoriel*) peut être définie comme le point fixe d'une fonction:

$$Bizarre\,f \;=\; \lambda\, n.if\; n \;=\; 0\; then\; 1\; else\; n{*}f(n{-}1)$$

Donc on peut écrire $factoriel \;=\; FIXE\; Bizarre$ où $FIXE$ a la propriété $FIXE\; x \;=\; x(FIXE\; x)$.

$Y$ possède cette propriété!!

$$\begin{aligned} Y\ H &= (\lambda\ x.\ H(x\ x))\ (\lambda\ x.\ H(x\ x)) \\ &= H((\lambda\ x.\ H(x\ x)(\lambda\ x.\ H(x\ x))) \\ &= H(Y\ H) \end{aligned}$$

(En fait on peut prendre $Y\ H \;=\; H(Y\ H)$ comme définition de $Y$ ou comme la règle de réduction à appliquer aux applications de $Y$ pour donner une implémentation plus rapide.)