Evaluation d'une application de fonction (rappel)

Une règle simple : On évalue une application de fonction $f(e_1, ..., e_n)$

- en évaluant les expressions $e_1, ..., e_n$ en les valeurs $v_1, ..., v_n$, puis
- en remplaçant l'application avec le corps de la fonction f, dans lequel
- les paramètres effectifs $v_1, ..., v_n$ viennent remplacer les paramètres formels de f.

On peut formaliser cela comme une *réécriture du programme lui-même*:

$$\textbf{def}\ f\ (x_1, ..., x_n) = B\ ;\ ...\ f\ (v_1, ..., v_n)$$
$$\rightarrow$$
$$\textbf{def}\ f\ (x_1, ..., x_n) = B\ ;\ ...\ [v_1/x_1, ..., v_n/x_n]\ B$$

Ici, $[v_1/x_1, ..., v_n/x_n]\ B$ est une notation pour B dans lequel toutes les occurences de x_i ont été remplacées par v_i.

$[v_1/x_1, ..., v_n/x_n]$ est appelé une *substitution*.

Exemple de réécriture :

On considère *gcd* :

```
def gcd(a: int, b: int): int = if (b == 0) a else gcd(b, a % b)
```

gcd(14, 21) s'évalue alors comme suit :

```
        gcd(14, 21)
→       if (21 == 0) 14 else gcd(21, 14 % 21)
→       if (False) 14 else gcd(21, 14 % 21)
→       gcd(21, 14 % 21)
→       gcd(21, 14)
→       if (14 == 0) 21 else gcd(14, 21 % 14)
→ →     gcd(14, 21 % 14)
→       gcd(14, 7)
→       if (7 == 0) 14 else gcd(7, 14 % 7)
→ →     gcd(7, 14 % 7)
→       gcd(7, 0)
→       if (0 == 0) 7 else gcd(0, 7 % 0)
→ →     7
```

Un autre exemple de réécriture :

On considère *factorial* :

$$\textbf{def}\ factorial(n\colon int)\colon int = \textbf{if}\ (n == 0)\ 1\ \textbf{else}\ n * factorial(n-1)$$

factorial(5) se réécrit alors comme suit :

$$\begin{array}{ll}
 & factorial(5) \\
\rightarrow & \textbf{if}\ (5 == 0)\ 1\ \textbf{else}\ 5 * factorial(5-1) \\
\rightarrow & 5 * factorial(5-1) \\
\rightarrow & 5 * factorial(4) \\
\rightarrow \dots \rightarrow & 5 * (4 * factorial(3)) \\
\rightarrow \dots \rightarrow & 5 * (4 * (3 * factorial(2))) \\
\rightarrow \dots \rightarrow & 5 * (4 * (3 * (2 * factorial(1)))) \\
\rightarrow \dots \rightarrow & 5 * (4 * (3 * (2 * (1 * factorial(0))))) \\
\rightarrow \dots \rightarrow & 5 * (4 * (3 * (2 * (1 * 1)))) \\
\rightarrow \dots \rightarrow & 120
\end{array}$$

Quelles différences y a-t-il entre les deux séquences de réécriture ?

Récursivité terminale

Remarque d'implantation : Si une fonction s'appelle elle-même dans sa dernière action, le bloc d'activation (stack frame) de la fonction peut être réutilisé. C'est ce qu'on appelle la "récursivité terminale" (tail recursion).

⇒ Les fonctions récursives terminales sont des processus itératifs.

De façon générale, si la dernière action d'une fonction consiste en un appel d'une autre fonction (qui peut être la même), un seul bloc d'activation suffit pour les deux fonctions. De tels appels sont appelés "appels terminaux" (tail calls).

Exercice: Concevoir une version récursive terminale de *factorial*.

Définitions de valeur

- Une définition

  def *f* = *expr*

  introduit *f* comme un nom pour l'*expression* *expr*.
- *expr* sera évaluée à chaque fois que *f* sera utilisée.
- Autrement dit, ***def*** *f* introduit une fonction sans paramètre.
- Par comparaison, une définition de valeur

  val *x* = *expr*

  introduit *x* comme un nom pour la *valeur* de l'expression *expr*.
- *expr* sera évaluée une seule fois, au point de définition de la valeur.

Exemple:

```
> val x = 2
val x: int = 2
> val y = square(x)
val y: int = 4
> y
4
```

Exemple:

```
> def loop: int = loop
def loop: int
> val x: int = loop          (boucle infinie)
```

Fonctions d'ordre supérieur

Les langages fonctionnels traitent les fonctions comme des "valeurs de première classe".

Cela signifie que, comme n'importe quelle autre valeur, une fonction peut être passée en paramètre et retournée comme résultat.

Cela fournit un moyen flexible pour la composition de programmes.

Les fonctions qui prennent d'autres fonctions en paramètres ou qui en retournent comme résultats sont appelées fonctions "d'ordre supérieur".

Exemple :

Sommer les entiers compris entre a et b :

```
def sumInts(a: int, b: int): double =
  if (a > b) 0 else a + sumInts(a + 1, b)
```

Sommer les cubes de tous les entiers compris entre a and b :

```
def cube(x: int): double = x * x * x
def sumCubes(a: int, b: int): double =
  if (a > b) 0 else cube(a) + sumCubes(a + 1, b)
```

Sommer les inverses des entiers compris entre a and b :

```
def sumReciprocals(a: int, b: int): double =
  if (a > b) 0 else 1.0 / a + sumReciprocals(a + 1, b)
```

Ce ne sont que des cas particuliers de $\sum_a^b f(n)$ pour différentes valeurs de f.

Peut-on factoriser le schéma commun ?

Sommer avec une fonction d'ordre supérieure

On définit :

```
def sum(f: int => double, a: int, b: int): double = {
  if (a > b) 0
  else f(a) + sum(f, a + 1, b)
}
```

On peut alors écrire :

```
def sumInts(a: int, b: int): double = sum(id, a, b);
def sumCubes(a: int, b: int): double = sum(cube, a, b);
def sumReciprocals(a: int, b: int): double = sum(reciprocal, a, b);
```

où

```
def id(x: int): double = x;
def cube(x: int): double = x * x * x;
def reciprocal(x: int): double = 1.0/x;
```

Le type *int ⇒ double* est le type des fonctions qui prennent un argument de type *int* et renvoient un résultat de type *double*.

Fonctions anonymes

- La paramétrisation par les fonctions amène à créer beaucoup de petites fonctions.
- Parfois il est lourd de devoir définir ces fonctions en utilisant ***def***.
- Une notation plus courte fait appel aux *fonctions anonymes*.
- Exemple : la fonction qui élève son argument au cube s'écrit

  ```
  x: int => x * x * x
  ```

  Ici, x: *int* est le paramètre de la fonction, et x * x * x est son corps.
- Le type du paramètre peut être omis si il est évident (pour le compilateur) d'après le contexte.
- S'il y a plusieurs paramètres, ils doivent être mis entre parenthèses. Exemple :

  ```
  (x: int, y: int) => x + y
  ```

Les fonctions anonymes sont du sucre syntaxique

- De manière générale $(x_1 : T_1, ..., x_n : T_n) \Rightarrow E$ définit une fonction qui associe aux paramètres $x_1, ..., x_n$ le résultat de l'expression E (où E peut faire référence à $x_1, ..., x_n$).
- Une fonction anonyme $(x_1 : T_1, ..., x_n : T_n \Rightarrow E)$ peut toujours être exprimée en utilisant ***def*** comme suit :

  $\{$ ***def*** $f\ (x_1 : T_1, ..., x_n : T_n) = E\ ;\ f\ \}$

  où f est un nom "frais" (pas encore utilisé dans le programme).
- On dit aussi que les fonctions anonymes sont du "sucre syntaxique".

Sommation avec fonctions anonymes

Maintenant on peut écrire de façon plus courte :

> ***def*** *sumInts* (*a*: *int*, *b*: *int*): *double* = *sum* (*x* ⇒ *x*, *a*, *b*);
> ***def*** *sumCubes* (*a*: *int*, *b*: *int*): *double* = *sum* (*x* ⇒ *x* ∗ *x* ∗ *x*, *a*, *b*);
> ***def*** *sumReciprocals* (*a*: *int*, *b*: *int*): *double* = *sum* (*x* ⇒ *1.0*/*x*, *a*, *b*);

Peut-on faire encore mieux ?

Indice : a, b apparaissent partout et ne semblent pas faire partie de combinaisons intéressantes. Peut-on s'en débarrasser ?

Currification

Réécrivons *sum* comme suit.

```
def sum (f: int => double) = {
  def sumF (a: int, b: int): double =
    if (a > b) 0
    else f(a) + sumF(a + 1, b);
  sumF
}
```

- *sum* est maintenant une fonction qui retourne une autre fonction, plus précisément la fonction de somme spécialisée *sumF* qui applique la fonction et somme les résultats. On peut alors définir :

```
def sumInts = sum (x => x);
def sumCubes = sum (x => x * x * x);
def sumReciprocals = sum (x => 1.0/x);
```

- Ces fonctions peuvent être appliquées comme les autres fonctions :

```
> sumCubes (1, 10) + sumReciprocals (10, 20)
```


Application currifiée

Comment appliquer une fonction qui retourne une fonction ?

Exemple :

```
> sum (cube) (1, 10)
3025.0
```

- *sum* (*cube*) applique *sum* à *cube* et retourne la “fonction de somme des cubes” (*sum* (*cube*) est donc équivalent à *sumCubes*).
- Cette fonction est ensuite appliquée aux arguments (1, 10).
- Par conséquent, l'application de fonction associe à gauche :

```
sum (cube) (1, 10)  ==  (sum (cube)) (1, 10)
```

Définition currifiée

La définition de fonctions retournant des fonctions est si utile en PF qu'il existe une syntaxe spéciale.

Par exemple, la définition suivante de *sum* est équivalente à la précédente, mais plus courte :

```
def sum (f: int => double) (a: int, b: int): double =
  if (a > b) 0
  else f(a) + sum (f) (a + 1, b)
```

De façon générale, une définition de fonction currifiée

def f $(args_1)$ ... $(args_n) = E$

où $n > 1$, est équivalente à

def f $(args_1)$ ... $(args_{n-1}) =$ (**def** g $(args_n) = E\ ;\ g$)

où g est un identificateur "frais". Ou bien, en plus court :

def f $(args_1)$ ... $(args_{n-1}) = (\ args_n \Rightarrow E\ )$

En répétant n fois le processus

def f $(args_1)$ ... $(args_{n-1})$ $(args_n) = E$

devient équivalent à

def $f = (args_1 \Rightarrow (\ args_2 \Rightarrow \ldots (\ args_n \Rightarrow E\ ) \ldots))$

- Ce style de définition et d'application de fonction est appelé *currification* d'après son instigateur, Haskell B. Curry, un logicien du 20ième siècle.
- En réalité, l'idée remonte à Schönfinkel, mais le mot "currifié" l'a emporté (peut-être parce que "schönfinkelifié" est plus difficile à prononcer).

Types des fonctions

Question: Etant donnée

```
def sum(f: int => double)(a: int, b: int): double = ...
```

Quel est le type de *sum* ?

Remarquons que les types fonctionnels associent à *droite*. C.-à-d. que

$$int \Rightarrow int \Rightarrow int$$

est équivalent à

$$int \Rightarrow (int \Rightarrow int)$$

Exercices :

1. La fonction *sum* utilise une récursivité linéaire. Pouvez-vous en écrire une version récursive terminale en remplissant les ?? ?

```
def sum(f: int => double)(a: int, b: int): double = {
  def iter (a, result) = {
    if (??) ??
    else iter (??, ??)
  }
  iter (??, ??)
}
```

2. Ecrire une fonction *product* qui calcule le produit des valeurs d'une fonction pour les points d'un intervalle donné.

3. Ecrire *factorial* en termes de *product*.

4. Pouvez-vous écrire une fonction encore plus générale, qui généralise à la fois *sum* et *product* ?

Trouver les points fixes d'une fonction

- Un nombre x est appelé un *point fixe* d'une fonction f si

  $f(x) = x$

- Pour certaines fonctions f on peut localiser le point fixe en commençant avec une estimation de départ puis en appliquant f de façon répétée.

  $x, f(x), f(f(x)), f(f(f(x))), \ldots$

  jusqu'à ce que la valeur ne varie plus (ou que la variation soit suffisamment petite).

Cela conduit à la fonction de recherche de point fixe suivante :

```
val tolerance = 0.0001;
def isCloseEnough(x: double, y: double) = abs((x - y) / x) < tolerance;
def fixedPoint(f: double => double)(firstGuess: double) = {
  def iterate(guess: double): double = {
    val next = f(guess);
    if (isCloseEnough(guess, next)) next
    else iterate(next)
  }
  iterate(firstGuess)
}
```

Retour aux racines carrées

Voici une *spécification* de la fonction *sqrt*.

$$\begin{aligned} sqrt(x) &= \text{le nombre } y \text{ tel que } \; y * y = x \\ &= \text{le nombre } y \text{ tel que } \; y = x \,/\, y \end{aligned}$$

Par conséquent, $sqrt(x)$ est un point fixe de la fonction $(y \Rightarrow x \,/\, y)$.

Cela suggère de calculer $sqrt(x)$ par itération vers un point fixe :

```
def sqrt(x: double) =
  fixedPoint(y => x / y)(1.0)
```

Malheureusement, ça ne converge pas. Equipons la fonction *fixedpoint* d'une instruction d'affichage de manière à suivre la valeur courante de *guess* :


```
def fixedPoint(f: double => double)(firstGuess: double) = {
  def iterate(guess: double): double = {
    val next = f(guess);
    java.lang.System.out.println(next);
    if (isCloseEnough(guess, next)) next
    else iterate(next)
  }
  iterate(firstGuess)
}
```

sqrt(2) produit alors :

```
  2.0
  1.0
  2.0
  1.0
  2.0
  ...
```

Une manière de contrôler de telles oscillations est d'empêcher l'estimation de trop varier. On y arrive en *moyennant* les valeurs successives de la séquence d'origine :

```
> def sqrt(x: double) = fixedPoint(y => (y + x / y) / 2)(1.0)
> sqrt(2.0)
  1.5
  1.4166666666666665
  1.4142156862745097
  1.4142135623746899
  1.4142135623746899
```

En fait, si l'on "déplie" la fonction *fixedPoint* on retrouve exactement la fonction racine carrée de la semaine dernière.

Fonctions comme valeurs de retour

- Les exemples précédents ont montré que le pouvoir d'expression d'un langage est considérablement augmenté si on peut passer les fonctions en arguments.
- L'exemple suivant montre que les fonctions retournant des fonctions peuvent aussi être très utiles.
- On considère à nouveau l'itération vers un point fixe.
- On commence par observer que $\sqrt(x)$ est un point fixe de la fonction $y \Rightarrow x \,/\, y$.
- Puis on fait converger l'itération en moyennant les valeurs successives.
- Cette technique de *stabilisation par la moyenne* est assez générale pour mériter d'être abstraite dans une fonction.

```
def averageDamp(f: double => double)(x: double) = (x + f(x)) / 2
```

- Utilisant *averageDamp*, on peut reformuler la fonction racine carrée comme suit.

    def *sqrt* (*x*: *double*) = *fixedPoint* (*averageDamp* (*y* ⇒ *x*/*y*)) (*1.0*)

- Cela exprime les éléments de l'algorithme aussi clairement que possible.

Exercice: Ecrire une fonction pour les racines cubiques en utilisant *fixedPoint* et *averageDamp*.

Résumé

- On a vu la semaine dernière que les fonctions sont des abstractions essentielles, car elles nous permettent d'introduire des méthodes générales de calcul comme des éléments explicites et nommés dans notre langage de programmation.
- Cette semaine on a vu que ces abstractions peuvent être combinées grâce aux fonctions d'ordre supérieur pour créer de nouvelles abstractions.
- En tant que programmeur, on doit guetter les opportunités d'abstraire et de réutiliser.
- Le plus haut niveau d'abstraction n'est pas toujours le meilleur, mais il est important de connaître les techniques d'abstraction, de manière à les utiliser quand c'est approprié.

Eléments du langage vus jusqu'à présent

- Nous avons vu les éléments du langage pour exprimer les types, les expressions et les définitions.
- On donne ci-dessous leur syntaxe non contextuelle sous la forme Backus-Naur étendue (EBNF), où '|' dénote une alternative, [...] une option (0 ou 1), et {...} une répétition (0 ou plus).

Types :

Type	= *SimpleType* | *FunctionType*
FunctionType	= *SimpleType* '⇒' *Type* | '(' [*Types*] ')' '⇒' *Type*
SimpleType	= *byte* | *short* | *char* | *int* | *long* | *double* | *float*
	| *boolean* | *String*
Types	= *Type* {',' *Type*}

Un type peut être :

- un types numérique: *int*, *double* (et *byte*, *short*, *char*, *long*, *float*),
- le type *boolean* avec les valeurs *true* et *false*,
- le type *String*,
- un type fonctionnel.

Expressions:

Expr	= *InfixExpr* | *FunctionExpr* | ***if*** ‘(’ *Expr* ‘)’ *Expr* ***else*** *Expr*
InfixExpr	= *PrefixExpr* | *InfixExpr Operator InfixExpr*
Operator	= *ident*
PrefixExpr	= [‘+’ | ‘−’ | ‘!’ | ‘~’] *SimpleExpr*
SimpleExpr	= *ident* | *literal* | *SimpleExpr* ‘.’ *ident* | *Block*
FunctionExpr	= *Bindings* ‘⇒’ *Expr*
Bindings	= *ident* [‘:’ *SimpleType*] | ‘(’ [*Binding* {‘,’ *Binding*}] ‘)’
Binding	= *ident* [‘:’ *Type*]
Block	= ‘{’ {*Def* ‘;’} *Expr* ‘}’

Une expression peut être :

- un identificateur tel que *x*, *isGoodEnough*,
- un littéral, comme *0*, *1.0*, *”abc”*,
- une application de fonction, comme $sqrt(x)$,
- une application d’opérateur, comme $-x$, $y + x$,
- une séléction, comme *java.lang.System.out.println*,
- une expression conditionnelle, comme ***if*** $(x < 0)\ -x$ ***else*** x,
- un bloc, comme { **val** $x = abs(y)$; $x * 2$ }
- une fonction anonyme, comme $(x \Rightarrow x + 1)$.

Définitions:

$$\begin{array}{lll}
\mathit{Def} & = & \mathit{FunDef} \;\mid\; \mathit{ValDef} \\
\mathit{FunDef} & = & \textbf{\textit{def}}\ \mathit{ident}\ [\text{'('}\ [\mathit{Parameters}]\ \text{')'}]\ [\text{':'}\ \mathit{Type}]\ \text{'='}\ \mathit{Expr} \\
\mathit{ValDef} & = & \textbf{val}\ \mathit{ident}\ [\text{':'}\ \mathit{Type}]\ \text{'='}\ \mathit{Expr} \\
\mathit{Parameter} & = & [\textbf{def}]\ \mathit{ident}\ \text{':'}\ \mathit{Type} \\
\mathit{Parameters} & = & \mathit{Parameter}\ \{\text{','}\ \mathit{Parameter}\}
\end{array}$$

Une définition peut être :

- une définition de fonction comme ***def*** $square(x\colon int) = x * x$
- une définition de valeur comme **val** $y = square(2)$