LINFO1104 : TP5 Lambda-Calcul

Le lambda calcul est un système formel sur lequel se base la programmation fonctionnelle. C'est un langage qui est Turing complet, ce qui veut dire qu'il peut faire tous les calculs qu'un langage de progammation "normal" peut faire. Il a l'avantage d'être petit et d'avoir une syntaxe simple. Le lambda-calcul définit un ensemble d'expressions comme étant valide. Ces expressions sont appelées des lambda-termes et se divisent en trois catégories:

variables : une variable x est lambda-terme.

abstraction : Si m est un lambda-terme et x est une variable, alors $(\lambda x.m)$ est un lambda-terme. Dans ce cas-ci, m est appelé le corps l'abstraction et x en est l'argument.

applications : Si m et n sont des lambda-termes, alors (mn) est un lambda-terme

Seules ces règles sont utilisées pour définir ce qu'est une expression valide en lambda-calcul. Pour clarifier les expressions, il est courant en lambda-calcul d'omettre certaines parenthèses et de compresser les séquences d'abstractions.

- Les parenthèses les plus à l'extérieur sont retirées: (mn) devient mn.
- Les applications sont effectuées en faisant l'association par la gauche: $((mn)p)$ devient mnp.
- Le corps d'une expression s'étend autant que possible vers la droite. $\lambda x.mn$ signifie $\lambda x.(mn)$ et non $(\lambda x.m)n$
- Une séquence d'abstraction peut-être abrégée. $\lambda x.\lambda y.\lambda z.n$ est abrégé en $\lambda xyz.n$.

Définition formelle du lambda-calcul

1. **Validité d'une expression.** Pour les expressions suivantes, identifiez celles qui sont valides.

   - $\lambda x.xyz$
   - $\lambda x.\lambda y$
   - m
   - $x\lambda wy.y$
   - $x\lambda$
   - $\lambda\lambda xz.zx$
   - $(mnop)(qrst)vw\lambda xyz.zxy$

2. **Variable libre et variable liée.** En lambda-calcul, le concept de variable libre et variable liée existe aussi. Quand une variable fait partie des arguments d'une abstraction, on dit qu'elle est liée à cette abstraction. Pour les expressions suivantes, identifiez pour chaque abstraction leur corps et indiquez pour chaque variable x à quelle abstraction est-elle liée.

   - $\lambda x.\lambda y.x$
   - $\lambda x.\lambda x.x$
   - $\lambda x.x\lambda y.x$
   - $\lambda x.x\lambda x.x$
   - $\lambda z.x\lambda y.x$
   - $\lambda z.x\lambda x.x$

3. **Renommage de variable ($\alpha - conversion$).** Pour chaque ligne tableau suivant, indiquez si la paire d'expression est α-equivalent.

$\lambda a.\lambda b.\, abb$	$\lambda b.\lambda a.baa$
$\lambda a.\lambda b.\lambda a.bb$	$\lambda i.\lambda j.jji$
$\lambda x.x\lambda y.x$	$\lambda e.e\lambda f.f$
$\lambda x.x\lambda y.x$	$\lambda e.e\lambda f.e$

4. **Réduction d'expression (β).** Reduisez au maximum les expressions en utilisant la β-reduction.

   - $(\lambda x.xx)y$
   - $(\lambda x.axxa)y$
   - $(\lambda x.(\lambda z.zx)q)y$
   - $(\lambda x.x((\lambda z.zx)(\lambda x.bx)))y$
   - $(\lambda m.m)(\lambda n.n)(\lambda c.cc)(\lambda d.d)$
   - $\lambda z.x\lambda x.x$

5. **Reduction d'expression (η).** Réduisez au maximum les expressions suivantes en utilisant l'η-reduction.

- $\lambda x.(\lambda y.y)x$
- $\lambda x.(\lambda y.(\lambda z.p)y)x$
- $\lambda x.(\lambda y.(\lambda z.z))x$
- $\lambda x.(\lambda y.yx)p$
- $(\lambda f.fx)(\lambda y.gy)$

On dit d'une expression qui n'est pas réductible par β-réduction ou par η-réduction est en *forme normale* $\beta - eta$.

Propriété de Church-Rosser

6. Le théorème de Church-Rosser déclare que lorsqu'on applique des règles de réduction à des termes du lambda-calcul, l'ordre dans lequel les réductions sont choisies ne fait pas de différence au résultat final.

   S'il y a deux réductions ou séquences de réductions distinctes qui peuvent être appliquées au même terme, alors il existe un terme qui peut être atteint à partir des deux résultats, en appliquant des séquences (éventuellement vides) de réductions supplémentaires.

   L'expression suivante peut être réduite de deux façons : $(\lambda x.\, \lambda y.\, x)\ ((\lambda x.\, x)\ y)$.

   - Simplifiez cette expression en réduisant d'abord les lambda-termes les plus à gauche possible. Refaites ensuite la même opération en réduisant d'abord les lambda-termes les plus à droite possible. Que remarquez-vous ?
   - Refaites la même procédure pour $(\lambda x.\, \lambda y.\, x\ y)\ (\lambda z.\, z)\ (\lambda w.\, w)$

   Le théorème de Church-Rosser stipule que la β-réduction est confluente. Si une expression conduit à deux formes normales irréductibles, elles sont α-équivalentes (équivalentes au renommage près).

Représentation de type de donnée

7. **Arithmetiques des booléens.** Il n'y a pas de booléen ou de nombre en lambda-calcul mais il est possible de les modéliser juste avec des fonctions. Pour répresenter les booléens, on utilise une notation appelée Church-booléen:

   - $true\ :=\ \lambda x.\lambda y.x$
   - $false\ :=\ \lambda x.\lambda y.y$

   Avec ces lambda-termes, les opérateurs logiques peuvent être aussi définis par des lambda-termes, par exemple l'opérateur *and* peut être défini de la façon suivante:

   $$and\ := \lambda p.\lambda q.pqp$$

   - Définissez les opérateurs *not* et *or*.
   - Evaluez l'expression $or\ true\ false$ en utilisant les lambda-termes définis au dessus.

8. **Arithmétiques des nombres.** Pour représenter les nombres naturels, on utilise une notation appelée Church-numéral:

- $0\ :=\ \lambda f.\lambda x.x$ ou $\lambda fx.x$
- $1\ :=\ \lambda f.\lambda x.fx$ ou $\lambda fx.(fx)$
- $2\ :=\ \lambda f.\lambda x.f(fx)$ ou $\lambda fx.f(fx)$
- $3\ :=\ \lambda f.\lambda x.f(f(fx))$ ou $\lambda fx.f(f(fx)$

Un nombre n est donc représenté par une fonction à deux arguments, une fonction f et une variable x, qui execute f sur x n fois.

- Définissez une fonction *increment* qui incrémente un nombre de 1. La fonction a 3 arguments.
- La fonction *plus* est définie comme ci-dessous, en calculant *plus* 1 2, est-que le résultat est 3?
- $plus := \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)$

9. **Les listes en lambda calcul**

- $paire := \lambda a.\lambda b.\lambda f.((fa)b)$
- Définissez une fonction $first$ et $second$ qui prennent en argument une paire et renvoie respectivement le premier et le second élément d'une paire.
- Créer une $paire$ 4 2 et appliquer vos fonction pour récupérer le premier et second élément
- En utilisant les paires, définissez une fonction $list$ qui crée une liste.

La récursivité en lambda-calcul

10. **Fonction récursive**. Contrairement à un langage de programation, le lambda-calcul ne supporte pas directement la récursion en appelant une fonction f à l'intérieur de la fonction f. Le lambda-calcul utilise alors une fonction "point-fixe".

Le point fixe d'une fonction f est un élément du domaine de f, mis en correspondance avec lui-même dans f. Par exemple, "c" est le point fixe de f si $f(c) = c$. De même $f(f(...(f(c))...)) = f^n(c) = c$.

Pour la fonction $f(x) = x^2$, 0 et 1 sont les seuls points fixes de f puisque $f(0) = 0$ et $f(1) = 1$.

Nous allons utiliser ce même principe pour créer des expression lamba récursives. Il existe un combinateur qui permet de créer une version récursive d'une fonction. Ce combinateur est lui même une fonction qui retournera un point fixe pour n'importe quelle fonction que l'on lui passe :

$$Y = \lambda f.(\lambda x.f\,(x\,x))(\lambda x.f(x\,x))$$

Lorsque Y est utilisé comme "constructeur", elle crée un point fixe sur l'argument qui lui est passé.

Regardons ce qu'il se passe lorsque g est appliqué à Y :

$$\begin{aligned} Y\,g &= (\lambda f.(x.f\,(x\,x))\ (\lambda x.f(x\,x)))\ g \\ &= (\lambda x.g\,(x\,x))\ (\lambda x.g\,(x\,x)) \\ &= g\ ((\lambda x.g\,(x\,x))\ (\lambda x.g\,(x\,x))) \\ &= g\ (Y\,g) \end{aligned}$$

Si le combinateur est appliqué indéfiniment, on obtient :

$$Y\,g = g\ (Y\,g) = g\ (g\ (Y\,g)) = g\ (...g\ (Y\,g)...)$$

C'est ainsi que la récursion est créée en s'assurant que g et Y continuent à s'étendre à chaque application.

Exercice: utilisez le combinateur Y pour appliquer une fonction *add* qui additionne 1 avec 1. Pour vous aider, traduisez ce pseudo-code en lambda-calcul :

```
def add f x y =
  if is_zero y then x
  else f (succ x) (pred x)
```

- Comment traduisez-vous les fonctions défninies dans la fonction `add` ? En d'autres termes, défninissez respectivement :
  - `if c then x else y`
  - `is_zero y`
  - `succ n`
  - `pred n`
- Utilisez les résultats de vos calculs précédents pour calculer en lamba calcul :
  - `is_zero` 2
  - `pred` 2
- Continuez l'évaluation de l'expression demandée :

$$Y\ add\ 1\ 1 = ...\text{(your job)}$$

Traduire l'entièreté des fonctions en lambda calcul rendrait le calcul beaucoup trop long. Utilisez les fonctions définies à l'exercice précédent pour calculer le résultat.