1 Implémentation du lambda-calcul

1.1 Objectifs

Le but de ce TP est de réaliser une implémentation de la β-réduction dans le cadre du lambda-calcul typé présenté en cours. Il s'agit donc de réaliser une fonction qui, étant donnée une expression quelconque, renvoie la forme la plus réduite possible de cette expression[1].
Exemples :

(1) a. `beta-reduit(`$(\lambda y.(\lambda P.(P)y)\lambda x.(\text{dort})x)\text{jean}$`) =` $(\text{dort})\text{jean}$
b. `beta-reduit(`$\lambda P.\exists x(P)x$`) =` $\lambda P.\exists x(P)x$

Dans son principe général, la β-réduction consiste à réécrire un terme de la forme $(\lambda x.M)N$ (où M et N sont des termes quelconques) en le terme M dans lequel toutes les occurrences libre de x sont remplacées par N. Il faut donc être en mesure (1) de reconnaître qu'on a affaire à un redex ; (2) d'identifier les occurrences *libres* d'une variable dans une formule ; (3) de substituer une variable par un terme dans une formule, ce qui suppose (4) qu'on soit capable de faire une copie d'un terme quelconque (chaque occurence libre de la variable x est remplacée par une nouvelle occurrence du terme N). Il faut ajouter une 5$^{\text{e}}$ étape très importante : si on applique sans discernement l'algorithme rappelé plus haut, on risque de tomber sur des cas de **capture de variable**. Il faut donc ajouter au code produit la prévention de ce risque.

De la capture des variables Exemple (langage pur) : la fonction $\lambda x.\lambda y.x$ renvoie normalement une fonction constante, si on l'applique à un terme ne faisant pas intervenir la variable y : $(\lambda x.\lambda y.xz) = \lambda y.z$. Mais si le terme auquel on applique ce combinateur contient une occurrence libre de la variable y (prenons le cas le plus simple, y elle-même), alors c'est une fonction non constante que l'on obtient : $(\lambda x.\lambda y.xy) = \lambda y.y$. Si on autorise ce genre de capture, le λ-calcul n'est plus « consistant ».
Pour éviter ces captures, il faut garantir que

> les variables *liées* de M sont disjointes des variables *libres* de N.

[1] En λ-calcul pur, un tel algorithme peut très bien ne jamais se terminer (par exemple avec l'expression $(\lambda x.(x)x)\lambda x.(x)x$), mais dans notre utilisation de la version typée de ce langage, ce risque de boucle infinie n'est pas à considérer.

1.2 Structure de données

Pour réaliser ces opérations, il faut manipuler la structure syntaxique des formules. Rappelons que les formules de la logique du premier ordre peuvent être décomposées de façon unique. Voir un exemple avec la formule (2-a), à gauche de la figure 1.
La prise en compte de la λ-abstraction et de l'application fonctionnelle (ainsi que de la curryfication), le tout dans une perspective d'implémentation, conduit à une représentation plus détaillée de la structure des formules, comme illustré, pour la formule (2-b), par l'arbre de droite de la figure 1.

(2) a. $\forall x(\neg\forall y(P(y) \rightarrow A(x,y)) \leftrightarrow B(x))$
b. $\lambda P.\lambda x.(P)\exists y.((\text{man})x \wedge ((\text{love})x)y)$,

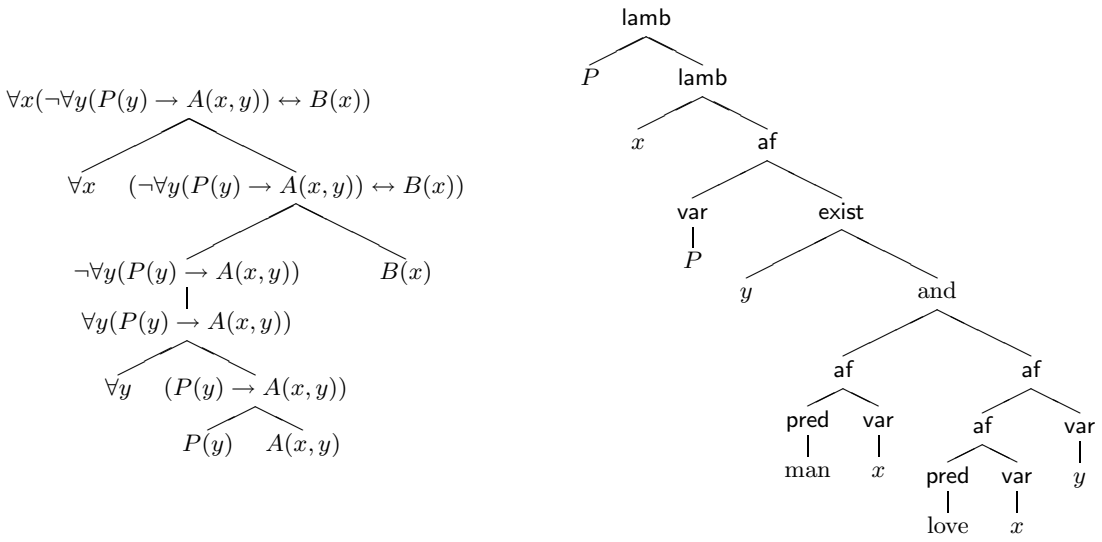

FIG. 1 – Décomposition syntaxique des formules logiques

Implémentation Une des manières les plus simples d'implémenter un arbre est de le représenter avec des listes enchâssées. Par exemple, la liste (3-a) pourrait correspondre à l'arbre donné à gauche de la figure 2. Si l'on veut que les nœuds internes portent une étiquette, on peut adopter la convention que le premier élément de chaque liste correspond à l'étiquette du nœud. Cela donne, pour l'exemple (3-b), l'arbre de droite de la figure 2.

(3) a. `(a, (b, c, (a, d)), (b, (a, c), (a, (b, c, d))))`
b. `(A, (B), (C, (A), (C), (D, A)), (B, (C), (D)))`

Pour faciliter encore la manipulation des formules, on va faire l'hypothèse que les nœuds de nos arbres sont tous représentés par un *triplet*, dont le premier membre est l'étiquette du nœud, et dont les deux autres membres sont de forme différente selon le type du nœud. Voici la convention proposée pour représenter chaque type de nœud (la notation `(...)` indique qu'un sous-terme peut occuper cette place) :

Feuilles

- variable : `(var, x, nil)`
- constante : `(cte, j, nil)`

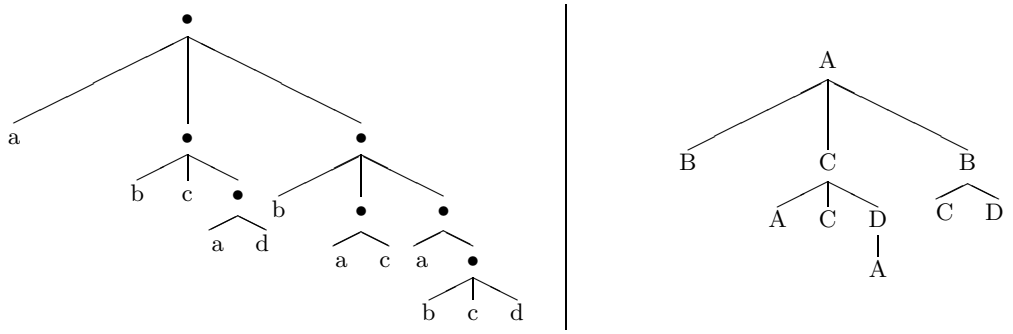

FIG. 2 – Arbres et listes enchâssées

– prédicat[2] : `(pred, sleep, nil)`

Arbres « unaires »

– négation : `(not, (...), nil)`

Arbres « binaires »

– application fonctionnelle : `(af, (...), (...))`

– opérateurs propositionnels binaires : $(\odot,$ `(...), (...))`, où $\odot \in \{\text{and}, \text{or}, \text{impl...}\}$

Arbres avec liage de variable

– lambda : `(lamb, x, (...))`

– existentiel : `(exist, x, (...))`

– universel : `(univ, x, (...))`

Avec ces conventions, il est facile de réaliser des fonctions récursives qui exploitent la structure arborescente. Par exemple une fonction d'affichage lisible des formules (« *pretty-print* ») pourrait s'écrire :

```
def pretty_print(f):
    print pretty_fmt(f)

def pretty_fmt(f):
    (r, fg, fd) = f
    if r=='var' or r=='cte' or r=='pred':
        return fg
    if r=='not':
        return '~ ' + pretty_fmt(fg)
    if r=='af':
        return "("+pretty_fmt(fg)+")"+pretty_fmt(fd)
    if r=='and':
        return "("+pretty_fmt(fg)+" & "+pretty_fmt(fd)+")"
    if r=='or':
        return "("+pretty_fmt(fg)+" | "+pretty_fmt(fd)+")"
    if r=='impl':
        return "("+pretty_fmt(fg)+" --> "+pretty_fmt(fd)+")"
    if r=='exist':
        return "exist "+fg+". "+pretty_fmt(fd)
    if r=='univ':
        return "univ "+fg+". "+pretty_fmt(fd)
    if r=='lamb':
        return "\\"+fg+". "+pretty_fmt(fd)
    return "null"
```

[2] On a la même représentation pour les prédicats, qu'ils soient unaires ou binaires, à cause de la curryfication.

1.3 Repérage des variables libres

Il s'agit d'implémenter une fonction qui, étant donnée une formule quelconque, renvoie l'ensemble des variables libre de cette formule.
On peut rappeler la définition inductive des variables libres (VL) donnée en cours à propos du langage pur, qu'il faut donc adapter ici.
Version inductive (VL = variables libres) :
- $VL(x) = \{x\}$ (où x est une variable)
- $VL(\ (t_1)t_2\ ) = VL(t_1) \cup VL(t_2)$ (où t_1 et t_2 sont des termes)
- $VL(\lambda x.t) = VL(t) \setminus \{x\}$

On implémentera aussi une fonction qui renvoie les liste des variables liées dans une formule.

1.4 Duplication d'une expression

Il est indispensable de disposer d'une fonction qui, étant donnée une formule quelconque, renvoie une copie de cette formule (qui doit être une "vraie" copie, c'est-à-dire un nouvel 'objet' informatique qui ne partage aucune référence avec l'original).
Il s'agit d'implémenter cette copie, qui peut se réaliser en `python` avec la méthode `copy()`, voire avec la méthode `deep.copy()`

1.5 Substitution d'une variable par un terme

L'étape suivante est l'implémentation d'une méthode qui, étant donnée un terme et une variable, remplace toutes les occurrences **libres** de cette variable par une copie d'un certain terme. Une version préliminaire et plus simple de cet algorithe est le renommage de variables. Dans ce dernier cas, le terme que l'on met à la place de la variable est une autre variable.

1.6 Implémentation effective de la β-réduction

On dispose maintenant de toutes les opérations nécessaires pour mettre en œuvre effectivement la β-réduction d'un terme φ :

1. Tant qu'il existe des redex dans φ : soit $\omega = (\lambda x.M)N$
2. Vérifier que les variables présentes dans M et N ne provoquent pas de capture, si oui, les renommer
3. Remplacer ω dans φ par le terme M dans lequel les occurrences libres de x ont été remplacées par (une copie de) N