Projet LRC M1

Rapport

Démonstrateur en Prolog basé sur l'algorithme des tableaux

Étudiants :
Hervé NGUYEN
Robin SOARES

Encadré par :
Colette FAUCHER

20 décembre 2023

Résumé

Dans ce projet, nous avons développé un démonstrateur en Prolog basé sur l'algorithme des tableaux pour la logique **ALC**. Le programme prend d'abord une TBox et une ABox à travers un fichier et vérifie sa validité au niveau formulation et syntaxe, puis prend en entrée une proposition de l'utilisateur afin de vérifier si cette proposition est valide dans le contexte de la ABox et la TBox. Pour obtenir ces résultats, il faut donc implémenter en Prolog l'algorithme des tableaux.

Repository GitHub : `https://github.com/TsunomakiWatamelon/M1-S1-ProjetLRC`

Table des matières

1 Etape préliminaire de vérification et de mise en forme de la Tbox et de la Abox

1.1 Correction syntaxique et sémantique

Dans cette première partie, nous vérifions la correction syntaxique et sémantique d'une Tbox et d'une Abox. Pour ce faire nous introduisons le prédicat ***concept*** :

Listing 1 – Correction syntaxique

```
concept(Concept) :- isCA(Concept).
concept(Concept) :- isCNA(Concept).
concept(anything).
concept(nothing).
concept(not(Concept)) :- concept(Concept).
concept(and(ConceptX,ConceptY)) :- concept(ConceptX),
   concept(ConceptY).
concept(or(ConceptX, ConceptY)) :- concept(ConceptX),
   concept(ConceptY).
concept(some(Role, Concept)) :- isR(Role), concept(Concept).
concept(all(Role, Concept)) :- isR(Role), concept(Concept).
```

Le prédicat ***concept*** vérifie si un terme est un concept atomique ou non atomique valide en effectuant des tests spécifiques sur sa structure. Si ***concept(Concept)*** est vrai, alors ***concept*** est considéré comme un concept syntaxiquement correct.

Listing 2 – Correction sémantique

```
getListe(CA, CNA, ID, R) :-
    setof(X, cnamea(X), CA),
    setof(X, cnamena(X), CNA),
    setof(X, iname(X), ID),
    setof(X, rname(X), R).

isCA(X)  :- getListe(CA, CNA, ID, R), member(X, CA), not(member(X,
   CNA)), not(member(X, ID)), not(member(X, R)).
isCNA(X) :- getListe(CA, CNA, ID, R), member(X, CNA),
   not(member(X, CA)), not(member(X, ID)), not(member(X, R)).
isId(X)  :- getListe(CA, CNA, ID, R), member(X, ID), not(member(X,
   CA)), not(member(X, CNA)), not(member(X, R)).
isR(X)   :- getListe(CA, CNA, ID, R), member(X, R), not(member(X,
   CA)), not(member(X, CNA)), not(member(X, ID)).
```

Les prédicats ***isCA***, ***isCNA***, ***isID*** et ***isR*** fournissent des moyens de classifier une entité en la comparant avec des listes prédéfinies de noms associés à différents types d'entités (CA, CNA, ID, R) Ils sont utilisés pour vérifier la validité sémantique des concepts, identifiants et relations dans notre TBox et ABox.

1.2 Verification de l'auto-référencement

L'auto-référencement peut introduire des ambiguïtés et compromettre la cohérence de la TBox et de l'ABox. Il est donc important de s'assurer qu'il n'y a pas d'auto-référencement. Nous introduisons donc les prédicats ***autoref*** et ***conceptAutoref*** :

Listing 3 – Verification de l'auto-référencement

```
autoref(Concept) :-
    equiv(Concept, Expression),
    (not(conceptAutoref(Concept, Expression, [])) ->
        write('Info␣:␣'), write(Concept), write("␣n'est␣pas␣
           autoreferent"), nl, false
    ; write('Info␣:␣'), write(Concept), write("␣est␣soit␣
       autor f rent␣ou␣bien␣contient␣une␣expression␣
       autor f rente"), nl, true).
conceptAutoref(Concept, Concept, _).

conceptAutoref(Concept, ConceptA, Visited) :-
    member(ConceptA, Visited);
    equiv(ConceptA, ConceptB),
    conceptAutoref(Concept, ConceptB, [ConceptA | Visited]).
conceptAutoref(Concept, not(Expression), Visited) :-
    member(not(Expression), Visited);
    conceptAutoref(Concept, Expression, [not(Expression) |
       Visited]).
conceptAutoref(Concept, and(Expression1, Expression2), Visited) :-
    member(and(Expression1, Expression2), Visited);
    conceptAutoref(Concept, Expression1, [and(Expression1,
       Expression2) | Visited]);
    conceptAutoref(Concept, Expression2, [and(Expression1,
       Expression2) | Visited]).
conceptAutoref(Concept, or(Expression1, Expression2), Visited) :-
    member(or(Expression1, Expression2), Visited);
    conceptAutoref(Concept, Expression1, [or(Expression1,
       Expression2) | Visited]);
    conceptAutoref(Concept, Expression2, [or(Expression1,
       Expression2) | Visited]).
conceptAutoref(Concept, some(R, Expression), Visited) :-
    member(some(R, Expression), Visited);
    conceptAutoref(Concept, Expression, [some(R, Expression) |
       Visited]).
conceptAutoref(Concept, all(R, Expression), Visited) :-
    member(all(R, Expression), Visited);
    conceptAutoref(Concept, Expression, [all(R, Expression) |
       Visited]).
```

Le prédicat ***autoref*** prend un concept en argument et utilise la relation d'équivalence ***equiv*** définie dans le fichier *tabox.pl* pour obtenir une expression équivalente au concept donné. Ensuite, il appelle le prédicat ***conceptAutoref*** pour vérifier s'il y a une auto-référence dans l'expression équivalente. Le prédicat ***conceptAutoref*** vérifie si le concept donné est identique à l'expression. Pour les expressions composées telles que some, not,

and, or, all, il vérifie si le concept s'auto-réfère dans au moins l'une des sous-expressions. Et pour les cas particuliers, tels que **some(_, Expression)** et **all(_, Expression)**, le concept est testé pour une auto-référence dans l'expression associée.

On peut également retrouver aussi l'utilisation d'une liste de "noeuds" visités afin de détécter si le concept de base contient un autre sous-concept autoréférent afin de ne pas rentrer dans une boucle de récursion infinie.

1.3 Mise en forme de la TBox et de la ABox

Nous devons ensuite obtenir une definition ne contenant que des concepts atomiques et la mettre sous forme normale négative.

Listing 4 – Mise en forme

```
definitionAtomique(Definition, Definition) :-
    cnamea(Definition).
definitionAtomique(Definition, Res) :- equiv(Definition, X),
   definitionAtomique(X, Res).
definitionAtomique(not(Definition), not(Res)) :-
    definitionAtomique(Definition, Res).
definitionAtomique(or(D1,D2), or(R1,R2)) :-
    definitionAtomique(D1,R1),
    definitionAtomique(D2,R2).
definitionAtomique(and(D1,D2), and(R1,R2)) :-
    definitionAtomique(D1,R1),
    definitionAtomique(D2,R2).
definitionAtomique(some(Role, Definition), some(Role, Res)) :-
    definitionAtomique(Definition, Res).
definitionAtomique(all(Role,Definition), all(Role, Res)) :-
    definitionAtomique(Definition, Res).

remplacement([], []).
remplacement([(Concept, Definition) | Reste], [(Concept,
   DefinitionTraite) | ResteTraite]) :-
    definitionAtomique(Definition, Atomique),
    nnf(Atomique, DefinitionTraite),
    remplacement(Reste, ResteTraite).
```

Le prédicat ***definitionAtomique*** nous permet d'obtenir une formule ne contenant que des concepts atomiques. Cela nous permet ensuite d'utiliser le prédicat ***remplacement*** qui va remplacer les concepts de la TBox et la ABox par des concepts composés de termes atomiques et les mettres sous forme normale négative.

Listing 5 – Traitement

```
traitement_Tbox(Initial, TBox) :- remplacement(Initial, TBox).
traitement_Abox(Initial, ABoxC) :- remplacement(Initial, ABoxC).
```

Finalement, on utilise le prédicat ***remplacement*** pour traiter la TBox et la ABox dans les prédicats ***traitement_Tbox*** et ***traitement_Abox***.

2 Saisie de la proposition à démontrer

Dans cette partie, nous allons implémenter la saisie et le traitement d'une proposition à démontrer. On cherche à ajouter la négation dans la ABox étendue.

L'utilisateur peut choisir la proposition parmi deux formes :

- Proposition de type $I : C$
- Proposition de type $C_1 \sqcap C_2 \sqsubseteq \bot$

Listing 6 – Saisie de la proposition

```
deuxieme_etape(Abi,Abi1,Tbox) :-
    saisie_et_traitement_prop_a_demontrer(Abi,Abi1,Tbox).

saisie_et_traitement_prop_a_demontrer(Abi,Abi1,Tbox) :-
    nl,write("Entrez le numero du type de proposition que vous
       voulez demontrer :"),nl,
    write("1 Une instance donnee appartient a un concept donne."),
       nl,
    write("2 Deux concepts n'ont pas d'elements en commun(ils
       ont une intersection vide)."),nl, read(R),
       suite(R,Abi,Abi1,Tbox).
```

Le prédicat ***saisie_et_traitement_prop_a_demontrer*** permet à l'utilisateur de choisir entre les deux types de propositions. Le choix *1* correspond à la proposition de type $I : C$ et le choix *2* correspond à la proposition de type $C_1 \sqcap C_2 \sqsubseteq \bot$

2.1 Proposition de type $I : C$ (1)

Listing 7 – Traitement de la proposition de type 1

```
acquisition_prop_type1(Abi, [(Inst, NCFinal) | Abi],Tbox) :-
    acquisition_type1_instance(Inst),
    (isId(Inst) -> true; write(Inst), write(" n'est pas une
       instance"), nl, false),
    acquisition_type1_concept(C),
    (concept(C) -> true; write(C), write(" n'est pas un concept"),
       nl, false), nl
    write('Info : Proposition a demontrer \"'), write(Inst),
       write(' : '), affiche_concept(C), write('\"'), nl,
    definitionAtomique(not(C), NCA),
    nnf(NCA, NCFinal).
```

Listing 8 – Saisie de l'instance et du concept pour les propositions de type 1

```
acquisition_type1_instance(Inst) :-
    nl,write("Entrez le nom de l'instance de votre proposition
       :"),nl,read(Inst).

acquisition_type1_concept(C) :-
    nl,write("Entrez le nom du concept de votre proposition
       :"),nl,read(C).
```

Le prédicat ***acquisition_prop_type1*** utilise les prédicats ***acquisition_type1_instance*** et ***acquisition_type1_concept*** qui nous permettent d'obtenir l'instance et le concept puis de verifier s'ils sont valides. Ensuite il génère la négation du concept et le mets sous forme normale négative mettant à jour l'abstraction avec la proposition de type 1.

2.2 Proposition de type $C_1 \sqcap C_2 \sqsubseteq \bot$ (2)

Listing 9 – Traitement de la proposition de type 2

```
acquisition_type2_concept(C,1) :-
    nl,write("Entrez le nom du premier concept C1 de votre
       proposition :"),nl,read(C).
acquisition_type2_concept(C,2) :-
    nl,write("Entrez le nom du deuxieme concept C2 de votre
       proposition :"),nl,read(C).
```

Le prédicat ***acquisition_type2_concept*** est utilisé dans le prédicat ***acquisition_prop_type2*** et permet à l'utilisateur de saisir un concept, soit pour le premier concept C1 si l'argument est *1*, soit pour le deuxième concept C2 si l'argument est *2*.

Listing 10 – Saisie des deux concepts pour les propositions de type 2

```
acquisition_prop_type2(Abi, [(Inst, and(CA1Final, CA2Final))|Abi])
   :-
    genere(Inst),
    acquisition_type2_concept(C1,1),
    (concept(C1) -> true; write("Warning : "), write(C1), write("
       n'est pas un concept"), nl, false),
    acquisition_type2_concept(C2,2),
    (concept(C2) -> true; write("Warning : "), write(C2), write("
       n'est pas un concept"), nl, false), nl
    write('Info : Proposition a demontrer \" '),
       affiche_concept(C1), write(' inter '), affiche_concept(C2),
       write(' include bottom\"'), nl,
    definitionAtomique(C1, CA1), definitionAtomique(C2, CA2),
    nnf(CA1, CA1Final), nnf(CA2, CA2Final).
```

La negation de $C1 \sqcap C2 \sqsubseteq \bot$ étant $\exists I : C1 \sqcap C2$, on veut donc ajouter cette dernière dans la TBox.

Le prédicat ***acquisition_prop_type2*** permet alors d'acquérir la première proposition et ajoute sa négation dans la ABox étendu en générant alors une instance *Inst* avec un nom unique. Il vérifie d'ailleurs que les concepts *C1* et *C2* sont valides puis on obtient leurs forme normales négative via ***nnf*** et on remplace toutes sous-expressions de concepts non-atomiques par leur équivalent en forme atomique.

3 Démonstration de la proposition

Dans cette dernière partie, nous allons implémenter l'algorithme qui reprends la méthode des tableaux.

Pour démontrer que la proposition entrée dans la partie 2 est présente, on développe alors la ABox avec l'algorithme des tableaux. On cherche alors à trouver la présence d'un clash parmis les feuilles qui seront trouvés à travers de l'algorithme.

Une feuille ayant un clash est donc une feuille ouverte dans le contexte de l'algorithme tandis qu'une feuille sans clash est une feuille fermée.

3.1 Tri et partionnement de la ABox

Pour procéder à la l'algorithme, il faut trier et partionner la ABox étendue en plusieurs listes qui contiennent respectivement :

- Pour les assertion de la forme $I : \exists R.C$, on les ajoutes dans la liste Lie.
- Pour les assertion de la forme $I : \forall R.C$, dans la liste Lpt.
- Pour les assertion de la forme $I : C1 \sqcap C2$, dans la liste Lpt.
- Pour les assertion de la forme $I : C1 \sqcup C2$, dans la liste Lu.
- Pour les assertion de la forme $I : C$ ou $I : \neg C$, dans la liste Ls.
- Les assertions de rôles étant déjà dans Abr, on n'a pas besoins de traitements additionels.

Listing 11 – Tri de la ABox

```
1  tri_Abox([], [], [], [], [], []).
2  tri_Abox([(I, some(R,C))|Abi],[(I, some(R,C)) | Lie],Lpt,Li,Lu,Ls)
       :-
3      tri_Abox(Abi,Lie,Lpt,Li,Lu,Ls).
4  tri_Abox([(I, all(R,C))|Abi],Lie,[(I, all(R,C)) | Lpt],Li,Lu,Ls) :-
5      tri_Abox(Abi,Lie,Lpt,Li,Lu,Ls).
6  tri_Abox([(I, and(C1,C2))|Abi],Lie,Lpt, [(I, and(C1,C2)) |
       Li],Lu,Ls) :-
7      tri_Abox(Abi,Lie,Lpt,Li,Lu,Ls).
8  tri_Abox([(I, or(C1,C2))|Abi],Lie,Lpt,Li,[(I, or(C1,C2)) | Lu],Ls)
       :-
9      tri_Abox(Abi,Lie,Lpt,Li,Lu,Ls).
10 tri_Abox([(I, C) | Abi],Lie,Lpt,Li,Lu,[(I, C) | Ls]) :-
11     cnamea(C),
12     tri_Abox(Abi,Lie,Lpt,Li,Lu,Ls).
13 tri_Abox([(I, not(C)) | Abi],Lie,Lpt,Li,Lu,[(I, not(C)) | Ls]) :-
14     cnamea(C),
15     tri_Abox(Abi,Lie,Lpt,Li,Lu,Ls).
```

A l'aide de ces listes, nous pouvons alors commencer à appliquer la méthodes des tableaux.

3.2 Résolution de la proposition par l'algorithme des tableaux

La récursion se fait à partir du prédicat resolution qui est alors le point de départ et de récursion de l'algorithme.

Listing 12 – Resolution

```
resolution(Lie, Lpt, Li, Lu, Ls, Abr) :-
    not(contient_clash(Ls)),
    complete_some(Lie, Lpt, Li, Lu, Ls, Abr).
resolution([], Lpt, Li, Lu, Ls, Abr) :-
    not(contient_clash(Ls)),
    deduction_all([], Lpt, Li, Lu, Ls, Abr).
resolution([], [], Li, Lu, Ls, Abr) :-
    not(contient_clash(Ls)),
    transformation_and([], [], Li, Lu, Ls, Abr).
resolution([], [], [], Lu, Ls, Abr) :-
    not(contient_clash(Ls)),
    transformation_or([], [], [], Lu, Ls, Abr).
resolution([], [], [], [], Ls, Abr) :-
    not(contient_clash(Ls)).
```

Au début de chaque récursion de l'algorithme, on vérifie d'abord si il y a un clash via le prédicat ***contient_clash*** dans la ABox (cf Listing 13), c'est-à-dire que Ls contient deux assertion contradictoires.

Puis le prédicat ***resolution*** tente d'appliquer une par une les règles de resolution de la méthodes des tableaux jusqu'à que ces dernières ne peuvent plus être appliqués en suivant cet l'ordre de priorité des règles suivante : ($\exists$, $\forall$, $\sqcap$, $\sqcup$). Lorsque aucune règles de résolution ne peuvent être appliqués, une dernière vérification pour les clashs se fait.

Listing 13 – Verification de clash dans la ABox

```
contient_clash([]) :- false.
contient_clash([(I, C) | Reste]) :-
    nnf(not(C), C1),
    member((I, C1), Reste) -> write("CLASH␣:␣Un␣clash␣a␣ete␣
       detecte"), nl,true ; contient_clash(Reste).
```

Le prédicat ***contient_clash*** cherche à trouver la présence de deux assertion sur une même instance qui sont contradictoires entre eux. Cela revient à prendre la tête d'une liste et vérifier la présence de sa négation parmis le reste de la liste.

Dans notre programme, ***contient_clash*** est utilisé exclusivement sur la liste Ls de la ABox étendue car c'est celle-ci qui contient les assertions de la forme $I : C$ et $I : \neg C$.

3.2.1 Règle $\exists$

La règle de résolution $\exists$ est traitée par le prédicat ***complete_some*** qui ajoute dans la ABox les deux assertions $< a, b >: R$ et $b : C$ lorsque l'on traite l'assertion de la forme $a : \exists R.C$.

Listing 14 – Application de la regle ∃

```
complete_some([(A, some(R,C)) | Lie],Lpt,Li,Lu,Ls,Abr) :-
    write("Application␣de␣la␣regle␣exists␣sur␣:␣"), nl, write("␣␣␣␣
       "), affiche_concept((A, some(R,C))), nl, nl,
    genere(B),
    evolue((B, C), Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1, Li1, Lu1,
       Ls1),
    affiche_evolution_Abox(Ls, [(A, some(R,C)) | Lie], Lpt, Li,
       Lu, Abr, Ls1, Lie1, Lpt1, Li1, Lu1, [(A, B, R) | Abr]),
    resolution(Lie1, Lpt1, Li1, Lu1, Ls1, [(A, B, R) | Abr]).
```

En ajoutant ces deux propositions dans la ABox, on génère aussi une nouvelle instance pour b. L'ajout de $b : C$ se fait avec le prédicat ***evolue***, l'ajout de $< a, b >: R$ se fait évidemment sur la liste Abr à la fin où on s'apprête à relancer une nouvelle itération.

On affiche aussi entre-temps l'état des ABox avant et après l'application de chaque règles de résolution avec le prédicat ***affiche_evolution_Abox***.

Listing 15 – Évolution de la ABox suite à l'ajout d'une assertion

```
evolue((I, C), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu, Ls) :-
    cnamea(C),
    member((I, C), Ls).
evolue((I, C), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu, [(I, C) |
   Ls]) :-
    cnamea(C),
    not(member((I, C), Ls)).
evolue((I, not(C)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu, Ls) :-
    cnamea(C),
    member((I, not(C)), Ls).
evolue((I, not(C)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu, [(I,
   not(C)) | Ls]) :-
    cnamea(C),
    not(member((I, not(C)), Ls)).
evolue((I, some(R, C)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu,
   Ls) :-
    member((I, some(R, C)), Lie).
evolue((I, some(R, C)), Lie, Lpt, Li, Lu, Ls, [(I, some(R,C)) |
   Lie], Lpt, Li, Lu, Ls) :-
    not(member((I, some(R, C)), Lie)).
evolue((I, all(R, C)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu, Ls)
   :-
    member((I, all(R, C)), Lpt).
evolue((I, all(R, C)), Lie, Lpt, Li, Lu, Ls, Lie, [(I, all(R,C)) |
   Lpt], Li, Lu, Ls) :-
    not(member((I, all(R, C)), Lpt)).
evolue((I, and(C1, C2)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu,
   Ls) :-
    member((I, and(C1, C2)), Li).
evolue((I, and(C1, C2)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, [(I,
   and(C1,C2)) | Li], Lu, Ls) :-
    not(member((I, and(C1, C2)), Li)).
```

```
25 evolue((I, or(C1, C2)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu,
       Ls) :-
26     member((I, or(C1, C2)), Lu).
27 evolue((I, or(C1, C2)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, [(I,
       or(C1,C2)) | Lu], Ls) :-
28     not(member((I, or(C1, C2)), Lu)).
```

Le prédicat ***evolue(A, Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1, Li1, Lu1, Ls1)*** tente d'ajouter dans la ABox l'assertion de concept A si elle n'est pas déjà présente dedans. Selon la nature de l'assertion, on l'ajoute dans une liste parmis *Lie, Lpt, Li, Lu, Ls*.

La ABox résultante est alors représentée par *Lie1, Lpt1, Li1, Lu1, Ls1*.

Pour afficher l'état des deux ABox, on utilise le prédicat ***affiche_evolution_Abox*** qui affiche l'état des deux ABox que l'on fournira au prédicat.

affiche_evolution_Abox affiche alors les deux ABox avec ***affiche_ABox*** qui affiche le contenu d'une ABox.

affiche_ABox quant à elle, se sert de ***affiche_Abr*** et ***affiche_Abi*** pour afficher respectivement les assertions de rôles et les assertions de concepts.

Listing 16 – Affichage de l'état de la ABox avant et après l'application d'une règle de résolution

```
1  affiche_evolution_Abox(Ls, Lie, Lpt, Li, Lu, Abr, Ls1, Lie1, Lpt1,
       Li1, Lu1, Abr1) :-
2      write("Info : Etat de la ABox etendue de depart :"), nl, nl,
3      affiche_ABox(Ls, Lie, Lpt, Li, Lu, Abr), nl,
4
5      write("Info : Etat de la ABox etendue d'arrivee :"), nl, nl,
6      affiche_ABox(Ls1, Lie1, Lpt1, Li1, Lu1, Abr1), nl.
7
8  affiche_ABox(Ls, Lie, Lpt, Li, Lu, Abr) :-
9      affiche_Abr(Abr),
10     affiche_Abi(Lie),
11     affiche_Abi(Lpt),
12     affiche_Abi(Li),
13     affiche_Abi(Lu),
14     affiche_Abi(Ls), nl.
```

affiche_Abr parcours simplement la liste d'assertion de rôle fournie et l'affiche ligne par ligne.

Tandis que ***affiche_Abi*** fonctionne de manière très similaire à ***affiche_Abr*** mais se sert de ***affiche_concept*** pour afficher les concepts à l'intérieur de l'assertion.

Listing 17 – Affichage d'une liste d'assertion

```
affiche_Abr([]).
affiche_Abr([(A, B, R) | Reste]) :-
    write("␣␣␣"),
    write("<"),write(A), write(","), write(B), write(">␣:␣"),
       write(R), nl,
    affiche_Abr(Reste).

affiche_Abi([]).
affiche_Abi([(A, C) | Reste]) :-
    write("␣␣␣"),
    write(A), write("␣:␣"), affiche_concept(C), nl,
    affiche_Abi(Reste).
```

affiche_ concept fonctionne de manière récursive et parcours la structure du concept pour afficher les sous-concepts utilisés dans la définition du concept de départ.

Pour des raisons techniques au niveau de LaTeX, nous avons remplacé les expressions : $\exists$, $\forall$, $\sqcap$, $\sqcup$, $\bot$, $\top$, $\neg$.

Par : exists, forall, inter, union, bottom, top, not.

Dans le code affiché ci-dessous.

Listing 18 – Affichage d'une liste d'assertion

```
affiche_concept(anything) :-
    write("top").
affiche_concept(nothing) :-
    write("bottom").
affiche_concept((I, C)) :-
    write(I), write("␣:␣"), affiche_concept(C).
affiche_concept(some(R,C)) :-
    write("exists"), write(R), write("."), affiche_concept(C).
affiche_concept(all(R,C)) :-
    write("forall"), write(R), write("."), affiche_concept(C).
affiche_concept(and(C1,C2)) :-
    write("("), affiche_concept(C1), write("␣inter␣"),
       affiche_concept(C2), write(")").
affiche_concept(or(C1,C2)) :-
    write("("), affiche_concept(C1), write("␣union␣"),
       affiche_concept(C2), write(")").
affiche_concept(C) :-
    cnamea(C),
    write(C).
affiche_concept(not(C)) :-
    write("not"), affiche_concept(C).
```

3.2.2 Règle ∀

Le prédicat ***deduction_ all*** représente l'application de la règle ∀.

Listing 19 – Application de la regle ∀

```
deduction_all(Lie, [(I, all(R, C)) | Lpt], Li, Lu, Ls, Abr) :-
    write("Application␣de␣la␣regle␣forall␣sur␣:␣"), nl, write("␣␣␣␣
       "), affiche_concept((I, all(R,C))), nl, nl,
    write("Abr"), nl, affiche_Abr(Abr), nl,
    findall((B, C), member((I, B, R), Abr), L),
    evolue_multi(L, Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1, Li1, Lu1,
       Ls1),
    affiche_evolution_Abox(Ls, Lie, [(I, all(R, C)) | Lpt], Li,
       Lu, Abr, Ls1, Lie1, Lpt1, Li1, Lu1, Abr),
    resolution(Lie1, Lpt1, Li1, Lu1, Ls1, Abr).
```

L'utilisation de *findall((B, C), member((I, B, R), Abr), L)* nous permet d'obtenir la liste des instances b où nous avons dans la ABox l'assertion de rôle $< a, b >: R$ (pour le rôle R de l'assertion $a : \forall R.C$ en traitement). Nous stockons donc les assertion $b : C$ dans L et nous faisons appel au prédicat ***evolue_ multi*** pour toutes les ajouter dans la ABox.

Listing 20 – Évolution de la ABox à partir d'une liste assertion

```
evolue_multi([Elem | Reste], Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1,
   Li1, Lu1, Ls1) :-
    evolue(Elem, Lie, Lpt, Li, Lu, Ls, Lie2, Lpt2, Li2, Lu2, Ls2),
    evolue_multi(Reste, Lie2, Lpt2, Li2, Lu2, Ls2, Lie1, Lpt1,
       Li1, Lu1, Ls1).
evolue_multi([], Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu, Ls).
```

Le prédicat ***evolue_ multi*** fait appel à ***evolue*** (cf Listing 15) sur le premier *Elem* de la liste, puis un appel récursif sur le *Reste*. La recursion s'arrete lorsque la liste est vide.

Finalement, on affiche l'evolution de la ABox via ***affiche_ evolution_ Abox***(cf Listing 16) et on recommence une nouvelle itération de l'algorithme en traitant un nouveau noeud.

3.2.3 Règle ⊓

Pour appliquer la règle ⊓ le prédicat ***transformation_and*** ajoute dans la ABox les assertion $I : C1$ et $I : C2$ à partir de l'assertion $I : C1 \sqcap C2$.

Et affiche l'état des ABox de départ et son état après l'application de la règle ⊓.

Pour enfin continuer et relancer une nouvelle itération de l'algorithme pour un nouveau noeud.

Listing 21 – Application de la regle ⊓

```
1  transformation_and(Lie,Lpt,[(I, and(C1, C2)) | Li],Lu,Ls,Abr) :-
2      write("Application␣de␣la␣regle␣inter␣sur␣:␣"), nl, write("␣␣␣␣
          "), affiche_concept((I, and(C1, C2))), nl, nl,
3      evolue((I, C1), Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1, Li1, Lu1,
          Ls1),
4      evolue((I, C2), Lie1, Lpt1, Li1, Lu1, Ls1, Lie2, Lpt2, Li2,
          Lu2, Ls2),
5      affiche_evolution_Abox(Ls, Lie, Lpt, [(I, and(C1, C2)) | Li],
          Lu, Abr, Ls2, Lie2, Lpt2, Li2, Lu2, Abr),
6      resolution(Lie2, Lpt2, Li2, Lu2, Ls2, Abr).
```

3.2.4 Règle ⊔

Pour appliquer la règle ⊔ le prédicat ***transformation_or*** crée deux noeuds frères dans l'arbre de l'algorithme pour ajouter dans la ABox $I : C1$ et $I : C2$ respectivement dans le premier noeud et deuxième noeud crée.

Les deux noeuds sont crée à travers le prédicat ***transformation_or_node(Node,C,Lie,Lpt,Li,[(C2))/Lu],Ls,Abr)***, où Node représente la numérotation d'un des deux noeuds crées dans la règle ⊔ pour les distinguer via un affichage, et *C1* et *C2* sont les concepts qui seront traité (pour ajouter soit l'assertion *I : C1* ou *I : C2* dans la ABox) dans ce noeud.

L'algorithme continue toujours aussi après l'application de la règles respectivement dans les deux branches qui sont donc crées par ***transformation_or***.

Listing 22 – Application de la regle ⊔

```
1  transformation_or(Lie,Lpt,Li,[(I, or(C1, C2)) | Lu],Ls,Abr) :-
2      transformation_or_node(1, C1, Lie,Lpt,Li,[(I, or(C1, C2)) |
          Lu],Ls,Abr),
3      transformation_or_node(2, C2, Lie,Lpt,Li,[(I, or(C1, C2)) |
          Lu],Ls,Abr).
4
5  transformation_or_node(Node, C, Lie,Lpt,Li,[(I, or(C1, C2)) |
      Lu],Ls,Abr) :-
6      write("Application␣de␣la␣regle␣union␣sur␣:␣"), nl, write("␣␣␣␣
          "), affiche_concept((I, or(C1, C2))), nl,
7      write("Branche␣"), write(Node), nl, nl,
8      evolue((I, C), Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1, Li1, Lu1,
          Ls1),
9      affiche_evolution_Abox(Ls, Lie, Lpt, Li, [(I, or(C1, C2)) |
          Lu], Abr, Ls1, Lie1, Lpt1, Li1, Lu1, Abr),
10     resolution(Lie1, Lpt1, Li1, Lu1, Ls1, Abr).
```

A Annexe

A.1 Manipulation du programme

Le programme est codé dans le fichier *solveur.pl* et il suffit de le charger dans *swipl*. Par défaut la TBox et la ABox sont contenues dans le fichier *tabox.pl*, pour changer de TBox et ABox on peut soit changer le contenu de *tabox.pl* ou bien changer le fichier à charger dans le prédicat *programme* dans le code au lieu d'avoir *tabox.pl*.

A.2 Tests

Les jeux de test sont disponibles dans le fichier à la fin du fichier *solveur.pl*. Il suffit de le charger dans *swipl* et de lancer le test que vous désirez.

Les tests disponibles sont :

- *test_autoref* pour le prédicat *autoref*, qui utilise le fichier *tabox_autoref.pl* dans le sous-répertoire test pour le tester sur des concepts autoréférents.
- *test_concept* pour le prédicat *concept*
- *test_partie3* pour le prédicat *troisieme_etape* pour tester l'ensemble de l'algorithme de résolution

Par exemple si nous voulons tester le prédicat ***autoref()***, nous allons appeler ***test_autoref()***. Il est aussi possible de lancer tous les tests simultanément via ***test()***. Ce dernier affichera interprétation de tous les tests effectués.

Pour tester les prédicats ***acquisition_prop_type1()*** et ***acquisition_prop_type2()*** il suffit de lancer le solveur naturellement et de tester lorsque le programme vous le demande les propositions de type 1 ou 2 et de lui donner différent concepts choisis spécialement pour le test.

Le test ***test_partie3()*** regroupe un test de tous les prédicats utilisé dans la partie 3. Il suffit de modifier *getAbi* et *getAbr* si vous souhaitez modifier Abi et Abr.

Par défaut, Abi et Abr dans ce test correspondent à ce qu'on aurait si on demande au programme de résoudre $auteur \sqcap sculpteur \sqsubseteq \bot$ à partir de la TBox et la ABox du sujet. Michel-Ange étant un sculpteur et un auteur, le programme ne devrait pas trouver de clash donc ***troisieme_etape(Abi1,Abr)*** doit donner false.