Algorithme des tableaux pour la logique de description ALC.
ABITBOL Yossef, 3804139,
SERRAF Dan, 3971120.

SORBONNE UNIVERSITÉ

RAPPORT PROJET LRC

ECRITURE EN PROLOG D'UN DÉMONSTRATEUR BASÉ SUR L'ALGORITHME DES TABLEAUX POUR LA LOGIQUE DE DESCRIPTION *ALC*

Encadrant: Colette Faucher

SORBONNE UNIVERSITÉ

Sommaire

Pour faciliter la lecture, le sommaire fait référence à la page de chaque prédicat, chaque prédicat étant séparé des autres. Pour trouver un prédicat il suffit de trouver sa page dans le sommaire et chaque page est numérotée (voir en bas à gauche).

I. INTRODUCTION :

Dans ce projet nous allons écrire à l'aide de prolog **un démonstrateur** basé sur l'algorithme des tableaux pour la logique de description ALC.
Pour cela, on va décomposer le projet en **3 parties majeures**. La première partie consiste à construire nos données en créant la Tbox et la Abox. Ensuite, dans la seconde partie nous allons saisir la proposition à démontrer et enfin dans la dernière partie on démontre la proposition saisie à l'aide du démonstrateur implémenté en prolog .

II. Architecture du projet :

Le fichier du projet nommé **'ABITBOL_SERRAF.zip'** sera structuré de la manière suivante :

- **projet.pl** : sera notre fichier principal où on écrit les différents prédicat ainsi que celui qui démarre notre programme,
- **LRC_ABITBOL_SERRAF.pdf :** sera notre fichier qui contiendra notre rapport de ce projet.
- **test.txt** : sera notre fichier qui contiendra les indications à copier et coller sur le terminal pour exécuter les tests.

III. Étape 1 , Préparation des données :

La première étape consiste à implémenter la Tbox et Abox de l'exercice 3 du TD4 dans le fichier **'premiere_etape.pl'.**

On considère la TBox suivante :

Sculpteur ≡ Personne ⊓ ∃a_cree.Sculpture
Auteur ≡ Personne ⊓ ∃a_ecrit.Livre
Editeur ≡ Personne ⊓ ¬(∃a_ecrit.Livre) ⊓ ∃a_edite.Livre
Parent ≡ Personne ⊓ ∃a_enfant.⊤

et la A Box suivante :

Michel-Ange : Personne
David : Sculpture
Sonnets : Livre
⟨Michel-Ange, David⟩ : a_cree
⟨Michel-Ange, Sonnets⟩ : a_ecrit
Vinci : Personne
Joconde : Objet
⟨Vinci, Joconde⟩ : a_cree

Pour ce faire nous aurons besoin de définir au préalable différents prédicats :

- **equiv(ConceptAtom,ConceptGen)** : nous permettra de préciser les équivalences entre concepts. Il prend deux paramètres , le premier paramètre doit être un concept atomique et le deuxième un concept quelconque.
- **cnamea(ConceptAtom)** : nous permettra de préciser les identificateurs des concepts atomiques. Il prend un unique paramètre qui est un concept atomique.
- **cnamena(ConceptNonAtom)** : nous permettra de préciser les identificateurs non atomiques. Il prend un unique paramètre qui est un concept non atomique.
- **iname(Instance)** : nous permettra de préciser les identificateurs des instances. Il prend un unique paramètre qui est une instance.
- **rname(Role)** : nous permettra de préciser les identificateurs de rôles . Il prend un unique paramètre qui est un rôle.
- **inst(Instance,ConceptGen)** : nous permettra de préciser les instanciations de concepts. Il prend deux paramètres, un premier paramètre qui est une instance et un second paramètre qui est un concept général.
- **instR(Instance1, Instance2, role)** : nous permettra de préciser les instantiations de rôles . Il prend trois paramètres, les deux premiers sont des instances et le dernier est un rôle.

Autoref

Quel est le but de ce prédicat:

Parmi l'ensemble de nos prédicats décrit ci dessus nous pouvons remarquer qu'il y a des concepts atomiques et des concepts complexes(qui peuvent être potentiellement composés d'autre concepts complexes).
Un **problème** se pose alors dans le cas où nous aurions à tester une équivalence entre un concept atomique et un concept complexe qui contient ce concept atomique car cela produirait **un bouclage dans le traitement des expressions de concepts.**
Ce problème nous a amené à créer un nouveau prédicat : **autoref**.
autoref(C,D) prend **2 paramètres** et teste si un concept C apparaît directement ou indirectement dans D.

Comment ce prédicat a été implémenté ?:

autoref a été construit de manière à prendre en compte les différentes constructions possibles.

```
61  /* /\/\/\ AUTOREF /\/\/\*/
62  /*Test si le predicat C apparait directement ou indirectement dans la definition de D*/
63  autoref(C,C). % cas de base D == C
64  autoref(C1,equiv(C2,X)) :- autoref(C1,X),!.
65  autoref(C,and(C1,C2)) :- autoref(C,C1),!.
66  autoref(C,and(C1,C2)) :- autoref(C,C2),!.
67  autoref(C,or(C1,C2)) :- autoref(C,C1),!.
68  autoref(C,or(C1,C2)) :- autoref(C,C2),!.
69  autoref(C,some(R,C1)) :- autoref(C,C1),!.
70  autoref(C,all(R,C1)) :- autoref(C,C1),!.
71  autoref(C,not(C1)) :- autoref(C,C1),!.
72  % on test si D est un concept non atomique, si oui, on fait un autoref sur sa valeur.
73  autoref(C1,C2) :- setof(X,cnamena(X),L) , member(C2,L) ,equiv(C2,Y),autoref(C1,Y).
74
```

Pour ce faire on doit **décomposer** tous les cas que D peut prendre:

- **AND, OR** : on test si C appartient a C1 ou C2
- **SOME, ALL :** on teste si C appartient a C1, R étant un rôle.
- Le plus important, **C2 un concept complexe,** on vérifie que C2 est bien un concept atomique et on remplace donc par son équivalent Y, on applique ensuite autoref(C,Y)

Tests du prédicat:

Le premier cas a tester est **autoref(personne,personne).** : (C appartient bien à D)

```
38 ?- autoref(personne,personne).
true .
```

Pour bien tester ce prédicat on va prendre la Tbox et Abox de l'annexe 1 et on va rajouter l'équivalence suivante: **`equiv(sculpture,and(objet,all(cree_par,sculpteur))).`**

ensuite, on va tester : **autoref(sculpture,sculpteur) :** (On a vu que sculpture appartient bien au concept complexe sculpteur qui est : `equiv(sculpteur,and(personne,some(aCree,sculpture))).`

```
39 ?- autoref(sculpture,sculpteur).
true.
```

Et si on test : **autoref(auteur,sculpteur).** On obtient bien FALSE.

Premiere_etape

Quel est le but de ce prédicat:

Ensuite, on va représenter notre Tbox par **une liste de doublets (1)** et notre Abox sera décomposée en 2 sous listes, une sous liste contiendra **les assertions de concepts (2)** et la seconde contiendra les **assertions de rôles (3)** .

Comment ce prédicat a été implémenté ?:

Ainsi le prédicat qui correspond à la première étape est définie de la manière suivante :

```
72  premiere_etape(Tbox,Abi,Abr) :-
73      setof((X,Y),equiv(X,Y),Tbox),       % (1)
74      setof((X,Y),inst(X,Y),Abi),         % (2)
75      setof((X,Y,Z),instR(X,Y,Z),Abr).    % (3)
```

setof(Motif, But, Liste)

setof/3 crée une liste des instantiations de Motif par retours arrière sur But et unifie le résultat dans Liste.
Ici, tout simplement en faisant **setof** on va pouvoir récupérer la liste des **equiv(X,Y)**, **inst(X,Y)**, **instR(X,Y,Z)** et unifier le résultats dans les listes Tbox, Abi et Abr respectivement.

Tests du prédicat:

On va tester cela dans le dernier prédicat qui est **programme.**

SORBONNE UNIVERSITÉ

IV. Étape 2: Saisie de la proposition à démontrer

Deuxieme_etape

Quel est le but de ce prédicat:

Dans cette seconde partie nous allons réaliser un programme qui permet à l'utilisateur de **saisir la proposition à démontrer.**
Pour ce faire nous avons créé un menu afin de gérer les différents cas possibles .

Comment ce prédicat a été implémenté ?:

```
/*<><><><><><><><><><><><><><><><><><><><><><><><>> DEUXIEME ETAPE <><><><><><><><><><><><><><><><><><><><><><><><>*/

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, % (1)
    write("2 - Deux concepts ne possede aucun elements en commun(ils ont une intersection vide)."),nl, % (2)
    nl,read(R),
    suite(R,Abi,Abi1,Tbox).

suite(1,Abi,Abi1,Tbox) :- acquisition_prop_type1(Abi,Abi1,Tbox),!.
suite(2,Abi,Abi1,Tbox) :- acquisition_prop_type2(Abi,Abi1,Tbox),!.
suite(R,Abi,Abi1,Tbox) :- nl, write("Cette reponse est incorrecte."),
    nl, saisie_et_traitement_prop_a_demontrer(Abi,Abi1,Tbox).
```

deuxieme_etape prend **3** paramètres qui sont respectivement la liste des assertions de concepts initiales de la Abox, la liste des assertions de concepts complétée après la soumission d'une proposition à démontrer et la liste représentant la Tbox. On va demander à l'utilisateur de choisir entre 1. et 2. et selon son choix on va lui demander d'entrer les informations nécessaires.

1) Pour une proposition de type I : **I : C**
2) Pour une proposition de type II : **C1 ⊓ C2 ⊑ ⊥**

Tests du prédicat:
On va tester cela dans le dernier prédicat qui est **programme.**

Concept

Quel est le but de ce prédicat:

concept(C), Instance(I), role(R) prennent un seul paramètre, le concept C,l'instance I ou le rôle R et test si C, I ou R existe bien, c'est à dire, qu'il a bien été initialisé dans la partie 1, c'est ce qu'on appelle **la correction sémantique.**

Comment ce prédicat a été implémenté ?:

3) Grammaire

concept ::= ⟨concept atomique⟩
| ⊤
| ⊥
| ¬ ⟨concept⟩
| ⟨concept⟩ ⊔ ⟨concept⟩
| ⟨concept⟩ ⊓ ⟨concept⟩
| ∃ ⟨rôle⟩ . ⟨concept⟩
| ∀ ⟨rôle⟩ . ⟨concept⟩

```
/* -------------------- On fait la verification SEMANTIQUE --------------------*/
concept(nothing).
concept(anything).
% on decompose les concepts
concept(not(C)) :- concept(C),!.
concept(and(C1,C2)):- concept(C1), concept(C2),!.
concept(or(C1,C2)) :- concept(C1), concept(C2),!.
concept(some(R,C)) :- role(R), concept(C),!.
concept(all(R,C)) :- role(R), concept(C),!.
% on vérifie bien qu'ils existent
% concept(C1) :- setof(X,iname(X),L), member(C1,L),!.
concept(C2) :- setof(X,cnamea(X),L), member(C2,L),!.
concept(C3) :- setof(X,cnamena(X),L), member(C3,L),!.
% On vérifie pour les instances
instance(I) :- setof(I, iname(I),L), member(I,L) ,!.
% on vérifie pour les roles
role(R) :- setof(X,rname(X),L), member(R,L),!.
```

On remarque que les différentes expressions d'un concept, à part les 3 premières, **sont récursives.**
Ainsi, d'après la 4ème expression, un concept peut être la négation d'un concept, ce qu'on traduit par la clause : concept(not(C)) :- concept(C). De même pour ⊔ et ⊓, on doit vérifier que C1 et C2 **sont bien des concepts**, et pour ∀ et ∃ ,qui font appel à des rôles, on doit vérifier que ce **sont bien des rôles**. Enfin, on a besoin de clauses sans appels récursifs qui traduisent le fait que concept(C) **est vrai si C appartient à la liste des concepts atomiques / non atomiques.** On ajoute aussi la clause pour l'instance I.

Tests du prédicat:

On teste un premier cas : **concept(and(personne,all(aCree,sculpteur))).**
—> (personne et sculpteur sont bien des concepts et aCree est bien un rôle).

```
[2] 60 ?- concept(and(personne,all(aCree,sculpteur))).
true.
```

On teste le cas avec un rôle erroné : **concept(and(personne,all(personne,sculpteur))).**
—> personne n'étant pas un rôle , on renvoi **False**

```
[2] 61 ?- concept(and(personne,all(personne,sculpteur))).
false.
```

On teste une instance, **instance(michelAnge).**
—> michelAnge est bien une instance.

```
[2] 62 ?- instance(michelAnge).
true.
```

Remplace

Quel est le but de ce prédicat:

Ce prédicat a pour but de, comme son nom l'indique, **remplacer** de manière récursive les identificateur de concepts complexes par leur définition. Il prend en paramètre 2 éléments, le premier, C est le concept complexe qu'on doit remplacer et on renvoie le remplacement dans RC.

Comment ce prédicat a été implémenté ?:

```
/* -------------------- On fait le REMPLACEMENT -------------------- */
% On remplace C1 et C2 dans RC1 et RC2
remplace(and(C1,C2),and(RC1,RC2)) :- remplace(C1,RC1), remplace(C2,RC2),!.
remplace(or(C1,C2),or(RC1,RC2)) :- remplace(C1,RC1), remplace(C2,RC2),!.
remplace(all(R,C),all(R,RC)) :- remplace(C,RC),!.
remplace(some(R,C),some(R,RC)) :- remplace(C,RC),!.
remplace(not(C),not(RC)) :- remplace(C,RC),!.

remplace(C,RC) :- setof(X,cnamena(X),L) , member(C,L) , equiv(C,RC),!. % concept non atomique
remplace(C,C) :- setof(X,cnamea(X),L) , member(C,L),!. % concept atomique
```

Pour faire cela, on vérifie que C est dans la liste **cnamena** grâce à la clause qui précise **les identificateurs des concepts non atomique,** si oui, on cherche son équivalent qu'on va mettre dans RC.
Or, C peut lui même être complexe, on doit alors faire des **appels récursifs** : C = and(C1,C2) alors on va remplacer C1 et remplacer C2 qu'on va mettre respectivement dans and(RC1,RC2). De même pour les autres.

Tests du prédicat:

On remplace un concept non atomique : **remplace(auteur,X).**

—> `equiv(auteur,and(personne,some(aEcrit,livre))).`

```
[2] 63 ?- remplace(auteur,X).
X = and(personne, some(aEcrit, livre)).
```

On remplace un concept atomique : **remplace(personne,X).**

```
[2] 64 ?- remplace(personne,X).
X = personne.
```

on remplace un concept complexe : **remplace(and(personne,all(aCree,or(editeur,objet))),X).**

—> `equiv(editeur,and(personne,and(not(some(aEcrit,livre)),some(aEdite,livre)))).`

```
[2] 65 ?- remplace(and(personne,all(aCree,or(editeur,objet))),X).
X = and(personne, all(aCree, or(and(personne, and(not(some(aEcrit, livre)), some(aEdite, livre))), objet))).
```

On doit aussi rajouter le prédicat **nnf** qui a été donné.

Acquisition_prop_type1

I : C L'instance I appartient au concept C.
La négation de cette proposition que l'on doit ajouter aux assertions de la Abox est : **I : ¬C**
Dans ce cas il saisit une instance I, on vérifie **sémantiquement** si cette instance en est bien une.
Suite à cela, il entre le concept C et on vérifie **sémantiquement** que ce concept existe et qu'il est correct.
Puis, sur **C** on **remplace** les identificateurs de concept complexes par leur définition.
Enfin on les mets sous forme normale négative **(nnf(not(RC)))** et on l'ajoute à notre liste.

```
/* Une instance donnee appartient a un concept donne I : C */
acquisition_prop_type1(Abi , Abi1 ,Tbox) :- nl,
    % INSTANCE
    write("entrer linstance "),nl, read(I), % on stock l'instance
    instance(I),nl, % on vérifie que c'est  sémantiquement  correct

    % CONCEPT
    write("entrer le concept"),nl, read(C),
    % on vérifie que c'est  sémantiquement  correct
    concept(C),

    % On REMPLACE les identificateurs de concept complexes par leur définition,
    remplace(C,RC),
    % On les mets sous FORME NORMAL NEGATIVE ( NNF ),
    nnf(not(RC),NRC),
    % On l'ajoute a la liste
    concat(Abi,[(I,NRC)],Abi1),
    nl, write("Abi1 etendu : "),nl, write(Abi1).
```

Tests du prédicat:

On va exécuter le programme et on va saisir : **michelAnge : personne :**

```
Entrez le numero du type de proposition que vous voulez demontrer :
1 - Une instance donnee appartient a un concept donne.
2 - Deux concepts ne possede aucun elements en commun(ils ont une intersection vide).

|: 1.

entrer linstance
|: michelAnge.

entrer le concept
|: personne.

Abi1 etendu :
[(david,sculpture),(joconde,objet),(michelAnge,personne),(sonnets,livre),(vinci,personne),(michelAnge,not(personne))]
```

Le menu s'affiche bien, on choisit 1. Pour notre premier cas, on rentre l'instance **michelAnge** et le concept **personne**. On peut voir l'impact sur la sous liste des assertions de concepts qui est Abi1 qui nous affiche bien [...,**(michelAnge, not(personne))**].

Acquisition_prop_type2

C1 ⊓ C2 ⊑ ⊥ : Les concepts C1 et C2 ont une intersection vide.
Dans ce cas, l'utilisateur saisit le premier concept C1, on vérifie **sémantiquement** que ce concept existe et qu'il est correct.
Suite à cela, il saisit le second concept C2, on vérifie **sémantiquement** que ce concept existe et qu'il est correct.
Après nous **remplaçons** l'intersection de ces deux concepts par leur définition.
Enfin nous prenons la **négation** de cette proposition c'est-à-dire qu'il existe **UNE (au moins) instance commune à C1 et C2.** Pour cela, on va générer une nouvelle instance automatique.
On la met sous forme normale négative **(nnf)** et on l'ajoute à notre liste.

```
/*Deux concepts ne possede aucun elements en commun(ils ont une intersection vide C1 ∩ C2 ⊆ ⊥ */
acquisition_prop_type2(Abi,Abi1,Tbox) :- nl,
    % Concept C1
    write("Entrer le concept 1 : "),nl,read(C1),
    % on vérifie que c'est  sémantiquement  correct
    concept(C1),

    % Concept C2
    nl,write("Entrer le concept 2 : "),nl,read(C2),
    % on vérifie que c'est  sémantiquement  correct
    concept(C2),

    % On REMPLACE les identificateurs de concept complexes par leur définition,
    remplace(and(C1,C2),RC),
    % On les mets sous FORME NORMAL NEGATIVE ( NNF ),
    nnf(not(RC),NRC),
    % On l'ajoute a la liste
    genere(Nom),
    concat(Abi,[(Nom,NRC)],Abi1),
    nl, write("Abi1 etendue : "),nl, write(Abi1).
```

Tests du prédicat:
On va exécuter le programme et on va saisir : **C1: livre** et **C2: objet :**

```
Entrez le numero du type de proposition que vous voulez demontrer :
1 - Une instance donnee appartient a un concept donne.
2 - Deux concepts ne possede aucun elements en commun(ils ont une intersection vide).

|: 2.

Entrer le concept 1 :
|: livre.

Entrer le concept 2 :
|: objet.

Abi1 etendue :
[(david,sculpture),(joconde,objet),(michelAnge,personne),(sonnets,livre),(vinci,personne),(inst1,or(not(livre),not(objet)))]
```

Le menu s'affiche bien, on choisit 2. Pour notre deuxième cas, on rentre le concept 1 : **livre** et le concept 2: **objet**.
On peut voir l'impact sur la sous liste des assertions de concepts qui est Abi1 qui nous affiche bien
[.....,(inst1, or(not(livre), not(objet)))].

V. ETAPE 3: Démonstration de la proposition :

Dans cette dernière partie nous allons implémenter l'**algorithme de résolution** basé sur la méthode des tableaux. Pour ce faire, on va se baser sur la Abox étendue que l'on a construite dans la partie 2 et l'on va démontrer qu'elle est **insatisfiable**. On va donc construire progressivement **un arbre de résolution** à partir de ces assertions.

Tri_Abox

La première étape consiste alors à avoir un prédicat qui va nous permettre de vérifier si on obtient des branches qui seront fermées. Pour accélérer le processus, on a d'abord créé un prédicat **tri_Abox**.

Quel est le but de ce prédicat:

Ce prédicat a pour but de, comme son nom l'indique, **trier** de manière récursive les assertions pour faciliter le développement de la démonstration. A partir de la liste des assertions de concepts de la Abox étendue après soumission d'une proposition à démontrer, génère 5 listes à savoir **Lie ,Lpt ,Li , Lu et enfin Ls .**

Comment ce prédicat a été implémenté ?:

```
% on commence par implementer le predicat tri_Abox :
tri_Abox([],[],[],[],[],[]).

tri_Abox([(I,C)|Abi],Lie,Lpt,Li,Lu,[(I,C)|Ls]) :- setof(X,cnamea(X),L), member(C,L) , tri_Abox(Abi,Lie,Lpt,Li,Lu,Ls),!.
tri_Abox([(I,not(C))|Abi],Lie,Lpt,Li,Lu,[(I,not(C))|Ls]) :- setof(X,cnamea(X),L), member(C,L) , tri_Abox(Abi,Lie,Lpt,Li,Lu,Ls),!.

tri_Abox([(I,some(R,C))|Abi],[(I,some(R,C))|Lie],Lpt,Li,Lu,Ls)  :- tri_Abox(Abi,Lie,Lpt,Li,Lu,Ls),!.
tri_Abox([(I,all(R,C))|Abi],Lie,[(I,all(R,C))|Lpt],Li,Lu,Ls) :- tri_Abox(Abi,Lie,Lpt,Li,Lu,Ls),!.
tri_Abox([(I,and(C1,C2))|Abi],Lie,Lpt,[(I,and(C1,C2))|Li],Lu,Ls)  :- tri_Abox(Abi,Lie,Lpt,Li,Lu,Ls),!.
tri_Abox([(I,or(C1,C2))|Abi],Lie,Lpt,Li,[(I,or(C1,C2))|Lu],Ls) :- tri_Abox(Abi,Lie,Lpt,Li,Lu,Ls),!.
```

On a implémenté ce prédicat de la manière suivante, tout d'abord le cas des listes vides. Ensuite selon les différents types d'éléments (some,all, and,or,not), on les met dans leur liste respective, à savoir Lie ,Lpt ,Li , Lu et enfin Ls .

Tests du prédicat:

```
[1] 40 ?- tri_Abox([(david,sculpture),(david,not(sculpture)),(michelAnge,some(aCree,personne)),(vinci,all(aCree,livre)),(david,and(s
culpture,personne)),(david,or(sculpture,personne))],Lie,Lpt,Li,Lu,Ls).
Lie = [(michelAnge, some(aCree, personne))],
Lpt = [(vinci, all(aCree, livre))],
Li = [(david, and(sculpture, personne))],
Lu = [(david, or(sculpture, personne))],
Ls = [(david, sculpture),  (david, not(sculpture))].
```

Suite à cette exécution on remarque qu' en sortie on obtient bien nos différents types d'éléments dans leur liste respective.

A présent on peut passer à notre fonction principale qui va nous permettre de résoudre cette démonstration, **résolution** .

Resolution

Quel est le but de ce prédicat:

Ce prédicat va nous permettre de **construire notre arbre** en vérifiant tout d'abord dans un premier temps s'il n'existe pas de contradiction déjà présente dans la Abox puis il essaye d'**appliquer une des règles** que nous allons développer plus bas jusqu'à aboutir à une **contradiction**.
Si aucun clash n'a été trouvé, la proposition n'a pas pu être démontrée.

Comment ce prédicat a été implémenté ?:

```
% Ensuite on a le prédicat
% /\/\/\  resolution  /\/\/\  :
resolution(Lie,Lpt,Li,Lu,Ls,Abr) :-
    clash(Ls),
    complete_some(Lie,Lpt,Li,Lu,Ls,Abr),
    transformation_and(Lie,Lpt,Li,Lu,Ls,Abr),
    deduction_all(Lie,Lpt,Li,Lu,Ls,Abr),
    transformation_or(Lie,Lpt,Li,Lu,Ls,Abr).
```

On test tout d'abord si il **ya un clash dans la liste Ls** qui correspond à la liste des assertions de concepts après le trie de tri_Abox, si il n'y a pas de clash on renvoi donc **True** (voir le prédicat clash), on va essayer d'appliquer la règle **complete_some** qui correspond à la règle du ∃ , si on arrive à l'appliquer, on fait évoluer les listes grâce au prédicat évolue et on refait la résolution avec les listes mises à jour. si complete_some n'a pas pu être exécutée on essaye avec la règle du **and** puis celle du **all** et enfin celle du **or** si dans aucune de ces règles on obtient un clash **on renvoi alors True.**

Tests du prédicat:

Pour pouvoir exécuter ce prédicat, il faut implémenter tous les prédicats auquel il fait référence, c'est ce qu'on va faire plus bas, le test sera fait grâce au prédicat **programme**, tout a la fin.

Clash

La première étape consiste à voir si on a un clash directement dans Ls obtenu grâce au tri_Abox avant même d'appliquer une des règles. Cela se fait grâce au prédicat **clash**.

Quel est le but de ce prédicat:

Le prédicat **clash(L)** va prendre un paramètre la liste L et nous renvoie **False si il trouve un clash**, c'est a dire 2 éléments complémentaires, sinon **True**.

Comment ce prédicat a été implémenté ?:

```
%---------------------------  CLASH  -----------------------------------
% et on va devoir tester si ya un clash a chaque nouvel ajout, on implemente donc le predicat, clash :
% FALSE => CLASH       &        TRUE => PAS DE CLASH
clash([]).
clash([(I,E)|L]) :- clash_elem(L,(I,E)), clash(L).

clash_elem([],(_,_)).
clash_elem([(U,A)|L],(I,E)) :-
    setof(X,iname(X),L3) , member(I,L3), member(U,L3),
    (U \== I ), clash_elem(L,(I,E)),!.

% le cas ou U et I sont des instances qui existent
clash_elem([(U,A)|L],(I,E)) :-
    setof(X,iname(X),L3) , member(I,L3), member(U,L3),
    U == I, % on verifie que c'est les memes
    % si ya un clash ca renvoi dont False
    A \== not(E),
    E \== not(A),
    setof(X,iname(X),L3) , member(I,L3),
    clash_elem(L,(I,E)),!.

% Le cas ou on a genere 'inst'
clash_elem([(U,A)|L],(I,E)) :-
    setof(X,iname(X),L3) ,
    not(member(I,L3)),
    A \== not(E),
    E \== not(A),
    clash_elem(L,(I,E)),!.
% ------------------------------------------------
```

Clash(L) a été pensé pour pouvoir renvoyer **False lors d'un clash** et **True quand il n'y en a pas**, selon 2 cas.

- Le premier cas est le cas où **U et I sont des instances qui existent** et il va donc comparer que U et I sont bien égaux, et si ils sont égaux il va voir ensuite les concepts A et E pour les comparer, si A est égal a not(E), c'est a dire qu'il a y'a un clash, il renvoi donc False sinon True.
- Le deuxième cas, correspond au fait que des fois dans la démonstration, et dans la partie 2, on **génère des instances grâce au prédicat donné génère(X)** qui génère des instances ne faisant donc pas

partie de la liste des iname. Dans ce cas, si on trouve que I ne fait pas partie de la liste, c'est donc forcément une instance générée et on doit donc vérifier pour le clash que A et E sont bien égaux pour renvoyer False sinon True.

On fait ce raisonnement sur chaque élément de la liste L.

Tests du prédicat:

On test un clash simple :
———> david == david et on a bien sculpture et son **complémentaire**, il y a un **clash**, on renvoie **False.**

```
[1] 10 ?- clash([(david,sculpture),(david,not(sculpture))]).
false.
```

On test maintenant le cas où un des deux instances est différentes de david :
———> david \== vinci, or vinci est bien une instance existante, il ya donc **pas de clash**, on renvoie **True.**

```
[1] 11 ?- clash([(david,sculpture),(vinci,not(sculpture))]).
true.
```

Enfin le test avec une instance qui n'est pas dans la liste des identificateurs des instances :
———> david \== inst1, or inst1 n'est pas une instance existante, c'est à dire qu'elle a été généré par le prédicat génère et on a bien sculpture et son **complémentaire**, il ya un **clash**, on renvoie **False.**

```
[1] 12 ?- clash([(david,sculpture),(inst1,not(sculpture))]).
false.
```

Evolue

Avant de passer aux prédicats correspondants à ceux utilisés dans résolution, il est important d'implémenter le prédicat **évolue**.

Quel est le but de ce prédicat:

evolue(A, Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1, Li1, Lu1, Ls1) prend plusieurs paramètre dont A qui représente une nouvelle assertion de concepts à intégrer dans l'une des listes Lie, Lpt, Li, Lu ou Ls qui décrivent les assertions de concepts de la Abox étendue et Lie1, Lpt1, Li1,Lu1 et Ls1 représentent les nouvelles listes mises à jour.

Comment ce prédicat a été implémenté ?:

```
% Lors de la resolution, on va devoir faire evoluer la liste a chaque fois grace au predicat
% /\/\/\  evolue  /\/\/\ :
evolue((B,C), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu,[(B,C)| Ls]) :-   setof(C, cnamea(C),L), member(C,L),!.
evolue((B,C), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu,[(B,C)| Ls]) :-   setof(C, cnamena(C),L), member(C,L) ,!.
evolue((B,not(C)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu,[(B,not(C))| Ls]) :-  setof(C, cnamena(C),L), member(C,L),!.
evolue((B,not(C)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu,[(B,not(C))| Ls]) :-  setof(C, cnamea(C),L), member(C,L) ,!.
evolue((I,and(C1,C2)), Lie, Lpt,Li, Lu, Ls, Lie, Lpt,  [(I,and(C1,C2))|Li], Lu, Ls):-!.
evolue((I,or(C1,C2)), Lie, Lpt, Li,Lu, Ls, Lie, Lpt, Li, [(I , or(C1,C2))| Lu], Ls):-!.
evolue((I,some(R,C)), Lie, Lpt, Li, Lu, Ls, [(I,some(R,C))|Lie], Lpt, Li, Lu, Ls):-!.
evolue((I,all(R,C)), Lie, Lpt, Li, Lu, Ls, Lie, [(I,all(R,C))|Lpt], Li, Lu, Ls):-!.
```

Selon les différents types d'éléments (some,all, and,or,not), on met à jour les différentes listes respectives, à savoir Lie1 ,Lpt1, Li1, Lu1 et enfin Ls1 en concaténant l'élément en question et la liste avant la mise à jour, à savoir Lie ,Lpt, Li, Lu et enfin Ls.

Tests du prédicat:

Premier test simple qui permet de tester un élément du style **(David,sculpture).**

———> l'assertion est une assertion simple, donc à l'ajouter avec la liste Ls :

```
[1] 13 ?- evolue((david,sculpture),Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu,Ls).
Ls = [(david, sculpture)|Ls].
```

Deuxième test avec un **and :**

———> Lorsqu'on a un **and** on doit donc mettre à jour la liste Li et on renvoie la concaténation de cette élément avec la liste Li.

```
[1] 15 ?- evolue((david,and(sculpture,livre)),Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu,Ls).
Li = [(david, and(sculpture, livre))|Li].
```

La seconde étape consiste à chercher à appliquer les différentes règles possibles sous un ordre arbitraire.

Complete_some

Quel est le but de ce prédicat:

La première règle consiste a cherché à appliquer **la règle ∃** c'est à dire si, on trouve une assertion de la forme a : ∃ **R.C** on ajoute les assertions **<a, b> : R** et **b : C,** où b est un nouvel objet et l'on génère un nouveau noeud à l'arbre de résolution.

Comment ce prédicat a été implémenté ?:

```
% /\/\/\  complete_some  /\/\/\ :
complete_some([],Lpt,Li,Lu,Ls,Abr).
complete_some([(A,some(R,C))|Lie],Lpt,Li,Lu,Ls,Abr):-
    genere(B), %on genere l'instance B
    evolue((inst,C), Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1, Li1, Lu1, Ls1),
    concat([(A,B,R)],Abr,Abr1), % on ajoute <a,b> : R
    % ===== AFFICHAGE =====
    nl, write(">>>>> REGLE complete_some : "),nl,
    affiche_evolution_Abox(Ls, [(A,some(R,C))|Lie], Lpt, Li, Lu, Abr, Ls1, Lie1, Lpt1, Li1, Lu1, Abr1),
    % =====================
    clash(Ls1), % on test si ya un clash
    resolution(Lie1, Lpt1,Li1,Lu1,Ls1,Abr1).
```

Donc d'abord le cas de la liste vide, si la liste est vide, on renvoi True et on passe au prédicat suivant, sinon Lie contient bien un élément de la forme **(A,some(R,C))** et dans ce cas on doit appliquer la règle **∃ , à savoir,** générer un nouvel objet B, ajouter l'assertion **B : C** ainsi que **<a, b> : R,** une fois ces assertions ajouté à la nouvelle liste Ls1 et Abr1, on teste si ya un clash, si ya un clash (comme vu plus haut) on renvoi directement **False** et on renvoi False pour résolution, sinon True et on fait une nouvelle résolution avec les nouvelles listes.

Tests du prédicat:

Pour réaliser les tests, on met en commentaire la résolution, car on veut que le programme s'arrête juste après le clash, pour des raisons d'affichages, on a mis en commentaire aussi l'affichage et enfin on a remplacé genere(B) par une instance nommée inst directement.

—>) Test avec un clash :

On va rajouter à Ls1 l'assertion **B : C** avec **C = not(personne)** et **B = genere(X)** , donc il y a un **clash** entre (michelAnge, personne) et (inst1,not(personne)) ⇒ on renvoie **False.**

```
[1] 8 ?- complete_some([(david,some(aCree,not(personne)))],[],[],[],[(michelAnge,personne)],[]).
false.
```

—>) Test sans clash:

On va ajouter à Ls1 l'assertion **B : C** avec **C=not(personne)** et **B=genere(X)**, donc il y a pas de **clash**: True

```
[1] 9 ?- complete_some([(david,some(aCree,not(personne)))],[],[],[],[(michelAnge,sculpture)],[]).
true.
```

Transformation_and

Quel est le but de ce prédicat:

Si, on a pas réussi à appliquer la règle précédente, on cherche à appliquer la seconde règle qui consiste a cherché à appliquer la règle du **and,** pour ce faire on ajoute les assertions **a : C** et **a : D** et l'on génère un nouveau nœud à l'arbre de résolution.

Comment ce prédicat a été implémenté ?:

```
% /\/\/\  transformation_and  /\/\/\  :
transformation_and(Lie,Lpt,[],Lu,Ls,Abr).
transformation_and(Lie, Lpt,[(I,(and(C1,C2)))|Li],Lu, Ls,Abr) :-
    % On fais avec C1
    evolue((I,C1), Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1, Li1, Lu1, Ls1),
    % ===== AFFICHAGE ===
    nl, write(">>>>> REGLE transformation_and : "),nl,
    affiche_evolution_Abox(Ls, Lie, Lpt, [(I,(and(C1,C2)))|Li], Lu, Abr, Ls1, Lie1, Lpt1, Li1, Lu1, Abr),
    % ======================
    clash(Ls1), % test du clash

    % On fais avec C2
    evolue((I,C2), Lie1, Lpt1, Li1, Lu1, Ls1, Lie2, Lpt2, Li2, Lu2, Ls2),
    % ===== AFFICHAGE =====
    affiche_evolution_Abox(Ls1, Lie1, Lpt1, Li1, Lu1, Abr, Ls2, Lie2, Lpt2, Li2, Lu2, Abr),
    % ======================
    clash(Ls2), % test du clash
    resolution(Lie2, Lpt2, Li2, Lu2, Ls2,Abr).
```

Donc d'abord le cas de la liste vide, si la liste est vide, on renvoi True et on passe au prédicat suivant, sinon Li contient bien un élément de la forme **(I,and(C1,C2))** et dans ce cas on doit appliquer la règle **and, à savoir,** ajouter l'assertion **I : C1** ainsi que **I : C2,** une fois ces assertions ajouté à la nouvelle liste Ls1, on teste si ya un clash, si ya un clash (comme vu plus haut) on renvoi directement **False** et on renvoi False pour résolution**,** sinon True et on fait une nouvelle résolution avec les nouvelles listes.

Tests du prédicat:

Pour réaliser les tests, on met en commentaire la résolution, car on veut que le programme s'arrête juste après le clash, pour des raisons d'affichages, on a mis en commentaire aussi l'affichage.

—>) Test avec un clash : on ajoute à Ls1 : (david,personne) et (david,not(sculpture)), il y a donc **clash** entre **(david,not(sculpture))** et **(david,sculpture)** ⇒ on renvoie **False.**

```
9 ?- transformation_and([],[],[(david,and(personne,not(sculpture)))],[],[(david,sculpture)],[]).
false.
```

—>) Test sans clash : on ajoute à Ls1 : (david,personne) et (david,livre), il y a donc **pas de clash** ⇒ on renvoie **True.**

```
11 ?- transformation_and([],[],[(david,and(personne,livre))],[],[(david,sculpture)],[]).
true.
```

Deduction_all:

Quel est le but de ce prédicat:

Si, on a pas réussi à appliquer aucune des règles précédente, on cherche à appliquer la troisième règle qui consiste a cherché à appliquer la règle du **all** pour ce faire on ajoute si on trouve des assertions de la forme **a : ∀R.C ET <a, b>: R**, alors on ajoute à l'assertion **b : C** et l'on génère un nouveau noeud de l'arbre de résolution.

Comment ce prédicat a été implémenté ?:

```
% /\/\/\  deduction_all  /\/\/\  :
deduction_all(Lie,[],Li,Lu,Ls,Abr).
deduction_all(Lie,[(A,all(R,C))|Lpt],Li,Lu,Ls, Abr) :-
    member((A,B,R),Abr), % on test si <a,b> : R est dans Abr
    evolue((B,C), Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1, Li1, Lu1, Ls1),
    % ===== AFFICHAGE =====
    nl, write(">>>>> REGLE deduction_all : "),nl,
    affiche_evolution_Abox(Ls, Lie, [(A,all(R,C))|Lpt], Li, Lu, Abr, Ls1, Lie1, Lpt1, Li1, Lu1, Abr),
    % ======================
    clash(Ls1), % test du clash
    resolution(Lie1, Lpt1, Li1, Lu1, Ls1,Abr).

deduction_all(Lie,[(A,all(R,C))|Lpt],Li,Lu,Ls, Abr) :-
    not(member((A,B,R),Abr)),
    % ===== AFFICHAGE =====
    nl, write(">>>>> REGLE deduction_all : "),nl,
    write("aucun changement car il nappartient pas a la liste des assertions de roles ").
    % ======================
```

Donc d'abord le cas de la liste vide, si la liste est vide, on renvoi True et on passe au prédicat suivant, sinon Lpt contient bien un élément de la forme **(A,all(R,C2))** et dans ce cas on peut essayer d'appliquer la règle **all,** pour cela, il faut trouver un B grâce a **member** tel que **<a,b> : R** est dans Abr puis ajouter l'assertion **B : C** avec **evolue.** Ainsi, une fois ces assertions ajoutées à la nouvelle liste Ls1, on teste si ya un clash, si ya un clash (comme vu plus haut) on renvoie directement **False,** sinon True et on fait une nouvelle résolution avec les nouvelles listes. **Si jamais**, **<a,b> : R** n'appartient pas a Abr on ajoute aucune assertion, on renvoie True et on passe à la règle suivante.

Tests du prédicat:

Pour réaliser les tests, on met en commentaire la résolution, car on veut que le programme s'arrête juste après le clash et pour des raisons d'affichages, on a mis en commentaire aussi l'affichage.

-) Test avec un clash : —> Lpt contient un élément et (david, **_B_** , aCree) est bien un élément de la liste Abr avec **B = michelAnge,** on ajoute alors **michelAnge : not(personne)** a Ls1, il ya un **Clash** et on, renvoi **False**.

```
2 ?- deduction_all([],[(david,all(aCree,not(personne)))],[],[],[(michelAnge,personne)],[(david,michelAnge,aCree)]).
false.
```

-)Test sans clash:

> le cas ou **<a,b> : R** n'appartient pas a Abr, —> (david,_B_, aCree) n'appartient pas a Abr : **True**

```
3 ?- deduction_all([],[(david,all(aCree,not(personne)))],[],[],[(michelAnge,personne)],[(david,michelAnge,aEcrit)]).
true.
```

> le cas ou **<a,b> : R** appartient a Abr mais pas de clash : —> (david, **_B_** , aCree) appartient a Abr avec **B = michelAnge,** on ajoute alors **michelAnge : not(personne)** a Ls1, il ya pas de **Clash** et on, renvoie **True**.

```
4 ?- deduction_all([],[(david,all(aCree,not(personne)))],[],[],[(michelAnge,sculpture)],[(david,michelAnge,aCree)]).
true .
```

Transformation_or

Quel est le but de ce prédicat:

Si, on a pas réussi à appliquer aucune des règles précédentes, on cherche à appliquer la dernière règle qui consiste a cherché à appliquer la règle du **Or,** pour ce faire on génère **deux nouveaux noeuds frères** de l'arbre de résolution. Dans l'un, on ajoute l'assertion **a : C** et dans l'autre, on ajoute l'assertion **a : D** . Ces deux nœuds sont les racines respectives de deux nouvelles branches de l'arbre de résolution.

Comment ce prédicat a été implémenté ?:

```
% /\/\/\  transformation_or  /\/\/\  :
transformation_or(Lie,Lpt,Li,[],Ls,Abr).
transformation_or(Lie,Lpt,Li,[(I,or(C1,C2))|Lu],Ls,Abr) :-
    evolue((I,C1), Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1, Li1, Lu1, Ls1),
    % ===== AFFICHAGE =====
    nl, write(">>>>> REGLE transformation_OR 1er branche  : "),nl,
    affiche_evolution_Abox(Ls, Lie, Lpt, Li, [(I,or(C1,C2))|Lu], Abr, Ls1, Lie1, Lpt1, Li1, Lu1, Abr),
    % =====================
    clash(Ls1),
    resolution(Lie1, Lpt1, Li1, Lu1, Ls1,Abr).

transformation_or(Lie,Lpt,Li,[(I,or(C1,C2))|Lu],Ls,Abr) :-
    evolue((I,C2), Lie, Lpt, Li, Lu, Ls, Lie2, Lpt2, Li2, Lu2, Ls2),
    % ===== AFFICHAGE =====
    nl, write(">>>>> REGLE transformation_OR 2e branche  : "),nl,
    affiche_evolution_Abox(Ls, Lie, Lpt, Li, [(I,or(C1,C2))|Lu], Abr, Ls2, Lie2, Lpt2, Li2, Lu2, Abr),
    % =====================
    clash(Ls2), % test du clash
    resolution(Lie2, Lpt2, Li2, Lu2, Ls2,Abr).
```

Donc d'abord le cas de la liste vide, si la liste est vide, on renvoi True et résolution renvoi True, sinon Lu contient bien un élément de la forme **(I,or(C1,C2))** et dans ce cas on doit appliquer la règle **or, à savoir,** générer deux nouveaux noeuds frères et ajouter l'assertion **I : C1** dans le premier et **I : C2** dans le deuxième, une fois ces assertions ajouté à la nouvelle liste Ls1, on teste si ya un clash pour chaque noeud, si ya un clash (comme vu plus haut) on renvoi **False** pour ce noeud mais il faut que l'autre noeud renvoi aussi False pour mettre fin à la résolution, sinon **si un des deux noeuds renvoi True** , on fait une nouvelle résolution avec les nouvelles listes. Pour pouvoir créer les 2 nœuds frères, on met le même nom de prédicats.

Tests du prédicat:

Pour réaliser les tests, on met en commentaire la résolution, car on veut que le programme s'arrête juste après le clash et pour des raisons d'affichages, on a mis en commentaire aussi l'affichage.

-) Test avec un clash : 1er noeud : **clash** entre (david,**sculpteur**) et (david,**not(sculpteur)**) et
le 2e nœud : **clash** entre (david,**not(personne)**) et (david,**personne**) ⇒ on renvoie **False.**

```
10 ?- transformation_or([],[],[],[(david,or(sculpteur,not(personne)))],[(david,personne),(david,not(sculpteur))],[]).
false.
```

-) Test sans clash:

> le cas ou **les 2 noeuds sont ouverts**: les 2 assertions commencent par david, **pas de clash**: **True**

```
8 ?- transformation_or([],[],[],[(david,or(sculpteur,not(personne)))],[(michelAnge,personne)],[]).
true .
```

> le cas ou **1 noeud est ouvert :** pas de clash pour le premier noeud avec (david,sculpteur): **True**

```
9 ?- transformation_or([],[],[],[(david,or(sculpteur,not(personne)))],[(david,personne)],[]).
true .
```

Affiche_evolution_Abox

Quel est le but de ce prédicat:

affiche_evolution_Abox(Ls1, Lie1, Lpt1, Li1, Lu1, Abr1, Ls2, Lie2, Lpt2, Li2, Lu2, Abr2) est le prédicat qui affiche l'évolution d'un état de la Abox étendue (en ce qui concerne les listes des assertions de concepts et la liste des assertions de rôles) vers un état suivant, les 6 premiers paramètres décrivant l'état de départ, les 6 suivants l'état d'arrivée. Abr1 et Abr2 représentent respectivement les états de départ et d'arrivée de l'ensemble des assertions de rôles de la Abox étendue.

Comment ce prédicat a été implémenté ?:

```
% Fonction affichage
affiche_evolution_Abox(Ls1, Lie1, Lpt1, Li1, Lu1, Abr1, Ls2, Lie2, Lpt2, Li2, Lu2, Abr2) :-
    write("><><><><><><><><><>< AVANT ><><><><><><><><><><><><><><><><><><"),nl,
    write("--------------------  Ls1 --------------------"),nl, affiche(Ls1), nl,
    write("--------------------  Lie1 --------------------"),nl, affiche(Lie1), nl,
    write("--------------------  Lpt1 --------------------"),nl, affiche(Lpt1), nl,
    write("--------------------  Li1 --------------------"),nl, affiche(Li1), nl,
    write("--------------------  Lu1 --------------------"),nl, affiche(Lu1), nl,
    write("--------------------  Abr1 --------------------"),nl, afficheRole(Abr1), nl,
    % write("><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><"),nl,
    write("><><><><><><><><><>< APRES ><><><><><><><><><><><><><><><><><><"),nl,
    write("--------------------  Ls2  --------------------"),nl, affiche(Ls2), nl,
    write("--------------------  Lie2 --------------------"),nl, affiche(Lie2), nl,
    write("--------------------  Lpt2 --------------------"),nl, affiche(Lpt2), nl,
    write("--------------------  Li2 --------------------"),nl, affiche(Li2), nl,
    write("--------------------  Lu2 --------------------"),nl, affiche(Lu2), nl,
    write("--------------------  Abr2 --------------------"),nl, afficheRole(Abr2), nl,
    write("-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>"),nl.

affiche([]).
affiche([(I , C)| L]) :- write(I),write(" : "),affiche_concept(C),nl, affiche(L),!.
afficheRole([(A , B , R)| L]) :- write("<"),write(A),write(,),write(B),write(">"), write(" : "), write(R),nl, afficheRole(L),!.
afficheRole([]).
affiche_concept(C) :- setof(X, cnamena(X),L), member(C,L), write(C).
affiche_concept(C) :- setof(X, cnamea(X),L), member(C,L), write(C).
affiche_concept(not(C)) :- write("¬"),affiche_concept(C).
affiche_concept(or(C1,C2)) :- write("(") , affiche_concept(C1) , write(") ⊔ write(") , affiche_concept(C2), write(")").
affiche_concept(and(C1,C2)) :- write("(") , affiche_concept(C1) , write(") ⊓ (") , affiche_concept(C2), write(")").
affiche_concept(all(R,C)) :-  write("∀."),write(R) , write("(") , affiche_concept(C),write(")").
affiche_concept(some(R,C)) :-  write("∃."),write(R) , write("(") , affiche_concept(C),write(")").
```

On implémente ici le dernier prédicat de notre projet qui va renvoyer juste un affichage, à chaque étape de notre résolution, on va afficher les listes avant la modification et après la modification en remplaçant **not** par son symbole, **and** par son symbole etc.

Tests du prédicat:

Ici on teste en exécutant le programme, on choisit le type 1 et on rentre **david : or(sculpture,personne).**

```
4 ?- programme.

Entrez le numero du type de proposition que vous voul
ez demontrer :
1 - Une instance donnee appartient a un concept donne
.
2 - Deux concepts ne possede aucun elements en commun
(ils ont une intersection vide).

|: 1.

entrer linstance
|: david.

entrer le concept
|: or(sculpture,personne).

Abi1 etendu :
[(david,sculpture),(joconde,objet),(michelAnge,person
ne),(sonnets,livre),(vinci,personne),(david,and(not(s
culpture),not(personne)))]
>>>>> REGLE transformation_and :
><><><><><><><><><>< AVANT ><><><><><><><><><><><><><
--------------------  Ls1 --------------------
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne

--------------------  Lie1 --------------------

--------------------  Lpt1 --------------------

--------------------  Li1 --------------------

david : (¬sculpture ⊓ ¬personne)

--------------------  Lu1 --------------------

--------------------  Abr1 --------------------
<michelAnge,david> : aCree
<michelAnge,sonnets> : aEcrit
<vinci,joconde> : aCree

><><><><><><><><><>< APRES ><><><><><><><><><><><><><
--------------------  Ls2  --------------------
david : ¬sculpture
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne

--------------------  Lie2 --------------------

--------------------  Lpt2 --------------------

--------------------  Li2 --------------------

--------------------  Lu2 --------------------

--------------------  Abr2 --------------------
<michelAnge,david> : aCree
<michelAnge,sonnets> : aEcrit
<vinci,joconde> : aCree

-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<
>-

Youpiiiiii, on a demontre la proposition initiale !!!
true.
```

SORBONNE UNIVERSITÉ

Troisieme_etape

Quel est le but de ce prédicat:
Une fois qu'on a implémenté tous les prédicat nécessaire au prédicat **résolution** et le predicat **tri_abox**, on peut implémenter le prédicat **troisieme_etape** qui prend 2 paramètres qui sont respectivement la liste des assertions de concepts complétée et la liste des assertions de rôles qui peut également évoluer lors de la démonstration

Comment ce prédicat a été implémenté ?:

```
/*<><><><><><><><><><><><><><><><><><><><><><><><><> TROISIEME ETAPE <><><><><><><><><><><><><><><><><><><><><><><><><><><>*/
troisieme_etape(Abi,Abr) :-
    tri_Abox(Abi,Lie,Lpt,Li,Lu,Ls),
    not(resolution(Lie,Lpt,Li,Lu,Ls,Abr)),
    nl,write('Youpiiiiii, on a demontre la proposition initiale !!!').
```

On va commencer par appliquer le prédicat tri_abox, qui va nous renvoyer les listes trier qu'on va mettre en parametre a resolution. Ensuite, le prédicat résolution va essayer d'appliquer chaque règle pour trouver des clash et comme expliqué plus haut si il trouve un clash cela **nous renvoi False,** c'est pour cela qu'on met **not(resolution(..))** qui va être égale ainsi à **True** et on va pouvoir renvoyer le message:

"Youpiiiii, on a demontre la proposition initiale !!!!"

Et si résolution renvoi **True**, c'est à dire qu'on n'a pas trouvé de clash, le **not(resolution)** va être égale à **False**, et ainsi **la démonstration n'a pas pu être faite**.

Tests du prédicat:
On va tester cela dans le dernier prédicat qui est **programme.**

——————————VI. Programme ——————————

Quel est le but de ce prédicat:

C'est la fin du projet :(, on a implémenté tous les prédicats nécessaires à la réussite de la premiere_etape, de la deuxieme_etape et enfin de la troisieme_etape. On a donc le prédicat **programme** qui ne prend pas de paramètre et **renvoie la démonstration de la proposition écrite.**

Comment ce prédicat a été implémenté ?:

```
programme :-
    premiere_etape(Tbox,Abi,Abr), % creation de la ABOX et TBOX
    deuxieme_etape(Abi,Abi1,Tbox), % saisie de la proposition a demontrer
    troisieme_etape(Abi1,Abr). % demonstrateur
```

On test d'abord avec la proposition de type 1 :

> ——————————— on a enlevé l'affichage pour avoir la réponse directement ———————————:

```
25 ?- programme.

Entrez le numero du type de proposition que vous voulez demontrer :
1 - Une instance donnee appartient a un concept donne.
2 - Deux concepts ne possede aucun elements en commun(ils ont une intersection vide).

|: 1.

entrer linstance
|: david.

entrer le concept
|: or(not(sculpture),personne).

Abi1 etendu :
[(david,sculpture),(joconde,objet),(michelAnge,personne),(sonnets,livre),(vinci,personne),(david,and(sculpture,not(personne)))]
false.
```

On renvoie **false**, c'est à dire qu'on a pas trouvé de clash et donc la proposition n'as pas pu être démontrée.
Suite à la règle du and, on a pas de clash avec les assertions ajoutées : (david,sculpture) et (david,not(personne)).

Ici, un exemple avec clash, tout les noeuds sont fermées :

```
25 ?- programme.

Entrez le numero du type de proposition que vous voulez demontrer :
1 - Une instance donnee appartient a un concept donne.
2 - Deux concepts ne possede aucun elements en commun(ils ont une intersection vide).

|: 1.

entrer linstance
|: michelAnge.

entrer le concept
|: auteur.

Abi1 etendu :
[(david,sculpture),(joconde,objet),(michelAnge,personne),(sonnets,livre),(vinci,personne),(michelAnge,or(not(personne),all(aEcrit,not(livre))))]
Youpiiiiii, on a demontre la proposition initiale !!!
true.
```

> ——————————————— on fait un test **avec l'affichage** ———————————————:

```
7 ?- programme.

Entrez le numero du type de proposition que vous voulez
demontrer :
1 - Une instance donnee appartient a un concept donne.
2 - Deux concepts ne possede aucun elements en commun(il
s ont une intersection vide).

|: 1.

entrer linstance
|: michelAnge.

entrer le concept
|: sculpteur.

Abi1 etendu :
[(david,sculpture),(joconde,objet),(michelAnge,personne)
,(sonnets,livre),(vinci,personne),(michelAnge,or(not(per
sonne),all(aCree,not(sculpture))))]
>>>>> REGLE transformation_OR 1er branche  :
><><><><><><><><><>< AVANT ><><><><><><><><><><><
--------------------  Ls1 --------------------
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne

--------------------  Lie1 --------------------

--------------------  Lpt1 --------------------

--------------------  Li1 --------------------

--------------------  Lu1 --------------------
michelAnge : (¬personne ⊔ ∀.aCree(¬sculpture))

--------------------  Abr1 --------------------
<michelAnge,david> : aCree
<michelAnge,sonnets> : aEcrit
<vinci,joconde> : aCree

><><><><><><><><><>< APRES ><><><><><><><><><><><
--------------------  Ls2  --------------------
michelAnge : ¬personne
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne

--------------------  Lie2 --------------------

--------------------  Lpt2 --------------------

--------------------  Li2 --------------------

--------------------  Lu2 --------------------

--------------------  Abr2 --------------------
<michelAnge,david> : aCree
<michelAnge,sonnets> : aEcrit
<vinci,joconde> : aCree

-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-

>>>>> REGLE transformation_OR 2e branche  :
><><><><><><><><><>< AVANT ><><><><><><><><><><><
--------------------  Ls1 --------------------
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne

--------------------  Lie1 --------------------

--------------------  Lpt1 --------------------

--------------------  Li1 --------------------

--------------------  Lu1 --------------------
michelAnge : (¬personne ⊔ ∀.aCree(¬sculpture))

--------------------  Abr1 --------------------
<michelAnge,david> : aCree
<michelAnge,sonnets> : aEcrit
<vinci,joconde> : aCree

><><><><><><><><><>< APRES ><><><><><><><><><><><
--------------------  Ls2  --------------------
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne

--------------------  Lie2 --------------------

--------------------  Lpt2 --------------------
michelAnge : ∀.aCree(¬sculpture)

--------------------  Li2 --------------------

--------------------  Lu2 --------------------

--------------------  Abr2 --------------------
<michelAnge,david> : aCree
<michelAnge,sonnets> : aEcrit
<vinci,joconde> : aCree

-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-

>>>>> REGLE deduction_all :
><><><><><><><><><>< AVANT ><><><><><><><><><><><
--------------------  Ls1 --------------------
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne

--------------------  Lie1 --------------------

--------------------  Lpt1 --------------------
michelAnge : ∀.aCree(¬sculpture)

--------------------  Li1 --------------------

--------------------  Lu1 --------------------

--------------------  Abr1 --------------------
<michelAnge,david> : aCree
<michelAnge,sonnets> : aEcrit
<vinci,joconde> : aCree

><><><><><><><><><>< APRES ><><><><><><><><><><><
--------------------  Ls2  --------------------
david : ¬sculpture
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne

--------------------  Lie2 --------------------

--------------------  Lpt2 --------------------

--------------------  Li2 --------------------

--------------------  Lu2 --------------------

--------------------  Abr2 --------------------
<michelAnge,david> : aCree
<michelAnge,sonnets> : aEcrit
<vinci,joconde> : aCree

-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-

Youpiiiiii, on a demontre la proposition initiale !!!
true.
```

On test d'abord avec la proposition de type 2 :

> ——————————— on a enlevé l'affichage pour avoir la réponse directement ———————————:

Premier Test : objet et livre sont 2 concepts avec une intersection vide, on renvoi bien **True**

```
25 ?- programme.

Entrez le numero du type de proposition que vous voulez demontrer :
1 - Une instance donnee appartient a un concept donne.
2 - Deux concepts ne possede aucun elements en commun(ils ont une intersection vide).

|: 2.

Entrer le concept 1 :
|: objet.

Entrer le concept 2 :
|: livre.



Abi1 etendue :
[(david,sculpture),(joconde,objet),(michelAnge,personne),(sonnets,livre),(vinci,personne),(inst1,or(not(objet),not(livre)))]
Youpiiiiii, on a demontre la proposition initiale !!!
true.
```

Deuxième Test : livre et auteur n'ont pas d'intersection vide, car livre appartient a auteur : on renvoie **False.**

```
25 ?- programme.

Entrez le numero du type de proposition que vous voulez demontrer :
1 - Une instance donnee appartient a un concept donne.
2 - Deux concepts ne possede aucun elements en commun(ils ont une intersection vide).

|: 2.

Entrer le concept 1 :
|: livre.

Entrer le concept 2 :
|: auteur.



Abi1 etendue :
[(david,sculpture),(joconde,objet),(michelAnge,personne),(sonnets,livre),(vinci,personne),(inst2,or(not(livre),or(not(personne),all(aEcrit,not(livre)))))]
false.
```

> ——————————————————— on fait un test **avec l'affichage** ———————————————————:

```
7 ?- programme.

Entrez le numero du type de proposition que vous voulez
demontrer :
1 - Une instance donnee appartient a un concept donne.
2 - Deux concepts ne possede aucun elements en commun(il
s ont une intersection vide).

|: 2.

Entrer le concept 1 :
|: personne.

Entrer le concept 2 :
|: livre.



Abi1 etendue :
[(david,sculpture),(joconde,objet),(michelAnge,personne)
,(sonnets,livre),(vinci,personne),(inst1,or(not(personne
),not(livre)))]
>>>>> REGLE transformation_OR 1er branche  :
><><><><><><><><><>< AVANT ><><><><><><><><><><><><><
--------------------  Ls1 --------------------
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne

--------------------  Lie1 --------------------

  --------------------  Lpt1 --------------------

  --------------------  Li1 --------------------

  --------------------  Lu1 --------------------
  inst1 : (¬personne ⊔ ¬livre)

  --------------------  Abr1 --------------------
  <michelAnge,david> : aCree
  <michelAnge,sonnets> : aEcrit
  <vinci,joconde> : aCree

  ><><><><><><><><><>< APRES ><><><><><><><><><><><><
  --------------------  Ls2  --------------------
  inst1 : ¬personne
  david : sculpture
  joconde : objet
  michelAnge : personne
  sonnets : livre
  vinci : personne

  --------------------  Lie2 --------------------

  --------------------  Lpt2 --------------------

  --------------------  Li2 --------------------

  --------------------  Lu2 --------------------

  --------------------  Abr2 --------------------
  <michelAnge,david> : aCree
  <michelAnge,sonnets> : aEcrit
  <vinci,joconde> : aCree
```

```
-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<

>>>>> REGLE transformation_OR 2e branche  :
><><><><><><><><><>< AVANT ><><><><><><><><><><><><><
--------------------  Ls1 --------------------
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne

--------------------  Lie1 --------------------

--------------------  Lpt1 --------------------

--------------------  Li1 --------------------

--------------------  Lu1 --------------------
inst1 : (¬personne ⊔ ¬livre)

--------------------  Abr1 --------------------
<michelAnge,david> : aCree
<michelAnge,sonnets> : aEcrit
<vinci,joconde> : aCree

><><><><><><><><><>< APRES ><><><><><><><><><><><><><
--------------------  Ls2  --------------------
inst1 : ¬livre
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne

--------------------  Lie2 --------------------

--------------------  Lpt2 --------------------

--------------------  Li2 --------------------

--------------------  Lu2 --------------------

--------------------  Abr2 --------------------
<michelAnge,david> : aCree
<michelAnge,sonnets> : aEcrit
<vinci,joconde> : aCree

-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-<>-

Youpiiiiii, on a demontre la proposition initiale !!!
true.
```

VII. Conclusion

Pour conclure, on a mis en place un **algorithme des tableaux pour la logique de description ALC** en divisant le problème en plusieurs étapes. Tout d'abord, **la première étape** consistait à préparer nos données (Abox et Tbox) de l'exercice 3 du td 4 en prenant soin au préalable de vérifier qu'il n'y est pas de bouclage à travers les différents concepts. Ensuite, **la seconde étape** a été implémentée pour permettre à l'utilisateur de saisir la proposition à démontrer, et à l'aide de plusieurs prédicats, à savoir, concept, remplace, ou encore nnf on a pu vérifier la correction sémantique et syntaxique de la proposition saisie et ajouter à la Abox sa négation. Pour finir, la **dernière étape**, qui est l'étape la "plus importante" correspondant au *cœur du démonstrateur*, consistait à montrer que la proposition saisie à l'étape précédente est insatisfiable.. Pour y arriver il a fallu créer un arbre en développant de manière progressive les prédicats avec l'aide de diverses transformations. Ces dernières donnent alors lieux à des valeurs atomique qui pourront être interprétées dans le cas d'une feuille en fonction de leur état (ouverte ou fermée) permettant ainsi de déduire la validité du prédicat saisie. Une remarque pertinente que l'on peut faire dans ce projet est que notre démonstrateur indique uniquement si la proposition a été démontrée ou non mais rien n'indique que la proposition soit fausse. Ce projet nous a surtout permis d'apprendre un nouveau langage, **prolog**, que nous avons trouvé très intéressant et très puissant qui nous a permis de résoudre ce problème de manière plutôt simple, de manière "logique" en suivant les différentes indications données dans le sujet de projet. De plus, grâce à la fonction 'trace.' Le débogage est plus simple que dans d'autres langages car ça nous permet de savoir exactement ce qu'on fait et où va notre programme. L'un des autres avantages est que l'ordre des prédicats n'est pas important. Enfin, ça nous a aussi permis de mieux assimiler la méthode des tableaux sur la logique de description ALC.