SCIENCES SORBONNE UNIVERSITÉ

Projet LRC : Rapport

Ecriture en ProLog d'un démonstrateur basé sur l'algorithme des tableaux pour la logique de description $\mathscr{ALC}$

Nour BOUCHOUCHI | Ella DIJKSMAN

Novembre-Décembre 2022

Table des matières

I. Introduction

Ce projet consiste à programmer en ProLog un démonstrateur basé sur l'algorithme des tableaux pour la logique de description $\mathscr{ALC}$. Pour ce faire, nous allons procéder en trois grandes étapes :

- une étape préliminaire de vérification et de mise en forme de la *TBox* et de la *ABox*,
- une étape de saisie de la proposition à démontrer,
- une étape de démonstration de la proposition.

A l'issue de ces trois étapes, nous aurons construit un démonstrateur capable de démontrer la validité ou non d'une proposition saisie. Le code du démonstrateur se trouve dans le fichier ***projet.pl***.

Le sommaire est constitué des trois grandes parties ainsi que de sous-parties correspondant chacune à un prédicat que l'on explicitera. Afin de faciliter la navigation à l'intérieur du rapport, les lignes de celui-ci sont des liens cliquables qui mènent directement à la partie souhaitée.

II. Partie 1 Etape préliminaire de vérification et de mise en forme de la *TBox* et de la *ABox*

Cette partie de notre projet consiste à vérifier la correction syntaxique et sémantique d'une *TBox* et d'une *ABox*, et à les mettre en forme afin que nous puissions nous en servir comme base de tests pour la suite. On commence donc par implémenter les données qui constitueront notre *TBox* ainsi que notre *ABox* :

```
/*A Box et T Box*/
equiv(sculpteur,and(personne,some(aCree,sculpture))).
equiv(auteur,and(personne,some(aEcrit,livre))).
equiv(editeur,and(personne,and(not(some(aEcrit,livre)),some(aEdite,livre)))).
equiv(parent,and(personne,some(aEnfant,anything))).
cnamea(personne).
cnamea(livre).
cnamea(objet).
cnamea(sculpture).
cnamea(anything).
cnamea(nothing).
cnamena(auteur).
cnamena(editeur).
cnamena(sculpteur).
cnamena(parent).
iname(michelAnge).
iname(david).
iname(sonnets).
iname(vinci).
iname(joconde).
rname(aCree).
rname(aEcrit).
rname(aEdite).
rname(aEnfant).
inst(michelAnge,personne).
inst(david,sculpture).
inst(sonnets,livre).
inst(vinci,personne).
inst(joconde,objet).
instR(michelAnge, david, aCree).
instR(michelAnge, sonnets, aEcrit).
instR(vinci, joconde, aCree).
```

1. Prédicat concept

Le prédicat concept va nous permettre de vérifier la correction syntaxique et sémantique de la *ABox* et de la *TBox*. Ce prédicat doit être défini de façon récursive. Le concept passé en argument doit correspondre à un des éléments de la liste suivante :

- faire partie de la liste des concepts atomiques de la *ABox* (cas de base) ;
- faire partie de la liste des concepts non atomiques de la *ABox* ;
- être la négation d'un concept (*not*) ;
- être l'union de deux concepts (*or*) ;
- être l'intersection de deux concepts (*and*) ;
- dans le cas où l'on vérifie un concept $\forall$ ou $\exists$, il faut vérifier que le premier élément soit bien un rôle (*rname*) et le second un concept.

```
/*Prédicat concept : vérifie la correction syntaxique et sémantique de la ABox et de la TBox.*/
concept(A) :- setof(X,cnamea(X),L), member(A,L),!. /*On vérifie que A est bien un concept atomique (cnamea) de la TBox*/
concept(A) :- setof(X,cnamena(X),L), member(A,L), equiv(A,B), concept(B),!. /*On vérifie que A est un concept non atomique de la TBox*/
concept(not(A)) :- concept(A),!.
concept(or(A,B)) :- concept(A), concept(B),!.
concept(and(A,B)) :- concept(A), concept(B),!.
concept(some(A,B)) :- setof(X,rname(X),L), member(A,L), concept(B),!. /*On vérifie que A est un rôle de la TBox*/
concept(all(A,B)) :-  setof(X,rname(X),L), member(A,L), concept(B),!. /*On vérifie que A est un rôle de la TBox*/
concept(inst(A,B)) :- setof(X1,iname(X1),L1), member(A,L1), /*On vérifie que A est un iname*/
                      setof(X2,cname(X2),L2), member(B,L2),!. /*On vérifie que B est un cname*/
concept(instR(A,B,C)) :- setof(X1,iname(X1),L1), member(A,L1), /*On vérifie que A est un iname*/
                         setof(X2,iname(X2),L2), member(B,L2), /*On vérifie que B est un iname*/
                         setof(X3,rname(X3),L3), member(C,L3),!. /*On vérifie que C est un rname*/
```

On peut alors vérifier les instances de rôles et de concepts :

```
?- concept(and(personne,some(aCree,sculpture))).
true.

?- concept(and(tableau,sculpture)).
false.
```

La première commande renvoie **true** puisque `personne` et `sculpteur` sont bien des concepts et que `aCree` est bien un rôle.

La deuxième en revanche renvoie `false` puisque `tableau` n'est pas un concept.

2. Prédicat autoref

Ce prédicat permet de tester si un concept est autoréférent. En effet, certains concepts non atomiques peuvent avoir dans leur expression conceptuelle équivalente un autre concept non atomique faisant référence à eux-mêmes. Il est ainsi important d'éviter un bouclage dans le traitement des expressions de concepts.

Nous devons donc vérifier si, dans l'expression conceptuelle équivalente d'un concept non atomique, il y a un concept qui fait référence (directement ou indirectement) à ce concept de départ.

Ainsi, dès lors qu'on appelle `autoref` avec en paramètres deux fois le même concept, notre fonction renvoie **`true`** (il s'agit du cas de base).

Dans les cas plus compliqués, il faut appeler récursivement `autoref` sur chaque concept définissant un concept non atomique : il faut vérifier que les deux concepts du *and* et du *or* ou le concept du *not*, *some* ou *all* ne font pas référence à un concept préalablement trouvé. Pour cela, nous stockons dans une liste les concepts déjà explorés et testons à chaque appel de fonction si le concept que l'on étudie fait déjà partie de cette liste.

```
/*Prédicat autoref : teste si un concept est autoréférent, ce qui pourrait faire tourner en boucle le programme*/
autoref(A,A,L):-!. /*Cas de base*/
autoref(A,B,L) :- member(B,L),!. /*Si B est déjà dans la liste alors le concept est autoréférent*/
autoref(A,B,L) :- equiv(B,X), autoref(B,X,[A|L]),!. /*Si B n'est pas un concept atomique, il faut explorer son expression conceptuelle équivalente*/
autoref(A,and(B,C),L) :- autoref(A,B,L),!.
autoref(A,and(B,C),L) :- autoref(A,C,L),!.
autoref(A,or(B,C),L) :- autoref(A,B,L),!.
autoref(A,or(B,C),L) :- autoref(A,C,L),!.
autoref(A,not(B),L) :- autoref(A,B,L),!.
autoref(A,some(R,C),L) :- autoref(A,C,L),!.
autoref(A,all(R,C),L) :- autoref(A,C,L),!.
```

```
?- autoref(sculpteur,sculpteur,[]).
true.

?- autoref(sculpteur,and(sculpteur,auteur),[]).
true.

?- autoref(sculpteur,auteur,[]).
false.
```

On observe dans le premier exemple qu'un concept, ici `sculpteur`, est bien évidemment autoréférent par rapport à lui-même. Le second exemple montre qu'un concept est également autoréférent s'il apparaît - directement ou indirectement - dans son expression conceptuelle équivalente. Enfin, le dernier exemple montre que, si l'on considère que `auteur` est l'expression conceptuelle équivalente de `sculpteur`, il n'y a pas d'autoréférence puisque `auteur` n'est pas autoréférent et que `sculpteur` n'apparaît pas dans l'expression conceptuelle équivalente de `auteur`.

3. *Prédicat* `remplace_cnamena`

Ce prédicat permet de remplacer récursivement une expression de concepts non atomiques par une expression composée uniquement de concepts atomiques.

Le cas de base correspond au cas où un concept est un concept atomique (l'on n'a donc pas besoin de le remplacer).

Lorsqu'un concept est non atomique, il faut appeler `remplace_cnamena` sur son expression conceptuelle.

Lorsque l'expression est une intersection ou une union de concepts, il faut appeler

`remplace_cnamena` sur les deux concepts.

Lorsqu'il s'agit de la négation d'un concept ou d'une expression de la forme *some* ou *all*, il faut appeler `remplace_cnamena` sur le concept afin de, si besoin, l'écrire sous la forme d'une expression de concepts atomiques.

```
/*Prédicat remplace_cnamena : remplace dans une expression complexe les concepts non atomiques
par leur expression conceptuelle composée uniquement de concepts atomiques*/
remplace_cnamena(A,A) :- cnamea(A),!. /*cas de base*/
remplace_cnamena(A,X) :- cnamena(A), equiv(A,B), remplace_cnamena(B,X),!.
remplace_cnamena(and(A,B),X) :- remplace_cnamena(A,R1), remplace_cnamena(B,R2), X=and(R1,R2),!.
remplace_cnamena(or(A,B),X) :- remplace_cnamena(A,R1), remplace_cnamena(B,R2), X=or(R1,R2),!.
remplace_cnamena(not(A),X) :- remplace_cnamena(A,R), X=not(R),!.
remplace_cnamena(some(R,C),some(R,R2)) :- rname(R), remplace_cnamena(C,R2), X=some(R,R2),!.
remplace_cnamena(all(R,C),all(R,R2)) :- rname(R), remplace_cnamena(C,R2), X=all(R,R2),!.
```

```
?- remplace_cnamena(personne,R).
R = personne.

?- remplace(and(sculpteur,auteur),R).
R = and(and(personne, some(aCree, sculpture)),
and(personne, some(aEcrit, livre))).
```

Dans le premier exemple, `personne` étant un concept atomique, l'expression de concepts atomiques équivalente est évidemment `personne`. Lorsque l'on veut effectuer ce traitement sur `and(sculpteur,auteur)`, on observe que les concepts `auteur` et `sculpteur` sont bien remplacés par leur expression équivalente uniquement composée de concepts atomiques.

4. *Prédicat* `traitement_Tbox`

Grâce à ce prédicat, on peut vérifier la correction syntaxique et sémantique d'un concept complexe de la *TBox* et le mettre sous forme *nnf*.

Il faut pour cela commencer par vérifier que le concept passé en argument est bien un concept non atomique afin de connaître son expression conceptuelle. On vérifie alors la correction sémantique et syntaxique du concept non atomique et de son expression conceptuelle. Ensuite, on fait bien attention à ce que ce concept ne soit pas autoréférent, puis on l'écrit sous la forme d'une expression de concepts non atomiques, et enfin on le met sous forme *nnf*.

```
/*Prédicat traitement_Tbox : vérification de la correction syntaxique et sémantique d'un concept complexe de la TBox et mise sous forme nnf*/
traitement_Tbox(A,Res) :- equiv(A,B), /*On récupère un concept non atomique et son expression conceptuelle*/
                          concept(A), concept(B), /*on vérifie la correction synthaxique et sémantique*/
                          not(autoref(A,B,[])), /*on vérifie que le concept n'est pas autoréférent*/
                          remplace_cnamena(B,X), /*l'expression conceptuelle n'est composée que de concepts atomiques*/
                          nnf(X,Res). /*on met l'expression conceptuelle sous forme nnf*/
```

```
?- traitement_Tbox(sculpteur,R).
R = and(personne, some(aCree, sculpture)).
```

5. *Prédicat* `traitement_Abox`

Ce prédicat a pour but de vérifier la correction syntaxique et sémantique d'un concept complexe de la *ABox*.

On distingue la *ABox* conceptuelle et la *ABox* de rôles dans le traitement à effectuer, respectivement au travers des prédicats `traitement_AboxConcept` et `traitement_AboxRole`. Pour la *ABox* conceptuelle, il faut récupérer le concept de l'instance, vérifier sa correction syntaxique et sémantique, puis l'écrire sous forme d'une expression de concepts atomiques et le mettre sous forme *nnf*. Pour la *ABox* de rôles, il s'agit d'effectuer ce même traitement syntaxique et sémantique sur les instances de rôle.

```
/*Prédicat traitement_Abox : vérification de la correction syntaxique et sémantique d'un concept complexe de la ABox
et remplace les concepts non atomiques par leur expression conceptuelle ne contenant que des concepts atomiques*/
traitement_AboxConcept(I,Res) :- inst(I,C), concept(C), remplace_cnamena(C,R), nnf(R,Res).
traitement_AboxRole(I,C,R) :- instR(I,C,R), concept(instR(I,C,R)) .

traitement_Abox(I,R) :- traitement_AboxConcept(I,R).
traitement_Abox(I,(C,R)) :- traitement_AboxRole(I,C,R).
```

```
?- traitement_Abox(joconde,R).
R = objet.

?- traitement_Abox(michelAnge,R).
R = personne ;
R = (david, aCree) ;
R = (sonnets, aEcrit).
```

III. Partie 2 Saisie de la proposition à démontrer

1. *Prédicat* `premiere_etape`

Ce prédicat permet de créer une liste représentant la *TBox*, les assertions de concepts de la *ABox* et les assertions de rôles de la *ABox*.

On utilise le prédicat `setof` afin de mettre dans les listes correspondant à la *TBox*, la *ABox* conceptuelle et la *ABox* de rôles les assertions après traitement.

```
/*Prédicat premiere_etape : liste représentant la TBox, les assertions de concepts de la ABox et les assertions de rôles de la ABox*/
premiere_etape(TBox, ABoxConcept, ABoxRole) :- setof((A,B), traitement_Tbox(A,B), TBox),
                                               setof((I,C), traitement_AboxConcept(I,C), ABoxConcept),
                                               setof((I,C,R), traitement_AboxRole(I,C,R), ABoxRole).
```

```
?- premiere_etape(TBox, ABoxConcept, ABoxRole).
TBox = [(auteur, and(personne, some(aEcrit, livre))),
(editeur, and(personne, and(all(aEcrit, not(livre)),
some(aEdite, livre)))), (parent, and(personne,
some(aEnfant, anything))), (sculpteur, and(personne,
some(aCree, sculpture)))],
ABoxConcept = [(david, sculpture), (joconde, objet),
(michelAnge, personne), (sonnets, livre), (vinci,
personne)],
ABoxRole = [(michelAnge, david, aCree), (michelAnge,
sonnets, aEcrit), (vinci, joconde, aCree)].
```

Ce prédicat permet, comme on le souhaite, de constituer trois listes : celle correspondant à notre *TBox*, celle correspondant à notre *ABox* de concepts et celle correspondant à notre *ABox* de rôles.

2. *Prédicat* `acquisition_prop_type1`

Ce prédicat permet de réaliser l'acquisition d'une proposition de type 1, à savoir de la forme `I:C`. On ajoutera la négation de ce concept à la *ABox* puis l'on cherchera à montrer que ce concept est insatisfiable.

L'utilisateur du solveur doit saisir l'instance puis le concept. La négation du concept est alors mise sous forme d'une expression de concepts atomiques puis sous forme *nnf*. Elle est ensuite ajoutée à la *ABox*.

```
/*Prédicat acquisition_prop_type1*/
acquisition_prop_type1(Abi, Abi1, Tbox) :- nl, write("Saisir instance : "),
                        nl,read(I), /*on récupère l'instance*/
                        nl, write("Saisir concept : "),
                        nl,read(C), /*on récupère le concept*/
                        remplace_cnamena(C,R), /*on remplace de manière récursive les identificateurs de concepts complexes par leur définition*/
                        nnf(not(R),Res), /*on met la négation du concept sous forme nnf*/
                        Abi1 = [(I,Res) | Abi]. /*on ajoute la négation de l'assertion à la ABox*/
```

```
?- acquisition_prop_type1([(david, sculpture),
(joconde, objet), (michelAnge, personne), (sonnets,
livre), (vinci, personne)], Abi1, [(auteur,
and(personne, some(aEcrit, livre))), (editeur,
and(personne, and(all(aEcrit, not(livre)), some(aEdite,
livre)))), (parent, and(personne, some(aEnfant,
anything))), (sculpteur, and(personne, some(aCree,
sculpture)))]).

Saisir instance :
|: joconde.

Saisir concept :
|: objet.

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

Nous donnons ici l'exemple du cas où l'utilisateur souhaite savoir si l'instance `joconde` est un objet. On observe que, suite à la saisie, la négation de l'assertion, à savoir `(joconde, not(objet))`, a bien été ajoutée à la *ABox*.

3. *Prédicat* `acquisition_prop_type2`

Ce prédicat permet de réaliser l'acquisition d'une proposition de type 2, à savoir de la forme `C1` $\sqcap$ `C2`. On ajoutera la négation de ce concept à la *ABox* puis l'on cherchera à montrer que ce concept est insatisfiable.

L'utilisateur du solveur doit saisir le premier puis le second concept. L'intersection de ces deux concepts est alors mise sous forme d'une expression de concepts atomiques puis sous forme *nnf*. La négation de ce concept est ensuite ajoutée à la *ABox*. Cela signifie donc que l'on génère une instance de la négation de ce concept.

```
/*Prédicat acquisition_prop_type2*/
acquisition_prop_type2(Abi, Abi1, Tbox) :- nl, write("Saisir concept C1 : "),
                    nl,read(C1), /*on récupère le concept*/
                    nl, write("Saisir concept C2 : "),
                    nl,read(C2), /*on récupère le concept*/
                    remplace_cnamena(and(C1,C2),R), /*on remplace de manière récursive les identificateurs de concepts complexes par leur définition*/
                    nnf(R,Res), /*on met le concept sous forme nnf*/
                    genere(Inst), /*génère un nom pour l'instance*/
                    Abi1 = [(Inst,Res)| Abi]. /*on ajoute l'assertion*/
```

```
?- acquisition_prop_type2([(david, sculpture),
(joconde, objet), (michelAnge, personne), (sonnets,
livre), (vinci, personne)], Abi1, [(auteur,
and(personne, some(aEcrit, livre))), (editeur,
and(personne, and(all(aEcrit, not(livre)), some(aEdite,
livre)))), (parent, and(personne, some(aEnfant,
anything))), (sculpteur, and(personne, some(aCree,
sculpture)))]).

Saisir concept C1 :
|: objet.

Saisir concept C2 :
|: livre.

Abi1 = [(inst1, and(objet, livre)), (david, sculpture),
(joconde, objet), (michelAnge, personne), (sonnets,
livre), (vinci, personne)] .
```

Dans cet, exemple l'utilisateur souhaite savoir si `objet` $\sqcap$ `livre` $\sqsubseteq \bot$. On observe que la négation de cette proposition, `(inst1, and(objet,livre))`, est ajoutée à la *ABox*.

IV. Partie 3 Démonstration de la proposition

1. *Prédicat* `tri_ABox`

Ce prédicat permet de trier la *ABox* en la décomposant en 5 sous-listes : une contenant les assertion de type `(I,some(R,C))`, une avec les assertions de type `(I,all(R,C))`, une avec les assertions de type `(I,and(C1,C2))`, une avec les assertions de type `(I,or(C1,C2))` et une avec le reste des assertions.

On traite de manière récursive la liste de la *ABox*, afin de mettre un par un les éléments dans la bonne liste, jusqu'à ce que la *ABox* soit vide.

```
/*Prédicat tri_Abox permettant de trier la ABox*/
tri_Abox([],[],[],[],[],[]). /*Cas de base*/
tri_Abox([X|L],[X|Lie],Lpt,Li,Lu,Ls) :- X=(I,some(R,C)), tri_Abox(L,Lie,Lpt,Li,Lu,Ls),!.
tri_Abox([X|L],Lie,[X|Lpt],Li,Lu,Ls) :- X=(I,all(R,C)), tri_Abox(L,Lie,Lpt,Li,Lu,Ls),!.
tri_Abox([X|L],Lie,Lpt,[X|Li],Lu,Ls) :- X=(I,and(C1,C2)), tri_Abox(L,Lie,Lpt,Li,Lu,Ls),!.
tri_Abox([X|L],Lie,Lpt,Li,[X|Lu],Ls) :- X=(I,or(C1,C2)), tri_Abox(L,Lie,Lpt,Li,Lu,Ls),!.
tri_Abox([X|L],Lie,Lpt,Li,Lu,[X|Ls]) :- X\=(I,some(R,C)), X\=(I,all(R,C)),
                                        X\=(I,and(C1,C2)), X\=(I,or(C1,C2)),
                                        tri_Abox(L,Lie,Lpt,Li,Lu,Ls).
```

```
?- tri_Abox([(joconde, not(objet)), (david, sculpture),
(joconde, objet), (michelAnge, personne), (sonnets,
livre), (vinci, personne)], Lie,Lpt,Li,Lu,Ls).
Lie = Lpt, Lpt = Li, Li = Lu, Lu = [],
Ls = [(joconde, not(objet)), (david, sculpture),
(joconde, objet), (michelAnge, personne), (sonnets,
livre), (vinci, personne)].
```

Dans cet exemple, on observe que la liste passée en argument ne comporte que des assertions du type `(I,C)` ou `(I,not(C))`. Il est donc logique que toutes ces assertions soient triées dans la liste `Ls` et que les autres listes soient vides.

2. *Prédicat* `resolution`

Ce prédicat permet de construire notre arbre.
On commence par regarder s'il y a un clash dans la *ABox*, c'est-à-dire si la *ABox* contient un concept et sa négation. Si ce n'est pas le cas, on va chercher à appliquer les différentes règles de résolution possible (la règle *il existe*, la règle d'intersection, la règle *pour tout* puis la règle de l'union). S'il n'y a aucune règle à appliquer et que l'on n'a pas de clash dans la feuille, alors on se trouve dans un noeud ouvert : la proposition initiale est donc fausse. Si par contre, à la fin de l'exploration, tous les noeuds sont fermés, alors on a bien démontré la proposition initiale.

```
/*Prédicat resolution permettant de savoir s'il y un clash dans la ABox ou de continuer l'exploration*/
resolution(Lie,Lpt,Li,Lu,Ls,Abr) :- test_clash(Ls). /*si on a un clash qui renvoie true : il faut désormais savoir si les autres feuilles ont aussi un clash ou non*/
resolution(Lie,Lpt,Li,Lu,Ls,Abr) :- complete_some(Lie,Lpt,Li,Lu,Ls,Abr). /*Si on n'a pas de clash il faut savoir si on peut appliquer l'une des règles de résolution*/
resolution(Lie,Lpt,Li,Lu,Ls,Abr) :- transformation_and(Lie,Lpt,Li,Lu,Ls,Abr).
resolution(Lie,Lpt,Li,Lu,Ls,Abr) :- deduction_all(Lie,Lpt,Li,Lu,Ls,Abr).
resolution(Lie,Lpt,Li,Lu,Ls,Abr) :- transformation_or(Lie,Lpt,Li,Lu,Ls,Abr).
resolution(Lie,Lpt,Li,Lu,Ls,Abr) :- !,nl, write("================ NOEUD OUVERT ================"),/*Si aucune règle de résolution ne peut s'appliquer alors que l'on
                                                                                                                    n'a pas eu de clash, alors la feuille est ouverte, on n'a donc pas
                                                                                                                    démontré l'assertion de base*/
                                    nl, write("Un noeud est ouvert. La proposition est donc fausse."),
                                    nl, notation_infixe(Ls),
                                    nl, write("============================================="),nl,
                                    test_clash(Ls),!. /*renvoie false*/
```

Ce prédicat ne peut pas être testé seul puisqu'il fait appel aux règles de résolution, qui elles-mêmes font appel à `resolution` et que nous verrons un peu plus loin dans notre compte-rendu.

3. *Prédicat* `test_clash`

Ce prédicat permet de tester s'il y a un clash dans la *ABox*.
On ne teste s'il y a un clash que dans la liste de la *ABox* contenant les assertions de type `(I,C)` et `(I,not(C))` puisque les autres listes contiennent des assertions sur lesquelles il faudra appliquer des règles de déduction afin de parvenir à des assertions de type `(I,C)` ou `(I,not(C))`. Pour savoir s'il y a un clash, il faut vérifier s'il y a une instance qui est à la fois une instance de `C` et de `not(C)`. Si c'est le cas, il y a un clash et le prédicat `test_clash(Ls)` renvoie **`true`**.

```
/*Prédicat test_clash permettant de savoir s'il y a un clash*/
test_clash(Ls) :- member((I,X),Ls), member((I,not(X)),Ls),
                  nl,write("================ CLASH ======================="),
                  notation_infixe(Ls),
                  nl,write("============================================="),!.
```

```
?- test_clash([(joconde,objet),(joconde,not(objet))]).

================ CLASH =======================
joconde : objet
joconde : ¬(objet)
=============================================
true.

?- test_clash([(joconde,objet),(vinci,not(objet))]).
false.
```

Dans le premier exemple, on observe qu'il y a un clash puisque la liste comporte à la fois `(joconde, objet)` et `(joconde, not(objet))`. Le prédicat retourne donc **true**.
En revanche, dans le second, le prédicat retourne false puisqu'il n'y a pas de clash

(on a bien une instance `objet` et une instance de `not(objet)` mais ces deux instances sont distinctes).

4. *Prédicat* `complete_some`

Ce prédicat permet d'appliquer la règle *il existe* sur la *ABox*.
Pour appliquer ce prédicat, Il faut que la liste contenant des assertions de type `(I,some(R,C))` ne soit pas vide. S'il existe au moins une expression de ce type, on génère une instance puis on ajoute l'assertion `(I,C)` à la *ABox* à l'aide du prédicat `evolue`.

On affiche ensuite l'évolution de la *ABox* puis on applique à nouveau le prédicat `resolution` sur la nouvelle *ABox* (contenant la nouvelle assertion et ne contenant plus l'assertion sur laquelle on a appliqué la règle *il existe*) afin de tester si la modification apportée à la *ABox* provoque un clash et, si ce n'est pas le cas, d'appliquer les règles de résolution (lorsque c'est possible) sur la nouvelle *ABox*.

```
/*Prédicat complete_some cherchant une assertion de concept de la forme (I,some(R,C)) dans Lie*/
complete_some([(A,some(R,C))|Lie],Lpt,Li,Lu,Ls,Abr) :- genere(B), /*On génère une instance B*/
                                evolue((B,C),Lie,Lpt,Li,Lu,Ls,Lie1,Lpt1,Li1,Lu1,Ls1), /*On ajoute l'assertion de concept*/
                                nl, write("======= Application de la règle some ======="),
                                nl, affiche_evolution_Abox(Ls, [(A,some(R,C))|Lie], Lpt, Li, Lu, Abr, Ls1, Lie1, Lpt1, Li1, Lu1, [(A,B,R)|Abr]),
                                nl,write("=========================================="),nl,
                                !,resolution(Lie1, Lpt1, Li1, Lu1, Ls1, [(A,B,R)|Abr]). /*On ajoute l'assertion de rôle*/
```

```
?- complete_some([(vinci,some(aEcrit,livre))],[],[],[],[],[]).

======= Application de la regle some =======

====== Avant ======
== ABox concept :
vinci : ∃aEcrit.(livre)
== ABox role :

====== Apres ======
== ABox concept :
inst1 : livre
== ABox role :
<vinci,inst1> : aEcrit

=============================================

================ NOEUD OUVERT ================
Un noeud est ouvert. La proposition est donc fausse.

inst1 : livre
=============================================
false.
```

On observe ici que l'assertion `vinci : ∃aEcrit.(livre)` est bien remplacée dans la *Abox* par `inst1 : livre` et `<vinci,inst1> : aEcrit`.

Notons que dans ce test, le prédicat appelle ensuite `resolution`, ainsi la résolution continue jusqu'à la fin. Il en sera de même pour les trois autres règles qui vont suivre.

5. *Prédicat* `transformation_and`

Ce prédicat permet d'appliquer la règle de l'intersection sur la *ABox*.
Pour appliquer ce prédicat, il faut que la liste contenant des assertions de type `(I,and(C1,C2))` ne soit pas vide. S'il existe au moins une assertion de ce type, on ajoute l'assertion `(A,C1)` puis `(A,C2)` à la *ABox*. On peut ensuite afficher l'évolution de la *ABox*. On applique alors le prédicat `resolution` sur la nouvelle *ABox* (contenant les deux nouvelles assertions et ne contenant plus l'assertion sur laquelle on a appliqué la règle de l'intersection) afin de tester si la modification apportée à la *ABox* provoque un clash. Si ce n'est pas le cas, on applique les règles de résolution, lorsque c'est possible, sur la nouvelle *ABox*.

```
/*Prédicat transformation_and cherchant une assertion de concept de la forme (I,and(C1,C2)) dans Li*/
transformation_and(Lie,Lpt,[(A,and(C,D))|Li],Lu,Ls,Abr) :- evolue((A,C),Lie,Lpt,Li,Lu,Ls,Lie1,Lpt1,Li1,Lu1,Ls1),
                                evolue((A,D),Lie1,Lpt1,Li1,Lu1,Ls1,Lie2,Lpt2,Li2,Lu2,Ls2),
                                nl, write("======= Application de la règle intersection ======="),
                                nl, affiche_evolution_Abox(Ls, Lie, Lpt, [(A,and(C,D))|Li], Lu, Abr, Ls2, Lie2, Lpt2, Li2, Lu2, Abr),
                                nl, write("================================================"),nl,
                                !,resolution(Lie2,Lpt2,Li2,Lu2,Ls2,Abr). /*On ajoute à Ls les deux assertions et l'on résoud*/
```

```
?- transformation_and([],[],[(parent,and(personne,
some(aEnfant,anything)))],[],[],[]).

======= Application de la regle intersection =======

====== Avant ======
== ABox concept :
parent : (personne ⊓ ∃aEnfant.(anything))
== ABox role :

====== Apres ======
== ABox concept :
parent : personne
parent : ∃aEnfant.(anything)
== ABox role :

=====================================================
```

(Suite page suivante)

(Suite)

```
======= Application de la regle some =======

====== Avant ======
== ABox concept :
parent : personne
parent : ∃aEnfant.(anything)
== ABox role :
====== Apres ======
== ABox concept :
inst2 : anything
parent : personne
== ABox role :
<parent,inst2> : aEnfant

=============================================

================= NOEUD OUVERT ==================
Un noeud est ouvert. La proposition est donc fausse.

inst2 : anything
parent : personne
================================================

================= NOEUD OUVERT ==================
Un noeud est ouvert. La proposition est donc fausse.

parent : personne
================================================
false.
```

Ici, l'assertion `parent : (personne ⊓ ∃aEnfant.(anything))` est bien remplacée dans la *ABox* par `parent : personne` ainsi que par `parent : ∃aEnfant.(anything)`.

6. *Prédicat* `deduction_all`

Ce prédicat permet d'appliquer la règle *pour tout* sur la *ABox*.
Pour appliquer ce prédicat, il faut que la liste des assertions de type `(A,all(R,C))` ne soit pas vide. S'il existe au moins une assertion de ce type alors on cherche dans la *ABox* des rôles s'il y a une assertion de la forme `(A,B,R)`. Si ce n'est pas le cas on applique le prédicat resolution sur la *ABox* ne contenant plus l'assertion sur laquelle on vient d'essayer d'appliquer la règle du *pour tout*. En revanche, s'il existe bien une

assertion de cette forme dans la *ABox* de rôles, on ajoute l'assertion `(B,C)` à notre *ABox* et on enlève cette assertion de la *ABox* de rôles. On peut ensuite afficher l'évolution de la *ABox*. Enfin, on applique à nouveau la règle du *pour tout* sur l'assertion `(A,all(R,C))` afin de faire en une fois toutes les déductions possibles.

```
/*Prédicat deduction_all cherchant une assertion de concept de la forme (I,all(R,C)) dans Lpt*/

deduction_all(Lie,[(A,all(R,C))|Lpt],Li,Lu,Ls,Abr) :- not(member((A,B,R), Abr)), /*S'il n'y a pas d'assertion de la forme <a,b>:R on résoud */
                        !,resolution(Lie, Lpt, Li, Lu, Ls, Abr).

deduction_all(Lie,[(A,all(R,C))|Lpt],Li,Lu,Ls,Abr) :- member((A,B,R), Abr), /*Tant qu'il y a une assertion de la forme <a,b>:R ...*/
                        evolue((B,C),Lie,Lpt,Li,Lu,Ls,Lie1,Lpt1,Li1,Lu1,Ls1), /*... on ajoute à Abe b:C */
                        enleve((A,B,R), Abr, Abr1),
                        nl,write("======= Application de la règle pour tout ======="),
                        nl,affiche_evolution_Abox(Ls, Lie, [(A,all(R,C))|Lpt], Li, Lu, Abr, Ls1, Lie1, Lpt1, Li1, Lu1, Abr1),
                        nl,write("================================================"),nl,
                        !,deduction_all(Lie1,[(A,all(R,C))|Lpt1],Li1,Lu1,Ls1,Abr1). /*On garde bien L'assertion (I,all(R,C)) en tête de liste*/
```

```
?- deduction_all([],[(auteur,all(aEcrit,livre))],[],[],
[],[(auteur,codex,aEcrit)]).

======= Application de la regle pour tout =======

====== Avant ======
== ABox concept :
auteur : ∀aEcrit.(livre)
== ABox role :
<auteur,codex> : aEcrit

====== Apres ======
== ABox concept :
codex : livre
== ABox role :

================================================

================ NOEUD OUVERT ================
Un noeud est ouvert. La proposition est donc fausse.

codex : livre
================================================
false.
```

On observe que l'assertion `auteur : ∀aEcrit.(livre)` est remplacée dans la *ABox* par `codex : livre`. Cela s'explique par le fait qu'il existe dans la *ABox* de rôles une assertion `<auteur,codex> : aEcrit`.

7. *Prédicat* `transformation_or`

Ce prédicat permet d'appliquer la règle de l'union sur la *ABox*.
Pour appliquer ce prédicat, il faut que la liste des assertion de type `(I,or(C1,C2))` ne soit pas vide. S'il existe au moins une assertion de ce type, on peut appliquer dessus la règle de l'union. Pour cela, il faut générer deux noeuds : l'un dans lequel on aura ajouté l'assertion `(I,C1)` à notre *ABox* (et enlevé l'assertion sur laquelle on applique la règle), l'autre dans lequel on aura ajouté l'assertion `(I,C2)` à notre *ABox* (et enlevé l'assertion sur laquelle on applique la règle). Pour ces deux nouveaux noeuds on affiche l'évolution de la *ABox* puis l'on applique le prédicat `resolution` sur la *ABox* modifiée. L'exploration de l'arbre se poursuit donc dans les deux nouvelles branches ainsi créées.

```
/*Prédicat transformation_or cherchant une assertion de concept de la forme (I,or(C1,C2)) dans Lu*/
transformation_or(Lie,Lpt,Li,[(A,or(C,D))|Lu],Ls,Abr) :- evolue((A,C),Lie,Lpt,Li,Lu,Ls,Lie1,Lpt1,Li1,Lu1,Ls1), /*On ajoute l'assertion (A,C) dans la ABox*/
                                                         nl,write("======= Application de la regle OU (1) ======="),
                                                         nl,affiche_evolution_Abox(Ls, Lie, Lpt, Li, [(A,or(C,D))|Lu], Abr, Ls1, Lie1, Lpt1, Li1, Lu1, Abr),
                                                         nl,write("=============================================="),nl,
                                                         !,resolution(Lie1, Lpt1, Li1, Lu1, Ls1, Abr), /*On résoud une branche*/
                                                         evolue((A,D),Lie,Lpt,Li,Lu,Ls,Lie2,Lpt2,Li2,Lu2,Ls2), /*On ajoute l'assertion (A,D) dans la ABox*/
                                                         nl,write("======= Application de la regle OU (2) ======="),
                                                         nl,affiche_evolution_Abox(Ls, Lie, Lpt, Li, [(A,or(C,D))|Lu], Abr, Ls2, Lie2, Lpt2, Li2, Lu2, Abr),
                                                         nl,write("=============================================="),nl,
                                                         !,resolution(Lie2, Lpt2, Li2, Lu2, Ls2, Abr). /*On résoud une branche*/
```

```
?- transformation_or([],[],[],[(joconde,or(not(personne),
all(aEcrit,not(livre))))],[(david,sculpture),
(joconde,objet), (michelAnge,personne),(sonnets,livre),
(vinci,personne)], [(michelAnge,david,aCree),
(michelAnge,sonnets,aEcrit),(vinci,joconde,aCree)]).

======= Application de la regle OU (1) =======

====== Avant ======
== ABox concept :
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne
joconde : (¬(personne) ⊔ aEcrit.(¬(livre)))
== ABox role :
<michelAnge,david> : aCree
<michelAnge,sonnets> : aEcrit
<vinci,joconde> : aCree
```

(Suite page suivante)

(Suite)

```
====== Apres ======
== ABox concept :
joconde : ¬(personne)
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne
== ABox role :
<michelAnge,david> : aCree
<michelAnge,sonnets> : aEcrit
<vinci,joconde> : aCree

================================================

================= NOEUD OUVERT ==================
Un noeud est ouvert. La proposition est donc fausse.
joconde : ¬(personne)
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne
================================================
false.
```

On observe que l'assertion `joconde :(¬(personne) aEcrit.(¬(livre)))` est remplacée dans le premier noeud du *ou* par `joconde : ¬(personne)`. Etant donné qu'un noeud est ouvert, le programme s'arrête. Néanmoins, si cela n'avait pas été le cas, on aurait pu voir dans le second noeud une assertion de la forme `joconde : aEcrit.(¬(livre))`.

8. *Prédicat* `evolue`

Ce prédicat permet d'insérer une assertion à la bonne sous-liste de la *ABox*. Il suffit de regarder la forme de l'assertion (*some*, *all*, *and*, *or* ou autre) puis d'ajouter le prédicat à la liste correspondante.

```
/*Prédicat evolue intégrant dans l'une des listes l'assertion A*/
evolue((I,some(R,C)),Lie,Lpt,Li,Lu,Ls,Lie1,Lpt,Li,Lu,Ls) :- Lie1=[(I,some(R,C))|Lie],!.
evolue((I,all(R,C)),Lie,Lpt,Li,Lu,Ls,Lie,Lpt1,Li,Lu,Ls) :- Lpt1=[(I,all(R,C))|Lpt],!.
evolue((I,and(C1,C2)),Lie,Lpt,Li,Lu,Ls,Lie,Lpt,Li1,Lu,Ls) :- Li1=[(I,and(C1,C2))|Li],!.
evolue((I,or(C1,C2)),Lie,Lpt,Li,Lu,Ls,Lie,Lpt,Li,Lu1,Ls) :- Lu1=[(I,or(C1,C2))|Lu],!.
evolue(X,Lie,Lpt,Li,Lu,Ls,Lie,Lpt,Li,Lu,Ls1) :- X\=(I,some(R,C)), X\=(I,all(R,C)),
                                                X\=(I,and(C1,C2)), X\=(I,or(C1,C2)),
                                                Ls1=[X|Ls].
```

```
?- evolue((a,and(b,c)),[],[],[],[],[],Lie,Lpt,Li,Lu,Ls).
Lie = Lpt, Lpt = Lu, Lu = Ls, Ls = [],
Li = [(a, and(b, c))].

?- evolue((a,some(b,c)),[],[],[],[],[],Lie,Lpt,Li,Lu,Ls).
Lie = [(a, some(b, c))],
Lpt = Li, Li = Lu, Lu = Ls, Ls = [].

?- evolue((a,not(b)),[],[],[],[],[],Lie,Lpt,Li,Lu,Ls).
Lie = Lpt, Lpt = Li, Li = Lu, Lu = [],
Ls = [(a, not(b))].
```

9. *Prédicat* `affiche_evolution_Abox`

Ce prédicat permet permet d'afficher l'évolution de la *ABox* suite à l'application d'une règle de résolution.

Plus exactement, il affiche l'état de chaque liste de la *ABox* (conceptuelle et de rôles) avant et après application d'une règle de résolution. Pour des raisons de lisibilité du contenu de la *ABox*, nous avons créé deux fonctions : `notation_infixe` et `afficheAbr`, que nous présenterons par la suite, permettant de formater respectivement la *ABox* conceptuelle et de rôles.

```
/*Prédicat affiche_evolution_Abox*/
affiche_evolution_Abox(Ls1, Lie1, Lpt1, Li1, Lu1, Abr1, Ls2, Lie2, Lpt2, Li2, Lu2, Abr2) :-
  nl, write("====== Avant ======"),
  nl, write("== ABox concept : "),
  notation_infixe(Ls1),
  notation_infixe(Lie1),
  notation_infixe(Lpt1),
  notation_infixe(Li1),
  notation_infixe(Lu1),
  nl, write("== ABox rôle : "),
  afficheAbr(Abr1),
  nl,nl, write("====== Après ======"),
  nl, write("== ABox concept : "),
  notation_infixe(Ls2),
  notation_infixe(Lie2),
  notation_infixe(Lpt2),
  notation_infixe(Li2),
  notation_infixe(Lu2),
  nl,write("== ABox rôle : "),
  afficheAbr(Abr2),nl.
```

```
?- affiche_evolution_Abox([(david,sculpture),
(joconde,objet),(michelAnge,personne)], [], [], [],
[], [(michelAnge,david,aCree),(vinci,joconde,aCree)],
[(david,sculpture), (joconde,objet),(michelAnge,personne),
(david,not(objet))], [], [], [], [], [(michelAnge,david,aCree),
(vinci,joconde,aCree)]).

====== Avant ======
== ABox concept :
david : sculpture
joconde : objet
michelAnge : personne
== ABox role :
<michelAnge,david> : aCree
<vinci,joconde> : aCree

====== Apres ======
== ABox concept :
david : sculpture
joconde : objet
michelAnge : personne
david : ¬(objet)
```

(Suite page suivante)

(Suite)
```
== ABox role :
<michelAnge,david> : aCree
<vinci,joconde> : aCree
true.
```

On a affiché dans un premier temps la première *ABox* (de concepts et de rôles) puis la seconde avec un affichage plus lisible que dans des listes.

10. Prédicat `notation_infixe`

Ce prédicat permet de mettre une expression de la *ABox* en notation infixe à l'aide des symboles ∃, ⊔, ¬, ∀ et ⊓.

Pour chaque assertion, le prédicat `notation_infixe` est appelé résursivement afin de réécrire les concepts jusqu'à ce qu'il n'y ait plus rien à modifier (c'est-à-dire jusqu'à ce que l'on arrive à un concept atomique qu'il faut donc retranscrire tel quel). Pour ce faire, il remplace toutes les assertions de la liste par appel récursif jusqu'à obtenir une liste vide.

```
/*Le prédicat notation_infixe permet de transformer une notation préfixe en notation infixe*/
notation_infixe([]).
notation_infixe([(I,A)|L]) :- nl, write(I), write(" : "), notation_infixe(A), notation_infixe(L).
notation_infixe(C) :- cnamea(C), write(C).
notation_infixe(some(R,C)) :- write("∃"), write(R), write("."), notation_infixe(C), write(")").
notation_infixe(or(C1,C2)) :- write("("), notation_infixe(C1), write(" ⊔ "), notation_infixe(C2), write(")").
notation_infixe(not(A)) :- write("¬("), notation_infixe(A), write(")").
notation_infixe(all(R,C)) :- write("∀"), write(R), write("."), notation_infixe(C), write(")").
notation_infixe(and(C1,C2)) :- write("("), notation_infixe(C1), write(" ⊓ "), notation_infixe(C2), write(")").
```

```
?- notation_infixe([(sculpteur,
and(personne,some(aCree,sculpture))), (joconde,
objet), (michelAnge, personne), (sonnets, livre),
(vinci, personne)]).

sculpteur : (personne ⊓ ∃aCree.(sculpture))
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne
true.
```

On affiche ici en notation infixe les éléments de la *ABox* conceptuelle qui étaient préalablement sous forme de liste. On a noté l'instance suivie du concept sous forme infixe.

11. Prédicat afficheAbr

Ce prédicat permet de réécrire la liste de la *ABox* de rôles sous une forme plus lisible.

Plus précisément, il réécrit la première assertion de la *ABox* de rôles puis fait de même sur le reste de la liste par un appel récursif à `afficheAbr`.

```
afficheAbr([]).
afficheAbr([(A,B,R)|L]) :- nl, write("<"), write(A), write(","), write(B), write("> : "), write(R), afficheAbr(L).
```

```
?- afficheAbr([(michelAnge, david, aCree),
(michelAnge, sonnets, aEcrit), (vinci, joconde,
aCree)]).

<michelAnge,david> : aCree
<michelAnge,sonnets> : aEcrit
<vinci,joconde> : aCree
true.
```

On affiche grâce à ce prédicat les éléments de la *ABox* de rôles sous la forme `<instance1, instance2> : rôle`.

12. Prédicat programme

Le prédicat `programme` est le résultat de notre projet. Il permet d'entamer le déroulement de la démonstration en faisant appel aux autres prédicats.

```
/*Prédicat programme : correspond au point d'entrée du programme*/
programme :- premiere_etape(Tbox,Abi,Abr),
             deuxieme_etape(Abi,Abi1,Tbox),
             troisieme_etape(Abi1,Abr).
```

Voici quelques tests parmi ceux que l'on a effectués afin de vérifier le bon fonctionnement de notre démonstrateur.

— ***Test 1***

```
?- programme.

Entrez le numero du type de proposition que vous voulez
demontrer :
1 Une instance donnee appartient a un concept donne.
2 Deux concepts n'ont pas d'elements en commun (ils ont
une intersection vide)
|: 1.

Saisir instance :
|: joconde.

Saisir concept :
|: objet.

================= CLASH ======================
joconde : ¬(objet)
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne
==============================================
Youpiiiiii, on a demontre la proposition initiale !!!
true.
```

— ***Test 2***

```
?- programme.

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

(Suite page suivante)

(Suite)

```
|: 1.

Saisir instance :
|: joconde.

Saisir concept :
|: auteur.

======= Application de la regle OU (1) =======

====== Avant ======
== ABox concept :
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne
joconde : (¬(personne) ⊔ ∀aEcrit.(¬(livre)))
== ABox role :
<michelAnge,david> : aCree
<michelAnge,sonnets> : aEcrit
<vinci,joconde> : aCree

====== Apres ======
== ABox concept :
joconde : ¬(personne)
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne
== ABox role :
<michelAnge,david> : aCree
<michelAnge,sonnets> : aEcrit
<vinci,joconde> : aCree

=============================================

================= NOEUD OUVERT ==================
```

(Suite page suivante)

```
(Suite)
Un noeud est ouvert. La proposition est donc fausse.

joconde : ¬(personne)
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne
================================================

================= NOEUD OUVERT ==================
Un noeud est ouvert. La proposition est donc fausse.

david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne
================================================
false.
```

— ***Test 3***

```
?- programme.

Entrez le numero du type de proposition que vous
voulez demontrer :
1 Une instance donnee appartient a un concept
donne.
2 Deux concepts n'ont pas d'elements en commun (ils
ont une intersection vide)
|: 2.

Saisir concept C1 :
|: personne.

Saisir concept C2 :
|: livre.

(Suite page suivante)
```

(Suite)

```
======= Application de la regle intersection =======

====== Avant ======
== ABox concept :
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne
inst9 : (personne ⊓ livre)
== ABox role :
<michelAnge,david> : aCree
<michelAnge,sonnets> : aEcrit
<vinci,joconde> : aCree

====== Apres ======
== ABox concept :
inst9 : livre
inst9 : personne
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne
== ABox role :
<michelAnge,david> : aCree
<michelAnge,sonnets> : aEcrit
<vinci,joconde> : aCree

=====================================================

================= NOEUD OUVERT ==================
Un noeud est ouvert. La proposition est donc fausse.

inst9 : livre
inst9 : personne
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne
```

(Suite page suivante)

```
(Suite)
================================================

================= NOEUD OUVERT ==================
Un noeud est ouvert. La proposition est donc fausse.

david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne
================================================
false.
```

— ***Test 4***

```
?- programme.

Entrez le numero du type de proposition que vous voulez
demontrer :
1 Une instance donnee appartient a un concept donne.
2 Deux concepts n'ont pas d'elements en commun (ils ont
une intersection vide)
|: 2.

Saisir concept C1 :
|: objet.

Saisir concept C2 :
|: livre.

======= Application de la regle intersection =======

====== Avant ======
== ABox concept :
david : sculpture
joconde : objet
michelAnge : personne

(Suite page suivante)
```

(Suite)

```
sonnets : livre
vinci : personne
inst1 : (objet ⊓ livre)
== ABox role :
<michelAnge,david> : aCree
<michelAnge,sonnets> : aEcrit
<vinci,joconde> : aCree

====== Apres ======
== ABox concept :
inst1 : livre
inst1 : objet
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne
== ABox role :
<michelAnge,david> : aCree
<michelAnge,sonnets> : aEcrit
<vinci,joconde> : aCree

=====================================================

================= NOEUD OUVERT ==================
Un noeud est ouvert. La proposition est donc fausse.

inst1 : livre
inst1 : objet
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne
=================================================

================= NOEUD OUVERT ==================
Un noeud est ouvert. La proposition est donc fausse.
```

(Suite page suivante)

(Suite)

```
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne
================================================
false.
```

V. Conclusion

Lors de ce projet, nous avons mis en place un démonstrateur basé sur l'algorithme des tableaux pour la logique de description $\mathscr{ALC}$. Le point d'entrée de ce démonstrateur est le prédicat programme qui se découpe en trois étapes. Lors de la première étape, nous avons préparé les données qui allaient constituer notre *ABox* et notre *TBox*.

La seconde partie permet ensuite à l'utilisateur du démonstrateur de saisir la proposition qu'il souhaite démontrer. Il a alors fallu vérifier la correction syntaxique et sémantique de l'assertion entrée par ce dernier.

Enfin, la dernière étape correspond à la démonstration en elle-même de l'assertion. Le but est de savoir si la négation de l'assertion entrée par l'utilisateur est ou non satisfiable. Dans le cas où elle est insatisfiable il est alors possible de conclure que l'assertion ayant été transmise par l'utilisateur est valide. Cette démonstration se base sur la création d'un arbre dont l'état des feuilles (ouvert ou fermé) permet de déduire la validité de l'assertion.

Ce projet nous a permis de nous familiariser avec le langage ProLog et l'usage de la récursion. Nous avons également pu, grâce à celui-ci, bien assimiler la méthode des tableaux en $\mathscr{ALC}$ en en décomposant le fonctionnement.