Rapport de projet

SORBONNE UNIVERSITÉ

# *Projet LRC*

## *Démonstrateur basé sur l'algorithme des tableaux pour la logique de description ALC*

Binôme :

- **Amayas SADI**
- **Ghiles OUHENIA**

# Contents

## Exécution du programme

Pour utiliser notre démonstrateur, faut avoir prolog d'installer.

Lancer la commande swipl sur le terminal

- Charger le programme principal (main.pl) avec la commande "[main]."
- charger les données (data.pl) avec "[data]."
- lancer le programme de résolution en lançant "programme."

Pour changer le jeu de données il suffit de modifier le fichier data.pl .

le prédicat programme ci dessous :

```
programme :-
    premiere_etape(Tbox, Abi, Abr),
    deuxieme_etape(Abi, NAbi, Tbox),
    troisieme_etape(NAbi, Abr).
```

vous demandera de choisir le type de proposition à démontrer, ensuite vous demandera de saisir cette dernière, et lancera la résolution en affichant les différentes étapes de démonstration (création des nœuds), et affichera le résultat de la démonstration.

# 1 Étape préliminaire de vérification et de mise en forme de la Tbox et de la Abox

## 1.1 autoref

L'auto-référence est le fait de retrouver un concept dans son expression équivalente, c'est-à-dire l'existence des cycles dans les définitions des concepts.

Pour le bon déroulement de notre démonstrateur, il est primordial de s'assurer que l'ensemble des clauses ne comporte pas d'auto-référence.

Pour cela, on utilise le prédicat autoref qui est basé sur le prédicat pas_autoref, qui lui prend deux arguments un identificateur de concept non atomique, et son expression équivalente dans la Tbox et vérifie si le concept non atomique est répété dans son expression équivalente.

Comme l'invocation du prédicat autoref se fait avant la génération de la Tbox, on extrait les expressions équivalentes directement depuis le fichier des données (en utilisant equiv).

```
pas_autoref(C, and(C1, C2)) :- pas_autoref(C, C1), pas_autoref(C, C2), !.
pas_autoref(C, or(C1, C2)) :- pas_autoref(C, C1), pas_autoref(C, C2), !.
pas_autoref(C, some(_, C1)) :-  pas_autoref(C, C1), !.
pas_autoref(C, all(_, C1)) :-  pas_autoref(C, C1), !.
pas_autoref(C, not(C1)) :-  pas_autoref(C, C1), !.
pas_autoref(C, C1) :-  cnamena(C1), C \== C1, equiv(C1, E), pas_autoref(C, E), !.
pas_autoref(_, C1) :-  cnamea(C1).

autoref(C, E) :- not(pas_autoref(C, E)).
```

## 1.2 concept

Pour vérifier syntaxiquement et sémantiquement la Tbox et la Abox en entrées, c'est-à-dire que toutes les clauses sont composées d'expressions bien formées, on utilise le prédicat concept.

Le prédicat concept vérifie si son argument est syntaxiquement correct (la structure du prédicat est inspirée de la grammaire des concepts en ALC), anything et nothing sont directement contenus dans cnamea.

```
concept(not(X)) :- concept(X), !.
concept(or(X, Y)) :- concept(X), concept(Y), !.
concept(and(X, Y)) :- concept(X), concept(Y), !.
concept(some(R, X)) :- rname(R), concept(X), !.
concept(all(R, X)) :- rname(R), concept(X), !.
concept(X) :- cnamena(X), !.
concept(X) :- cnamea(X), !.
```

## 1.3 traitement_Tbox

Notre démonstrateur a besoin de traiter la Tbox, c'est-à-dire, réécrire toute expression en expression contenant que des concepts **atomiques**, et sous **forme normale négative**.

On a ajouté un argument Tbox initiale pour le traitement de la Tbox, afin d'avoir en notre possession toutes les expressions des concepts non atomiques existantes.

```
traitement_Tbox(Tbox, NTbox) :- traitement_Tbox_rec(Tbox, Tbox, NTbox), !.
```

### 1.3.1 traitement_Tbox_rec

Le traitement de la Tbox consiste à :

- Vérifie la correction syntaxique et sémantique d'une expression en utilisant les prédicats cnamena et concept.
- Développe les expressions des concepts avec le prédicat developper_expression.
- Transforme les expressions en forme normale négative.

```
traitement_Tbox_rec(_, [], []).

traitement_Tbox_rec(_, [(C, _) | _], [(C, _) | _]) :-
    cnamea(C),
    write('Attention ! '),
    write(C),
    write(' est un concept atomique, il ne peut pas avoir de definition'),
    nl, nl, halt, !.

traitement_Tbox_rec(TboxInit, [(C, E) | Tbox], [(C, NNFDE) | NTbox]) :-
    cnamena(C),
    developper_expression(TboxInit, E, DE),
    concept(DE),
    nnf(DE, NNFDE),
    traitement_Tbox_rec(TboxInit, Tbox, NTbox), !.

traitement_Tbox_rec(_, [(C, _) | _], _) :-
    nl, write(C),
    write(' n\'existe pas ou n\'est pas syntaxiquement correct.'),
    nl, nl, halt.
```

### 1.3.2 developper_expression

Comme vu précédemment, afin de traiter la Tbox, on a besoin de développer les expressions de chaque concept, ce qui se fait avec le prédicat developper_expression.

Il prend en argument une expression qui sera remplacée par une expression où ne figurent plus que des identificateurs de concepts atomiques, en se basant sur la Tbox fournie.

```
developper_expression(Tbox, and(C1, C2), and(DC1, DC2)) :-
    developper_expression(Tbox, C1, DC1),
    developper_expression(Tbox, C2, DC2), !.

developper_expression(Tbox, or(C1, C2), or(DC1, DC2)) :-
    developper_expression(Tbox, C1, DC1),
    developper_expression(Tbox, C2, DC2), !.

developper_expression(Tbox, some(R, C), some(R, DC)) :-
    developper_expression(Tbox, C, DC), !.

developper_expression(Tbox, all(R, C), all(R, DC)) :-
    developper_expression(Tbox, C, DC), !.

developper_expression(Tbox, not(C), not(DC)) :-
    developper_expression(Tbox, C, DC), !.

developper_expression(Tbox, C, DC) :-
    cnamena(C),
    get_equiv(Tbox, C, E),
    developper_expression(Tbox, E, DC), !.

developper_expression(_, C, C).
```

### 1.3.3 get_equiv

get_equiv sert à renvoyer l'expression équivalente d'un identificateur d'un concept non atomique depuis la liste Tbox.

```
get_equiv([(C, E) | _], C, E) :- !.
get_equiv([(_, _) | Tbox], C, E) :- get_equiv(Tbox, C, E).
```

## 1.4 traitement_Abox

Nous avons aussi besoin de traiter la Abox, c'est-à-dire, remplacer les identificateurs de concepts par leur expression équivalente dans la Tbox traitée, et vérifier la correction syntaxique et sémantique des instances, rôles et concepts.

Le prédicat traitement_Abox traite les deux parties Abi et Abr, en invoquant respectivement les prédicats traitement_Abi et traitement_Abr.

```
traitement_Abox(_, [], []).
traitement_Abox(Tbox, Abi , Abr, NAbi , NAbr) :-
    traitement_Abi(Tbox, Abi, NAbi),
    traitement_Abr(Tbox, Abr, NAbr), !.
```

### 1.4.1 traitement_Abi

Le traitement de Abi se résume à :

- Vérifier la correction syntaxique et sémantique d'une expression en utilisant les prédicats iname, cnamena et cnamea.
- Remplacer les identificateurs des concepts non atomiques par leur expression équivalente en concepts atomiques extraite depuis la Tbox traitée.

```
traitement_Abi(_, [], []).
traitement_Abi(Tbox, [(I, C) | Abi], [(I, C) | NAbi]) :-
    iname(I),
    cnamea(C),
    traitement_Abi(Tbox, Abi, NAbi), !.

traitement_Abi(Tbox, [(I, C) | Abi], [(I, E) | NAbi]) :-
    iname(I),
    cnamena(C),
    get_equiv(Tbox, C, E),
    traitement_Abi(Tbox, Abi, NAbi), !.

traitement_Abi(Tbox, [(I, C) | Abi], [(I, E) | NAbi]) :-
    iname(I),
    developper_expression(Tbox, C, E),
    traitement_Abi(Tbox, Abi, NAbi).
```

### 1.4.2 traitement_Abr

Le traitement d'Abr consiste en :

- La vérification de la correction syntaxique et sémantique d'une expression en utilisant les prédicats iname et rname.

```
traitement_Abr(_, [], []).
traitement_Abr(Tbox, [(I1, I2, R) | Abr], [(I1, I2, R) | NAbr]) :-
    iname(I1),
    iname(I2),
    rname(R),
    traitement_Abr(Tbox, Abr, NAbr).
```

### 1.5 premiere_etape

On codera l'étape préliminaire de vérification et de mise en forme de la Tbox et de la Abox avec le prédicat premiere_etape, qui :

- Génère la Tbox.
- Vérifie s'il n'y a pas d'autoréférence.
- Si c'est le cas, arrête le programme et affiche un message d'erreur.
- Sinon, traite la Tbox puis la Abox.

```
premiere_etape(Tbox, Abi, Abr) :-
    generateur_Tbox(T),
    not_autoref_Tbox(T),
    traitement_Tbox(T, Tbox),
    generateur_Abox(OAbi, OAbr),
    traitement_Abox(Tbox, OAbi, OAbr, Abi, Abr),
    welcome.
```

Pour tester s'il existe un cas d'auto-ref dans une Tbox, on utilisera le prédicat ci-dessous, qui parcourt toute la Tbox et vérifie chaque concept avec le prédicat autoref.

```
not_autoref_Tbox([]).
not_autoref_Tbox([(C, E) | _]) :-
    autoref(C, E),
    nl, write('Attention ! Auto-reference detectee pour le concept : '),
    write(C), nl, nl, halt, !.
not_autoref_Tbox([(_, _) | Tbox]) :- not_autoref_Tbox(Tbox).
```

Pour générer la Tbox et Abox on utilisera respectivement les prédicat generateur_Tbox et generateur_Abox, qui se basent sur set_of.

```
generateur_Tbox(Tbox) :- setof((X, Y), equiv(X, Y), Tbox).

generateur_Abox(Abi, Abr) :-
    setof( (X, Y),   inst(X, Y), Abi),
    setof((X, Y, R), instR(X, Y, R), Abr).
```

# 2 Saisie de la proposition à démontrer

## 2.1 acquisition_prop_type1

On récupère les inputs de l'utilisateur pour l'instance et le concept, on vérifie si le concept est syntaxiquement correct, puis on le développe pour avoir une forme normale négative simplifiée, et enfin, on l'ajoute à Abi.

```
acquisition_prop_type1(Abi, [(I, DCNNF) | Abi], Tbox) :-
    nl, write('Veuillez saisir l instance I'),
    nl, nl, read(I),
    nl, write('ainsi que le concept C'),
    nl, nl, read(C),
    nl,
    concept(C),
    developper_expression(Tbox, C, DC),
    nnf(not(DC), DCNNF), !.

acquisition_prop_type1(Abi, NAbi, Tbox) :-
   nl, write('Attention ! le concept que vous avez saisi n existe pas
            ou n est pas syntaxiquement correct.'), nl,
   saisie_et_traitement_prop_a_demontrer(Abi, NAbi, Tbox).
```

## 2.2 acquisition_prop_type2

On récupère les inputs de l'utilisateur pour les deux concepts, ensuite on vérifie si les concepts sont syntaxiquement corrects, afin de les développer en forme normale négative simplifiée, puis on l'ajoute à Abi.

```
acquisition_prop_type2(Abi, [(I, DCNNF) | Abi], Tbox) :-
    genere(I),
    nl, write('L instance generee est : '), write(I),
    nl, nl, write('Veuillez saisir les deux concepts'), nl, nl,
    write(' C1 : '), nl, read(C1),
    nl, write(' C2 : '), nl, read(C2),
    nl,
    concept(C1),
    concept(C2),
    developper_expression(Tbox, and(C1, C2), DC),
    nnf(DC, DCNNF), !.

acquisition_prop_type2(Abi, NAbi, Tbox) :-
    nl, write('Attention ! le concept que vous avez saisi
        n existe pas ou n est pas syntaxiquement correct.'), nl,
    saisie_et_traitement_prop_a_demontrer(Abi, NAbi, Tbox).
```

## 2.3 deuxieme_etape

Ce prédicat sert à récupérer la formule à démontrer, on prend l'initiative de vérifier que le concept saisi est bien formulé, dans le cas contraire, on redonne la main à l'utilisateur avec un message d'erreur.

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

# 3 Démonstration de la proposition

Avant de lancer notre démonstrateur, on aura d'abord besoin de fractionner Abi afin de nous faciliter la recherche des expressions à évaluer suivant un ordre précis ($\exists \rightarrow \sqcap \rightarrow \forall \rightarrow \sqcup$).

Ensuite, on lance la résolution sur le nœud racine, auquel on applique l'une des règles définies, cette dernière crée un nouveau nœud avec une mise à jour des listes.

On testera le nœud créer d'une telle sorte:

- Si il est en clash, on termine avec la branche courante et on passe aux autres branches, dans le cas où toutes les branches sont traitées, on termine notre démonstration avec succès, c'est-à-dire on a pu démontrer notre proposition.
- Sinon, on essaie d'appliquer une nouvelle règle pour générer un nouveau nœud qu'on traitera de la même manière que le courant. (on fera appel à résolution avec le nœud créé afin de respecter l'ordre d'application des règles défini).
- S'il ne reste plus de règles à appliquer et qu'y a pas de clash (nœud ouvert) alors la résolution se termine avec comme résultat la non démonstration de la formule.

```
troisieme_etape(Abi, Abr) :-
    tri_Abox(Abi, Lie, Lpt, Li, Lu, Ls),
    resolution(Lie, Lpt, Li, Lu, Ls, Abr),
    nl, write(' >>>> On a demontre la proposition initiale.'), !.

troisieme_etape(_, _) :-
   nl, write(' >>>> On a pas pu demontre la proposition initiale.').
```

Pour y parvenir on aura besoin d'une panoplie de prédicats, qui sont :

## 3.1 tri_Abox

Tri Abox prend Abi en paramètre et le fractionne en plusieurs sous listes (Lie, Lpt, Li, Lu, Ls) en fonction de la forme de ses expressions.

```
tri_Abox([], [], [], [], [], []).
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), !.
tri_Abox([(I, C) | Abi], Lie, Lpt, Li, Lu, [(I, C) | Ls]) :-
            tri_Abox(Abi, Lie, Lpt, Li, Lu, Ls), !.
```

## 3.2 resolution

La résolution démarre en appliquant la première règle $\exists$ sur le noeud racine, qui déclenchera les différents enchaînements des règles.

```
resolution(Lie, Lpt, Li, Lu, Ls, Abr) :-
    complete_some(Lie, Lpt, Li, Lu, Ls, Abr).
```

### 3.3 evolue

Le prédicat evolue prend un nouveau concept et l'insert dans la liste adéquate.
On vérifie à chaque fois que le concept n'est pas dans la liste (pour éviter les doublons).

```
evolue((I, some(R, C)), Lie, Lpt, Li, Lu, Ls, [(I, some(R, C)) | Lie], Lpt, Li, Lu, Ls) :-
                                                    not(member((I, some(R, C)), Lie)), !.
evolue((_, some(_, _)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu, Ls) :- !.

evolue((I, all(R, C)), Lie, Lpt, Li, Lu, Ls, Lie, [(I, all(R, C)) | Lpt], Li, Lu, Ls) :-
                                                    not(member((I, all(R, C)), Lpt)), !.
evolue((_, all(_, _)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu, Ls) :- !.

evolue((I, and(C1, C2)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, [(I, and(C1, C2)) | Li], Lu, Ls) :-
                                                    not(member((I, and(C1, C2)), Li)), !.
evolue((_, and(_, _)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu, Ls) :- !.

evolue((I, or(C1, C2)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, [(I, or(C1, C2)) | Lu], Ls) :-
                                                    not(member((I, or(C1, C2)), Lu)), !.
evolue((_, or(_, _)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu, Ls) :- !.

evolue((I, C), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu, [(I, C) | Ls]) :-
                                                not(member((I, C), Ls)), !.
evolue(_, Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu, Ls).
```

### 3.4 clash

Pour vérifier l'existence d'un clash ou non, on utilise le prédicat clash qui prend la liste Ls, et cherche s'il existe un (I : C) et (I : ¬C) dedans. On utilise nnf pour réduire le ¬(¬C) en C.

```
non_clash([]).
non_clash([(I, C) | Ls]) :- nnf(not(C), NNFNC), not(member((I, NNFNC), Ls)), non_clash(Ls).

clash(Ls) :-
    not(non_clash(Ls)).
```

### 3.5 complete_some

Le prédicat complete_some applique la règle ∃, pour cela, on extrait une clause de la liste contenant les instances d'existence (Lie) de la forme (I : ∃ R. C), puis crée une clause (B : C) avec B un nom d'instance généré qu'on ajoutera à sa liste correspondante grâce au prédicat evolue, et aussi une clause ((I, B) : R) qu'on ajoutera à Abr.

Ainsi on a créé un nouveau nœud dans lequel on testera le clash pour déterminer la prochaine étape.
Si la liste Lie est vide, dans ce cas-là on essayera d'appliquer la règle ⊓ , en faisant appel au prédicat transformation_and.

```
complete_some_rec([], Lpt, Li, Lu, Ls, Abr, _) :-
    transformation_and([], Lpt, Li, Lu, Ls, Abr).

complete_some_rec([(I, some(R, C)) | Lie], Lpt, Li, Lu, Ls, Abr, B) :-
    evolue((B, C), Lie, Lpt, Li, Lu, Ls, NLie, NLpt, NLi, NLu, NLs),
    str(some(R, C), S),
    write(' >>> Resolution de '), write(I), write(' : '), write(S), nl,
    affiche_evolution_Abox([(I, some(R, C)) | Lie], Lpt, Li, Lu, Ls, Abr,
                    NLie, NLpt, NLi, NLu, NLs,  [(I, B, R) | Abr]),
    clash(NLs), display_clash, !.
```

```
complete_some_rec([(I, some(R, C)) | Lie], Lpt, Li, Lu, Ls, Abr, B) :-
    evolue((B, C), Lie, Lpt, Li, Lu, Ls, NLie, NLpt, NLi, NLu, NLs),
    resolution(NLie, NLpt, NLi, NLu, NLs, [(I, B, R) | Abr]).

complete_some(Lie, Lpt, Li, Lu, Ls, Abr) :-
    genere(B),
    complete_some_rec(Lie, Lpt, Li, Lu, Ls, Abr, B).
```

### 3.6 transformation_and

Le prédicat transformation_and applique la règle $\sqcap$, pour cela, on extrait une clause (I : C1 $\sqcap$ C2) depuis Li, afin de créer deux nouvelles clauses qui sont (I : C1) et (I : C2) qu'on ajoutera à leur liste respective avec le prédicat evolue.

On se retrouve avec un nouveau nœud auquel on testera le clash.
Si la liste Li est vide, on appliquera la règle $\forall$, en faisant appel au prédicat deduction_all.

```
transformation_and(Lie, Lpt, [], Lu, Ls, Abr) :-
    deduction_all(Lie, Lpt, [], Lu, Ls, Abr).

transformation_and(Lie, Lpt, [(I, and(C1, C2)) | Li], Lu, Ls, Abr) :-
    evolue((I, C1), Lie, Lpt, Li, Lu, Ls, TLie, TLpt, TLi, TLu, TLs),
    evolue((I, C2), TLie, TLpt, TLi, TLu, TLs, NLie, NLpt, NLi, NLu, NLs),
    str(and(C1, C2), S),
    write(' >>> Resolution de '), write(I), write(' : '), write(S), nl,
    affiche_evolution_Abox(Lie, Lpt, [(I, and(C1, C2)) | Li], Lu, Ls, Abr,
                           NLie, NLpt, NLi, NLu, NLs, Abr),
    clash(NLs), display_clash, !.

transformation_and(Lie, Lpt, [(I, and(C1, C2)) | Li], Lu, Ls, Abr) :-
    evolue((I, C1), Lie, Lpt, Li, Lu, Ls, TLie, TLpt, TLi, TLu, TLs),
    evolue((I, C2), TLie, TLpt, TLi, TLu, TLs, NLie, NLpt, NLi, NLu, NLs),
    resolution(NLie, NLpt, NLi, NLu, NLs, Abr).
```

### 3.7 deduction_all

Ce prédicat applique la règle $\forall$, en extrayant une clause (I: $\forall$ R.C) depuis Lpt, et ajoutant aux listes d'Abi autant de clauses ($X_i$ : C) que de clauses ( (I, $X_i$) : R) apparaissant dans Abr, cet ajout se fait avec le prédicat inst_all.

On testera le clash du nouveau nœud obtenu pour déterminer la suite de la démonstration.

Comme précédemment, si Lpt ne contient aucune clause on passera à la règle $\sqcup$.

```
deduction_all(Lie, [], Li, Lu, Ls, Abr) :-
    transformation_or(Lie, [], Li, Lu, Ls, Abr).

deduction_all(Lie, [(I, all(R, C)) | Lpt], Li, Lu, Ls, Abr) :-
    inst_all(I, C, R, Abr, Lie, Lpt, Li, Lu, Ls, NLie, NLpt, NLi, NLu, NLs),
    str(all(R, C), S),
    write(' >>> Resolution de '), write(I), write(' : '), write(S), nl,
    affiche_evolution_Abox(Lie, [(I, all(R, C)) | Lpt], Li, Lu, Ls, Abr,
                                        NLie, NLpt, NLi, NLu, NLs, Abr),
    clash(NLs), display_clash, !.

deduction_all(Lie, [(I, all(R, C)) | Lpt], Li, Lu, Ls, Abr) :-
    inst_all(I, C, R, Abr, Lie, Lpt, Li, Lu, Ls, NLie, NLpt, NLi, NLu, NLs),
    resolution(NLie, NLpt, NLi, NLu, NLs, Abr).
```

Dans le cas d'un ∀, avant de supprimer la clause (I: ∀ R.C), on doit d'abord traiter toutes les expressions ((I, $X_i$) : R) dans Abr et créer les clauses ($X_i$ : C), par conséquent on parcourt toute la liste Abr ce qui est fait par le prédicat inst_all.

```
inst_all(_, _, _, [], Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu, Ls).

inst_all(I, C, R, [(I, B, R) | Abr], Lie, Lpt, Li, Lu, Ls,
                                NLie, NLpt, NLi, NLu, NLs) :-
    evolue((B, C), Lie, Lpt, Li, Lu, Ls, TLie, TLpt, TLi, TLu, TLs),
    inst_all(I, C, R, Abr, TLie, TLpt, TLi, TLu, TLs,
                           NLie, NLpt, NLi, NLu, NLs), !.

inst_all(I, C, R, [(_, _, _) | Abr], Lie, Lpt, Li, Lu, Ls,
                                NLie, NLpt, NLi, NLu, NLs) :-
    inst_all(I, C, R, Abr, Lie, Lpt, Li, Lu, Ls, NLie, NLpt, NLi, NLu, NLs).
```

### 3.8 transformation_or

Ce prédicat applique la règle ⊔, en extrayant une clause (I : C1 ⊔ C2) depuis Lu, et crée deux nouveaux nœuds contenant respectivement (I : C1) et (I : C2) qui sont ajoutés à leurs listes correspondantes.

Si les deux sous-noeuds créés clashent cela implique que la branche du noeud père est traitée avec succès.

Sinon on continue la résolution sur les nœuds qui clashent pas.

```
transformation_or(_, _, _, _, Ls, _) :- clash(Ls), !.

transformation_or(Lie, Lpt, Li, [(I, or(C1, C2)) | Lu], Ls, Abr) :-
    evolue((I, C1), Lie, Lpt, Li, Lu, Ls, NLie1, NLpt1, NLi1, NLu1, NLs1),
    str(or(C1, C2), S),
    write(' >>> Resolution du cote gauche de '), write(I), write(' : '), write(S), nl,
    affiche_evolution_Abox(Lie, Lpt, Li, [(I, or(C1, C2)) | Lu], Ls, Abr,
                                    NLie1, NLpt1, NLi1, NLu1, NLs1, Abr),
    clash(NLs1),
    display_clash,

    evolue((I, C2), Lie, Lpt, Li, Lu, Ls, NLie2, NLpt2, NLi2, NLu2, NLs2),
    write(' >>> Resolution du cote droit de '), write(I), write(' : '), write(S), nl,
    affiche_evolution_Abox(Lie, Lpt, Li, [(I, or(C1, C2)) | Lu], Ls, Abr,
                                    NLie2, NLpt2, NLi2, NLu2, NLs2, Abr),
    clash(NLs2),
    display_clash, !.

transformation_or(Lie, Lpt, Li, [(I, or(C1, C2)) | Lu], Ls, Abr) :-
    evolue((I, C1), Lie, Lpt, Li, Lu, Ls, NLie1, NLpt1, NLi1, NLu1, NLs1),
    resolution(NLie1, NLpt1, NLi1, NLu1, NLs1, Abr),

    evolue((I, C2), Lie, Lpt, Li, Lu, Ls, NLie2, NLpt2, NLi2, NLu2, NLs2),
    resolution(NLie2, NLpt2, NLi2, NLu2, NLs2, Abr).
```

### 3.9 affiche_evolution_Abox

Afin d'avoir une belle interface d'utilisation de notre démonstrateur, on a créé différents prédicats (**str** pour la représentation textuelle d'un concept, **affiche_list** pour afficher les listes d'Abi et **affiche_abr** pour l'affichage de Abr).

L'affichage de la Abox se réduit à des appels à ces derniers.

```
str(and(C1, C2), S) :-
    str(C1, SC1), str(C2, SC2),
    string_concat('(', SC1, T0),
    string_concat(T0, ' ⊓ ', T1),
    string_concat(T1, SC2, T2),
    string_concat(T2, ')', S), !.

str(or(C1, C2), S) :-
    str(C1, SC1), str(C2, SC2),
    string_concat('(', SC1, T0),
    string_concat(T0, ' ⊔ ', T1),
    string_concat(T1, SC2, T2),
    string_concat(T2, ')', S), !.

str(some(R, C), S) :-
    str(C, SC),
    string_concat('(', '∃ ', T0),
    string_concat(T0, R, T1),
    string_concat(T1, '.', T2),
    string_concat(T2, SC, T3),
    string_concat(T3, ')', S), !.

str(all(R, C), S) :-
    str(C, SC),
    string_concat('(', '∀ ', T0),
    string_concat(T0, R, T1),
    string_concat(T1, '.', T2),
    string_concat(T2, SC, T3),
    string_concat(T3, ')', S), !.

str(not(C), S) :-
    str(C, SC), string_concat('¬', SC, S), !.

str(anything, 'T') :- !.
str(nothing, '⊥') :- !.
str(X, X).

% affiche_list pour afficher les listes d'Abi

affiche_list_rec([]).
affiche_list_rec([(I, X) | L]) :- str(X, SX), write('('), write(I),
                                  write(' : '), write(SX), write('), '),
                                  affiche_list_rec(L).
affiche_list(L) :- write('['), affiche_list_rec(L), write(']').

% affiche_abr pour l'affichage de Abr

affiche_abr_rec([]).
affiche_abr_rec([(I1, I2, R) | L]) :- write('(<'), write(I1), write(','), write(I2),
                                      write('>'), write(' : '), write(R),
                                      write('), '), affiche_abr_rec(L).
affiche_abr(L) :- write('['), affiche_abr_rec(L), write(']').

affiche_Abox(Lie, Lpt, Li, Lu, Ls, Abr) :-
    write('    Lie = '), affiche_list(Lie), nl,
    write('    Lpt = '), affiche_list(Lpt), nl,
```

```
    write('    Li = '), affiche_list(Li), nl,
    write('    Lu = '), affiche_list(Lu), nl,
    write('    Ls = '), affiche_list(Ls), nl,
    write('    Abr = '), affiche_abr(Abr), nl.

% l'affichage de la Abox se réduit à des appels à affiche_list et affiche_abr

affiche_evolution_Abox(Lie, Lpt, Li, Lu, Ls, Abr, NLie, NLpt, NLi, NLu, NLs, NAbr) :-
    nl,
    write('    -------- Avant --------'), nl,
    affiche_Abox(Lie, Lpt, Li, Lu, Ls, Abr),
    nl,
    write('    -------- Apres --------'), nl,
    affiche_Abox(NLie, NLpt, NLi, NLu, NLs, NAbr),
    nl, nl, nl.
```