Projet LRC : Ecriture en Prolog d'un démonstrateur basé sur l'algorithme des tableaux pour la logique de description $\boldsymbol{ALC}$

Roxane CELLIER —— Lou MOULIN-ROUSSEL Année 2022-2023

Table des matières

1 Préambule

Dans ce rapport de projet, nous présentons et expliquons le code que nous avons écrit pour répondre au sujet. Pour plus de lisibilité, ce rapport est composé de trois parties, correspondant aux parties indiquées dans le sujet.
Le code est composé de trois fichiers, un pour chaque partie, ainsi que d'un fichier contenant les prédicats indiqués en annexe du sujet :

- `partie1.pl`
- `partie2.pl`
- `partie3.pl`
- `preliminaire.pl`

2 Partie I - Etape préliminaire de vérification et de mise en forme de la Tbox et de la Abox

2.1 Prédicat `concept`

Le prédicat `concept` permet de vérifier la correction syntaxique et la correction sémantique de la Tbox et de la Abox en entrées, ainsi que celles des expressions que l'utilisateur entrera au clavier lorsqu'il sera sollicité pour fournir une proposition à démontrer. En voici le code en *Prolog* :

```
concept(C) :- cnamea(C), !.
concept(C) :- cnamena(C), !.
concept(not(C)) :- concept(C), !.
concept(or(C1,C2)) :- concept(C1), concept(C2), !.
concept(and(C1,C2)) :- concept(C1), concept(C2), !.
concept(some(R,C)) :-   role(R), concept(C), !.
concept(all(R,C)) :-    role(R), concept(C), !.

instance(I) :-  iname(I), !.
role(R) :-      rname(R), !.
```

Les deux premières lignes permettent de vérifier les cas de base : elles vérifient s'il s'agit de concepts (atomiques ou non) définis au préalable par les lignes de code données dans le sujet. Ensuite, on utilise la récursivité pour vérifier que les concepts considérés sont construits à l'aide des opérateurs `not, or, and, some, all`

et des concepts de base.

2.2 Prédicat `autoref`

Nous allons écrire le prédicat `remplace(concept, definition)` qui nous permettra de remplacer un concept par une expression équivalente, sa définition, où ne figurent plus que des identificateurs de concepts atomiques. Ce prédicat s'écrit en *Prolog* :

```
remplace(CA, CA) :- cnamea(CA), !.
remplace(CNA, DCA) :- equiv(CNA, D), remplace(D, DCA) !.
remplace(not(CNA), not(CA)) :- remplace(CNA, CA), !.
remplace(or(CNA1, CNA2), or(CA1, CA2)) :- remplace(CNA1, CA1),
                                          remplace(CNA2, CA2), !.
remplace(and(CNA1, CNA2), and(CA1, CA2)) :- remplace(CNA1, CA1),
                                            remplace(CNA2, CA2), !.
remplace(some(R, CNA), some(R, CA)) :- remplace(CNA, CA), !.
remplace(all(R, CNA), all(R, CA)) :- remplace(CNA, CA), !.
```

`remplace(concept, definition)` est vrai si et seulement si `concept` est équivalent à `definition` qui est une expression ne comprenant que des concepts atomiques. Autrement dit, on "développe" `concept` pour trouver sa définition construite seulement avec des concepts atomiques.

Le prédicat `autoref` permet de vérifier qu'il n'y a pas d'autoréférencement dans les définitions des concepts. Il est écrit de manière récursive pour pouvoir l'appliquer avec des concepts non atomiques.

```
autoref(C, C).
autoref(C, equiv(C,D)) :- remplace(D,DA), autoref(C,DA), !.
autoref(C, and(A,B)) :- autoref(C,A), autoref(C,B), !.
autoref(C, or(A,B)) :- autoref(C,A), autoref(C,B), !.
autoref(C, some(R,B)) :- autoref(C,B), !.
autoref(C, all(R,B)) :- autoref(C,B), !.
```

On a utilisé `remplace` afin de transformer la définition D du concept C par une expression composée de concepts atomiques. Ainsi, cela nous permet de définir `autoref` de manière récursive en ne considérant que les cas où un concept est atomique ou défini avec les opérateurs `and, or, some` et `all`.

2.3 Prédicat `traitement-Tbox`

Ensuite nous n'avons plus qu'à mettre cette expression sous forme normale négative pour faire le traitement de la Tbox demandé.

```
traitement-Tbox([], []).
traitement-Tbox([(C,D) | Tbox], [(NC,ND) | L]) :- concept(C), concept(D),
                                                   not(autoref(C,D)),
                                                   remplace(C,CA), remplace(D,DA),
                                                   nnf(CA, NC), nnf(DA,ND),
                                                   traitement-Tbox(Tbox, L), !.
```

Le prédicat `traitement_Tbox(Tbox,FN)` est vrai si et seulement si la liste `FN` correspond aux concepts de la liste `Tbox` mis sous forme normale négative. Il est défini de manière récursive sur les éléments de la liste Tbox donnée en entrée. On utilise le prédicat `remplace` pour trouver les expressions ne comprenant que des concepts atomiques puis on la met sous forme normale négative avec le prédicat `nnf` donné dans l'énoncé.

2.4 Prédicat `traitement-Abox`

Le prédicat `traitement_Abox` est construit sur le même principe que le prédicat `traitement_Tbox(Tbox,FN)`. Mais la liste donnée en entrée du prédicat représente des assertions d'instance et de rôles plutôt que des doublets de concepts. Il faut donc adapter le code et pour cela, nous allons écrire deux prédicats récursifs à l'aide du prédicat `remplace`, l'un pour le traitement des assertions d'instance et l'autre pour les assertions de rôle :

```
traitement-AboxI([], []).
traitement-AboxI([(I,C) | Abox], [(I,NC) | L] ) :-
                                    instance(I), concept(C),
                                    remplace(C,CA),
                                    nnf(CA, NC),
                                    traitement-AboxI(Abox), !.

traitement-AboxR([]).
traitement-AboxR([(I1,I2,R) | Abox]) :- instance(I1), instance(I2), role(R),
                                        traitement-AboxR(Abox), !.
```

3 Partie II - Saisie de la proposition à démontrer

3.1 Proposition de type $I : C$

Le prédicat `acquisition_prop_type1(Abi,Abi1,Tbox)` va nous permettre de demander à l'utilisateur d'entrer une proposition de type 1 ($I : C$) et de l'acquérir. On utilise le prédicat `remplace` afin de s'assurer que le concept ajouté est bien composé d'éléments atomiques, puis `nnf` pour obtenir le concept sous forme normale négative et on ajoute la négation de ce concept à liste des concepts.

```
acquisition_prop_type1(Abi,Abi1,Tbox) :- nl,
        write('Entrez le nom de l"instance que vous souhaitez tester :'),
        nl, read(I), instance(I), nl,
        write('Entrez le concept associe que vous souhaitez tester :'),
        nl, read(C), concept(C),
        remplace(C, CA), nnf(not(CA), NCA),
        concat(Abi, [(I,NCA)], Abi1), !.
```

3.2 Proposition de type $C1 \sqcap C2 \sqsubseteq \bot$

Le prédicat `acquisition_prop_type2(Abi,Abi1,Tbox)` fonctionne selon le même principe. Mais pour pouvoir ajouter dans la ABox le concept $C1 \sqcap C2$, il faut générer un nom d'instance aléatoire. C'est pour cela que nous utilisons le prédicat `genere`, rappelé en annexe du sujet, dans le code ci-dessous :

```
acquisition_prop_type2(Abi,Abi1,Tbox) :- nl,
        write('Entrez premier concept que vous souhaitez tester :'),
        nl, read(C1), concept(C1), remplace(C1, CA1),nl,
        write('Entrez le concept associe que vous souhaitez tester :'),
        nl, read(C2), concept(C2), remplace(C2, CA2),
        nnf(and(CA1, CA2), NCA), genere(Nom),
        concat(Abi, [(Nom, NCA)], Abi1), !.
```

4 Partie III - Démonstration de la proposition

4.1 Prédicats préliminaires

Dans cette sous-partie, nous allons définir des prédicats utiles pour l'implémentation.
Le prédicat `evolue` permet d'ajouter de nouvelles assertions de contraintes aux

listes Lie, Lpt, Li, Lu et Ls. En effet chaque liste représente un type de relation, que nous allons mettre à jour selon l'étape où nous sommes.

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

evolue((I,some(R,C)), Lie, Lpt, Li, Lu, Ls, Lie1, Lpt, Li, Lu, Ls) :-
                                concat([ (I,some(R,C)) ], Lie, Lie1), !.
evolue((I,all(R,C)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt1, Li, Lu, Ls) :-
                                concat([ (I,all(R,C)) ], Lpt, Lpt1), !.
evolue((I,and(C1,C2)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li1, Lu, Ls) :-
                                concat([ (I,and(C1,C2)) ], Li, Li1), !.
evolue((I,or(C1,C2)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu1, Ls) :-
                                concat([ (I,or(C1,C2)) ], Lu, Lu1), !.
evolue((I,C), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu, Ls1) :-
                                concat([ (I,C) ], Ls, Ls1), !.
evolue((I,not(C)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu, Ls1) :-
                                concat([ (I,not(C)) ], Ls, Ls1), !.
```

Le code ci-dessus modifie les listes d'assertions en fonction du type de la nouvelle assertion.

4.2 Prédicat `resolution`

4.2.1 Prédicat `complete-some`

On applique la règle de résolution pour les assertions de concept de la forme (I,some(R,C)).

```
complete_some([ (I1,some(R,C)) | Lie], Lpt, Li, Lu, Ls, Abr) :-
                                                genere(I2),
                % generation d'un nom d'instance
                % modifie en consequence les listes concernees

                        evolue((I2,C), Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1,
                    Li1, Lu1, Ls1),
                % affiche l'evolution a cette etape

                        affiche_evolution_Abox(Ls, [ (I1,some(R,C)) | Lie],
                        Lpt, Li, Lu, Abr, Ls1, Lie1, Lpt1, Li1, Lu1,
                        [ (I1, I2, R) | Abr]),

                % retour a la resolution avec les nouvelles listes
```

```
        resolution(Lie1, Lpt1, Li1, Lu1, Ls1,
        [(I1, I2, R) | Abr]), !.
```

Le prédicat `complete_some` est utilisé dans le prédicat `resolution` (qui lui est utilisé dans `complete_some`), cela entraîne donc une récursion dans le prédicat `resolution`. Ce prédicat et le prédicat `affiche_evolution_Abox`, tous deux utilisés ici, sont définis plus loin dans le rapport.

4.2.2 Prédicat `transformation-and`

On applique la règle de résolution pour les assertions de concept de la forme (I,and(C1,C2)). Ce prédicat est écrit selon le même principe que `complete_some`.

```
transformation_and(Lie, Lpt, [(I,and(C1,C2)) | Li], Lu, Ls, Abr) :-

        % modifie en consequence les listes concernees
                evolue([(I,C1), (I,C2)], Lie, Lpt, Li, Lu, Ls1, Lie1,
                Lpt1, Li1, Lu1, Ls1),

        % affiche l'evolution a cette etape
                affiche_evolution_Abox(Ls, Lie, Lpt, [(I,and(C1,C2)) | Li],
                Lu, Abr, Ls1, Lie1, Lpt1, Li1, Lu1, Abr),

        % retour a la resolution avec les nouvelles listes
                resolution(Lie1, Lpt1, Li1, Lu1, Ls1, Abr), !.
```

4.2.3 Prédicat `deduction-all`

On applique la règle de résolution pour les assertions de concept de la forme (I,all(R,C)) selon le même principe que pour les prédicats précédents.

```
deduction_all(Lie, [(I1,all(R,C)) | Lpt], Li, Lu, Ls, Abr) :-
        member((I1,I2,R), Abr),
    % on regarde si une relation de role existe dans la A-Box
    % modifie en consequence les listes concernees
        evolue((I2,C), Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1, Li1,
            Lu1, Ls1),
    % affiche l'evolution a cette etape
        affiche_evolution_Abox(Ls, Lie, [(I1,all(R,C)) | Lpt],
            Li, Lu, Abr, Ls1, Lie1, Lpt1, Li1, Lu1, Abr),
    % retour a la resolution avec les nouvelles listes
        resolution(Lie1, Lpt, Li1, Lu1, Ls1, Abr), !.
```

4.2.4 Prédicat `transformation-or`

On applique la règle de résolution pour les assertions de concept de la forme (I,or(C1,C2)).

```
transformation_or(Lie, Lpt, Li, [(I,or(C1,C2)) | Lu], Ls, Abr) :-
    % modifie en consequence les listes concernees (deux noeuds)
        evolue((I,C1), Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1, Li1, Lu1, Ls1),
        evolue((I,C2), Lie, Lpt, Li, Lu, Ls, Lie2, Lpt2, Li2, Lu2, Ls2),
        % affiche l'evolution a cette etape
        affiche_evolution_Abox(Ls, Lie, Lpt, Li,
                [(I,or(C1,C2)) | Lu], Abr, Ls1, Lie1, Lpt1, Li1, Lu1, Abr),
        affiche_evolution_Abox(Ls, Lie, Lpt, Li,
                [(I,or(C1,C2)) | Lu], Abr, Ls2, Lie2, Lpt2, Li2, Lu2, Abr),
        % retour a la resolution avec les nouvelles listes (deux noeuds)
        resolution(Lie1, Lpt1, Li1, Lu1, Ls1, Abr),
        resolution(Lie2, Lpt2, Li2, Lu2, Ls2, Abr), !.
```

4.2.5 Prédicat `resolution`

On commence par écrire le prédicat `pas_clash` qui vérifie si une assertion de type (I,C) est contredite dans la liste des assertions de type (I,C). par

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

Remarque : Ce prédicat semble comporter une erreur d'après la trace de Prolog. *Nous n'avons cependant pas réussi à le faire fonctionner malgré nos tentatives de modification du code.*

Nous nous en servons pour écrire le prédicat `resolution` :

```
resolution(Lie, Lpt, Li, Lu, Ls, Abr) :- pas_clash(Ls),
                complete_some(Lie, Lpt, Li, Lu, Ls, Abr), !.
resolution([], Lpt, Li, Lu, Ls, Abr) :- pas_clash(Ls),
                transformation_and([], Lpt, Li, Lu, Ls, Abr), !.
resolution([], Lpt, [], Lu, Ls, Abr) :- pas_clash(Ls),
                deduction_all([], Lpt, [], Lu, Ls, Abr), !.
resolution([], [], [], Lu, Ls, Abr):- pas_clash(Ls),
                transformation_or([], [], [], Lu, Ls, Abr), !.
resolution([], [], [], [], Ls, Abr):- pas_clash(Ls), !.
```

Le prédicat `resolution` nous permet de mettre en oeuvre les prédicats écrits précédemment. Ainsi, il nous permet d'appliquer les règles de résolution. Dans chaque cas, il faut vérifier qu'il n'y a pas de clash, d'où l'importance de `pas_clash`.

4.3 Affichage

Pour écrire le prédicat `affiche_evolution_Abox`, on écrit tout d'abord le prédicat `affiche` dont le code est donné ci-dessous :

```
% recursivite d'affichage sur les listes

affiche([]) :-  nl.
affiche([Elem | L]) :- affiche(Elem), nl,
                       affiche(L), !.

% affichage des concepts

affiche(C) :- write(C), !.

affiche((I1,I2,R)) :- write('<'), write(I1), write(','), write(I2),
write('> : '), write(R), !.

affiche((I,C)) :- write(I), write(' : '), affiche(C), !.

affiche((I,not(C))) :- write(I), write(' : ¬('), affiche(C),
write(')'), !.

affiche((I,some(R,C))) :- write(I), write(' : ∃ '),
                          write(R), write('.('), affiche(C), write(')'), !.

affiche((I,all(R,C))) :- write(I), write(' : ∀ '),
                         write(R), write('.('), affiche(C), write(')'), !.

affiche((I,and(C1,C2))) :- write(I), write(' : ('),
                           affiche(C1), write(' ⊓ '), affiche(C2), write(')'), !.

affiche((I,or(C1,C2))) :- write(I), write(' : ('),
                          affiche(C1), write(' ⊔ '), affiche(C2), write(')'), !.
```

Ce prédicat nous permet d'afficher les différentes relations entre concepts, rôles et instances avec les symboles mathématiques correspondant. On l'implémente de manière récursive pour pouvoir l'appliquer à des listes. On l'utilise ensuite pour écrire `affiche_evolution_Abox`.

```
affiche_evolution_Abox(Ls1, Lie1, Lpt1, Li1, Lu1, Abr1, Ls2, Lie2, Lpt2,
        Li2, Lu2, Abr2):- write('Etat Depart :'),nl,
                          write('- Ls : '), nl, affiche(Ls1),
                          write('- Lie : '), nl, affiche(Lie1),
                          write('- Lpt : '), nl, affiche(Lpt1),
                          write('- Li : '), nl, affiche(Li1),
                          write('- Lu : '), nl, affiche(Lu1),
```

```
write('- Abr : '), nl, affiche(Abr1),
nl,
write('Etat Arrivee :'),nl,
write('- Ls : '), nl, affiche(Ls2),
write('- Lie : '), nl, affiche(Lie2),
write('- Lpt : '), nl, affiche(Lpt2),
write('- Li : '), nl, affiche(Li2),
write('- Lu : '), nl, affiche(Lu2),
write('- Abr : '), nl, affiche(Abr2),
nl, write('Fin de l etape.').
```

Le prédicat `affiche_evolution_Abox` affiche l'évolution d'un état de la Abox étendue (état de départ) vers un état suivant (état d'arrivée) à l'aide de `affiche`.

4.4 Implémentation de la démonstration de la proposition

Comme expliqué dans le sujet, il faut un prédicat `tri_Abox` afin de générer les listes des propositions Lie, Lpt, Li, Lu et Ls sur lesquelles seront appliqué le prédicat `resolution` présenté ci-dessus.

```
% Tri de la A_Box etendue en fonction de la relation consideree
tri_Abox([], 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), !.
tri_Abox([ I,C) | Abi], Lie, Lpt, Li, Lu, [ (I,C) | Ls]) :-
tri_Abox(Abi, Lie, Lpt, Li, Lu, Ls), !.
tri_Abox([(I,not(C)) | Abi], Lie, Lpt, Li, Lu, [ (I,not(C)) | Ls]) :-
                                tri_Abox(Abi, Lie, Lpt, Li, Lu, Ls), !.
```

Enfin, il n'y a plus qu'à écrire le prédicat `troisieme_etape`, comme indiqué dans le sujet, à l'aide des prédicats `tri_Abox` et `resolution`.

Puis finalement on écrit `programme` :

```
programme :- premiere_etape(Tbox,Abi,Abr),
             deuxieme_etape(Abi,Abi1,Tbox),
             troisieme_etape(Abi1,Abr).
```

Nous pouvons maintenant l'utiliser comme démonstrateur basé sur l'algorithme des tableaux pour la logique de description ALC.