

Rapport de projet

Écriture en Prolog d'un démonstrateur basé sur l'algorithme
des tableaux pour la logique de description \mathcal{ALC}

Charles Vin

Barthélémy Dang-Nhu

Table des matières

1	Introduction	2
2	Description générale et fonctionnement	2
2.1	Les fichiers	2
2.2	Utilisation du programme	2
2.2.1	Initialisation de la TBox et de la ABox	2
2.2.2	Lancement et saisie de la proposition à prouver	2
2.3	Exemple d'utilisation	2
3	Étape préliminaire de vérification et de mise en forme de la Tbox et de la Abox	2
3.1	Correction syntaxique et sémantique	2
3.2	Vérification de l'auto-référencement	3
3.3	Mise sous forme	4
4	Saisie de la proposition à montrer	4
4.1	Proposition de type 1 : $I : C$	4
4.2	Proposition de type 2 : $C1 \sqcap C2 \sqsubseteq \perp$	5
5	Démonstration de la proposition	6
5.1	Prédicat utile à l'implémentation	6
5.2	Algorithme de résolution	6

1 Introduction

2 Description générale et fonctionnement

2.1 Les fichiers

Pour une meilleure compréhension du code, nous avons séparé celui-ci en plusieurs fichiers. Chacun à sa spécialisation :

- `T-A_box.pl` : C'est ici que l'utilisateur entre la TBox et la ABox initiale.
- `run.pl` : Contient le prédicat `programme/0` qui appelle les grandes étapes nécessaires à la résolution.
- `part1.pl` : Contient les fonctions liées à la première partie décrite par le sujet.
- `part2.pl` : Contient les fonctions liées à la deuxième partie décrite par le sujet.
- `part3.pl` : Contient les fonctions liées à la troisième partie décrite par le sujet.
- `helper.pl` : Contient quelques prédicats utiles

2.2 Utilisation du programme

2.2.1 Initialisation de la TBox et de la ABox

2.2.2 Lancement et saisie de la proposition à prouver

Pour lancer le programme entrez cette commande à la racine du projet : `swipl -f run.pl`. Puis dans l'interpréteur prolog utilisez le prédicat `programme/0` pour lancer le programme.

2.3 Exemple d'utilisation

3 Étape préliminaire de vérification et de mise en forme de la Tbox et de la Abox

À ce stade les Abox et Tbox ne sont pas altérées, il est donc inutile de les mettre en argument des prédicats car on peut y accéder avec les prédicats `cnamea`, `cnamena`, `iname`, `rname`, `equiv`, `inst`, `instR`.

3.1 Correction syntaxique et sémantique

Dans cette première partie nous commençons par vérifier la correction sémantique et syntaxique des deux box. Pour ce faire nous implémentons les prédicats d'arité 1 `concept`, `instance`, `role` qui vérifient si des objets sont de ce type. Les cas de base sont les suivants :

```
concept(C) :- cnamea(C), !.  
concept(CG) :- cnamena(CG), !.  
instance(I) :- iname(I), !.  
role(R) :- rname(R), !.
```

Le prédicat `concept` nécessite de la récursivité à cause des concepts non-atomiques. Nous vérifions la grammaire avec les prédicats suivants :

```
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), !.
```

Nous utilisons ces prédicats pour définir le prédicat `definition` d'arité 2 qui vérifie si la définition d'une équivalence est juste : il faut que le premier élément soit un concept non-atomique et que le deuxième soit bien la définition d'un concept :

```
definition(CA, CG) :- cnamena(CA), concept(CG), !.
```

Grâce à ce prédicat nous pouvons vérifier la Tbox avec `verif_Tbox` qui prend en argument une TBox sous forme d'une liste de d'équivalence et qui vérifie la correction syntaxique et sémantique :

```
verif_Tbox([(CA, CG) | Q]) :-
    definition(CA, CG),
    verif_Tbox(Q).
verif_Tbox([]).
```

On fait de même avec la Abox

3.2 Vérification de l'auto-référencement

Nous voulons ensuite s'assurer qu'il n'y a pas d'auto-référencement dans la définition des concepts non-atomiques. S'il y en a, au moment de développer les concepts pour n'avoir que des concepts atomiques il y aura une boucle infinie. Nous implémentons le prédicat `pautoref(C, Def)` qui prend en argument un concept non-atomique C et la définition de concept Def et qui est vraie ssi C n'est pas présent récursivement dans Def. Le cas de base est le suivant :

```
pautoref(C, Def) :- cnamea(Def).
```

Nous pouvons ainsi construire le prédicat `verif_Autoref(L)` qui prend en argument la liste des concept non-atomiques et qui vérifie s'il n'y a pas auto-référencement dans leur définition :

```
verif_Autoref([]).
verif_Autoref([C|L]) :-
    equiv(C, Def_C),
    pautoref(C, Def_C),
    verif_Autoref(L).
```

3.3 Mise sous forme

Une fois que nous sommes sûr qu'il n'y a pas d'auto-référencement on peut développer les concepts non-atomiques. Nous implémentons le prédicat `developpe(C,D)` qui est vraie ssi D est le développement de C. Le cas de base est le suivant :

```
developpe(C, C) :- cnamea(C).
```

On peut ensuite écrire le prédicat `transforme(L1,L2)` qui prend en argument deux box L1 et L2 et qui est vraie si et seulement si L2 est la box équivalent à L1 mais dans laquelle les concepts non-atomiques sont développés puis mis sous formes normales négatives :

```
transforme([], []).
transforme([(X,C) | L], [(X,D) | M]) :-
    developpe(C, E),
    nnf(E, D),
    transforme(L, M).
```

Ce prédicat `transforme / 2` combine `traitement_Tbox` et `traitement_Abox` mentionnés par le sujet.

4 Saisie de la proposition à montrer

Dans la deuxième partie, l'utilisateur peut choisir d'entrer deux types de propositions à démontrer :

- Un proposition de type $I : C$ qui sera géré par le prédicat `acquisition_prop_type1 / 3`
- Un proposition de type $C1 \sqcap C2 \sqsubseteq \perp$ qui sera géré par le prédicat `acquisition_prop_type2 / 3`

4.1 Proposition de type 1 : $I : C$

On commence par demander à l'utilisateur d'entrer I et C par le biais du prédicat `input_prop_type1` C'est également ici que l'on vérifie si l'entrée de l'utilisateur est correct syntaxiquement avec le prédicat `instanciationC / 2`.

```
input_prop_type1(I, CG) :-
    write('Ajoutons une instance de concept à la ABox :'), nl,
    write('Elle a la forme "I : C"'), nl,
    write('Entrez I :'), nl,
    read(I), nl,
    write('Entrez C :'), nl,
    read(CG),
    (instanciationC(I, CG) -> % if
        write("Input correct")
        ; ( % else
            write('Erreur : I n\'est pas une instance déclarée ou C n\'est pas un concept'), nl,
```

```

write('Veuillez recommencer'), nl
% input_prop_type1(I, CG) % boucler ne semble pas fonctionner
)), nl.

```

Dans la suite d'`acquisition_prop_type1` on effectue quelques traitements sur $\neg C$ en remplaçant de manière récursive les identificateurs de concepts complexes par leur définition et en mettant le tout sous forme normale négative (prédicat `transforme`). On peut ensuite ajouter le tout dans la ABox avec `concat`.

```

acquisition_prop_type1(Abi, Abi1, Tbox) :-
    input_prop_type1(I, CG), % User input
    transforme([(I, not(CG))], [(I, CG_dev_nnf)]), % Développement + nnf
    concat(Abi, [(I, CG_dev_nnf)], Abi1), % Ajout de l'input de l'utilisateur dans la ABox
    write("Abi1"), write(Abi1).

```

4.2 Proposition de type 2 : $C1 \sqcap C2 \sqsubseteq \perp$

Comme précédemment, on commence par demander à l'utilisateur d'entrer $C1$ et $C2$ par le biais du prédicat `input_prop_type2 / 2`. Et on vérifie si l'entrée de l'utilisateur est correct syntaxiquement avec le prédicat `concept / 2`.

```

input_prop_type2(C1, C2) :-
    write('Ajoutons une proposition de type 2.'), nl,
    write('Entrez C1 :'), nl,
    read(C1), nl,
    write('Entrez C2 :'), nl,
    read(C2),
    (concept(and(C1, C2)) -> % if
        write("Input correct")
        ; ( % else
            write('Erreur : C1 ou C2 n\'est pas un concept déclarée'), nl,
            write('Veuillez recommencer'), nl
            % input_prop_type2(C1, C2) % boucler ne semble pas fonctionner
        )), nl.

```

Cette fois-ci il faut générer un nom de concept aléatoire `Random_CName` afin de pouvoir ajouter dans la ABox `Random_CName : C1 \sqcap C2`, c'est ce qui est fait par le biais de `genere / 1` et `transforme / 2`. Puis le tout est `concat / 3` dans la ABox.

```

acquisition_prop_type2(Abi, Abi1, Tbox) :-
    input_prop_type2(C1, C2), % User input
    genere(Random_CName),
    transforme([(Random_CName, and(C1, C2))], [(Random_CName, and(C1_dev_nnf, C2_dev_nnf))]), % Déve

```

```
concat(Abi, [(Random_CName, and(C1_dev_nnf, C2_dev_nnf))], Abi1), % Ajout de l'input de l'utilisateur
write("Abi1"), write(Abi1).
```

5 Démonstration de la proposition

Dans cette partie, nous allons démontrer par la méthode des tableaux si la Abox *Abe* construite dans la partie précédente, par l'ajout de la négation de l'entrée de l'utilisateur, mène à une contradiction.

5.1 Prédicat utile à l'implémentation

Pour faciliter l'implémentation, on commence par extraire chaque type d'assertion de la Abox pour les mettre dans 5 listes grâce au prédicat `tri_Abox / 6`

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

Le deuxième prédicat important pour l'implémentation est `evolve / 11` qui prends en première paramètre une nouvelle proposition et l'ajoute dans la liste correspondante en vérifiant qu'il n'y est pas déjà. Le cas de base est

```
evolve(A, Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu, Ls) :-
    member(A, Ls).
evolve(A, Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu, [A|Ls]).
```

Enfin, le dernier prédicat qui nous sera utile pour la suite est `non_clash / 1` qui vérifie l'absence de clash dans la liste d'assertion reçu.

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

5.2 Algorithme de résolution