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

Année : 2022/2023

SORBONNE UNIVERSITÉ

Table des matières

1 Description générale et fonctionnement

1.1 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 appel 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

1.2 Utilisation du programme

1.2.1 Initialisation de la TBox et de la ABox

Les informations de la TBox et de la ABox sont à saisir dans le fichier `T-A_Box.pl` à la racine du projet. On utilisera les prédicats suivant :

- equiv(ConceptAtomique, ConceptGénérique) pour indiquer les équivalences.
- inst(Instance, ConceptGénérique) pour indiquer les instanciations de concepts.
- instR(Instance1, Instance2, rôle) pour indiquer les instanciations de rôles.
- cnamea(ConceptAtomique) pour indiquer les identificateurs de concepts atomiques ?
- cnamena(ConceptAtomique) pour indiquer les identificateurs de concepts non atomiques.
- iname(Instance) pour indiquer les identificateurs d'instances
- rname(Rôle) pour indiquer les rôles

Le programme s'arrêtera et l'utilisateur sera averti en cas d'une erreur de grammaire ou de syntaxe mais également si la Tbox est cyclique.

1.2.2 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.

1.3 Exemple d'utilisation

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

2.1 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 vérifie 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 en vérifiant l'instanciation des concepts avec instanciationC(I, CG) et l'instanciation des relations avec instanciationR(I1, I2, R).

2.2 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 si et seulement si 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 concepts 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).
```

L'utilisation d'equiv nous permet de récupérer directement la définition de C, sans avoir à passer une box en argument. En effet les boxs n'ont juste là pas été altérées, on peut donc récupérer toutes les informations nécessaires directement avec les prédicats cnamea, cnamena, iname, rname, equiv, inst, instR.

2.3 Mise sous forme

Une fois que nous sommes sûrs 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 si et seulement si D est le développement de C. Le cas de base est le suivant :

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

Voici un exemple avec le "ou" logique :

```
developpe(or(C1,C2), or(D1,D2)) :-
        developpe(C1, D1),
        developpe(C2, D2).
```

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 équivalente 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.

3 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 $C1 \sqcap C2 \sqsubseteq \bot$ qui sera géré par le prédicat acquisition_prop_type2 / 3

3.1 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 correcte 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).
```

3.2 Proposition de type 2 : $C1 \sqcap C2 \sqsubseteq \bot$

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 correcte 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 déclaré'), 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, la liste unaire est mise en premier argument car dans le cas contraire il y aurait un parcours intégral de Abi d'après l'implémentation de concat.

```
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))]),
    concat([(Random_CName, and(C1_dev_nnf, C2_dev_nnf))], Abi, Abi1),
    write("Abi1"), write(Abi1).
```

4 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.

4.1 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([(I,C)|L], Lie, Lpt, Li, Lu, [(I,C)|Ls]) :-
    cnamea(C),
    tri_Abox(L, Lie, Lpt, Li, Lu, Ls).
tri_Abox([(I,not(C))|L], Lie, Lpt, Li, Lu, [(I,not(C))|Ls]) :-
    cnamea(C),
    tri_Abox(L, Lie, Lpt, Li, Lu, Ls).
```

Le deuxième prédicat important pour l'implémentation est evolue / 11 qui prends en premier paramètre une nouvelle proposition et l'ajoute dans la liste correspondante en vérifiant qu'il n'y est pas déjà. Voici l'exemple avec Lie :

```
evolue((I, some(R,C)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu, Ls) :-
    member((I, some(R,C)), Lie).
evolue((I, some(R,C)), Lie, Lpt, Li, Lu, Ls, [(I, some(R,C)) | Lie], Lpt, Li, Lu, Ls):-
    \+ member((I, some(R,C)), Lie).
```

Nous avons aussi besoin d'une fonction evolue_rec qui est similaire mais qui prend une liste d'instanciation en premier argument et applique evolue à tous ses éléments.

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).
```

L'appel de nnf permet de transformer un $\neg\neg$C en C et ne change pas un $\neg$C. Nous avons aussi implémenté diverses fonctions d'affichage qui peut représenter des Abox avec une écriture infixe des concepts.

4.2 Algorithme de résolution

Nous avons implémenté le prédicat resolution/6 qui prend en argument un ABox triée et qui applique la méthode des tableaux, elle renvoie vraie si et seulement si une feuille ouverte est trouvée. Le cas de base est donc le suivant :

```
resolution([], [], [], [], Ls, _):-
        non_clash(Ls).
```

Nous allons ensuite définir cette fonction récursivement en utilisant l'ordre de priorité donnée dans le sujet :

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

resolution([], Lpt, Li, Lu, Ls, Abr) :-
    non_clash(Ls),
    transformation_and([], Lpt, Li, Lu, Ls, Abr).

resolution([], Lpt, [], Lu, Ls, Abr) :-
    non_clash(Ls),
    deduction_all([], Lpt, [], Lu, Ls, Abr).

resolution([], [], [], Lu, Ls, Abr):-
        non_clash(Ls),
        transformation_and([], [], [], Lu, Ls, Abr).
```

À chaque fois nous vérifions la présence de clash avant l'appel du traitement.

4.2.1 Intersection

On souhaite traiter une instanciation $I : C_1 \sqcap C_2$.

```
transformation_and(Lie, Lpt, [(I, and(C1,C2)) | Li], Lu, Ls, Abr) :-
        write('Utilisation de la règle \u2A05 sur : '),affiche_Abi([(I, and(C1,C2))]),nl,
        evolue_rec([(I,C1),(I,C2)], Lie, Lpt, Li, Lu, Ls,
            Lie1, Lpt1, Li1, Lu1, Ls1),
        affiche_evolution_Abox(Ls, Lie, Lpt, [(I, and(C1,C2)) | Li], Lu, Abr, Ls1, Lie1, Lpt1, Li1, L
        resolution(Lie1, Lpt1, Li1, Lu1, Ls1, Abr).
```

L'appel de evolur_rec permet de rajouter $I : C_1$ et $I : C_2$ à Lie sans faire de doublon ensuite nous traitons la résolution de la ABox modifée.

4.2.2 Union

On souhaite traiter une instanciation $I : C_1 \sqcup C_2$.

```
transformation_or(Lie, Lpt, Li, [(I, or(C1,C2)) | Lu], Ls, Abr) :-
        write('Utilisation de la règle \u2A06 sur : '),affiche_Abi([(I, or(C1,C2))]),nl,

        write('Br1 : avec '),affiche_concept(C1),nl,
        evolue((I, C1), Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1, Li1, Lu1, Ls1),
        affiche_evolution_Abox(Ls, Lie, Lpt, Li, [(I, or(C1,C2)) | Lu], Abr,
            Ls1, Lie1, Lpt1, Li1, Lu1, Abr),
        resolution(Lie1, Lpt1, Li1, Lu1, Ls1, Abr).

transformation_or(Lie, Lpt, Li, [(I, or(C1,C2)) | Lu], Ls, Abr) :-
        write('Utilisation de la règle \u2A06 sur : '),affiche_Abi([(I, or(C1,C2))]),nl,

        write('Br2 : avec '),affiche_concept(C2),nl,
        evolue((I, C2),Lie, Lpt, Li, Lu, Ls, Lie2, Lpt2, Li2, Lu2, Ls2),
        affiche_evolution_Abox(Ls, Lie, Lpt, Li, [(I, or(C1,C2)) | Lu], Abr,
            Ls2, Lie2, Lpt2, Li2, Lu2, Abr),
        resolution(Lie2, Lpt2, Li2, Lu2, Ls2, Abr).
```

La structure est similaire à l'intersection mais il nous faut maintenant deux implémentations du même prédicat car il y a deux façons de trouver une branche ouverte, elle peut être dans la branche Br_1 ou dans la branche Br_2. Lors d'un appel de transformation_or, les arguments vont s'unifier à la première clause de Horn et va chercher une feuille ouverte dans le premier arbre, s'il n'en trouve pas l'appel va s'unifier à la deuxième clause et chercher dans le deuxième arbre.

C'est le seul cas où les appels de resolution ne sont pas linéaires car il y a ici une recherche en profondeur.

4.2.3 Il existe

On souhaite traiter une instanciation $I_1 : \exists R.C$.

```
complete_some([(I1,some(R,C)) | Lie], Lpt, Li, Lu, Ls, Abr) :-
        write('Utilisation de la règle \u2203 sur : '),affiche_Abi([(I1, some(R,C))]),nl,
        genere(I2),
        evolue((I2, C), Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1, Li1, Lu1, Ls1),
        affiche_evolution_Abox(Ls, [(I1,some(R,C)) | Lie], Lpt, Li, Lu, Abr,
            Ls1, Lie1, Lpt1, Li1, Lu1, [(I1, I2, R) | Abr]),
        resolution(Lie1, Lpt1, Li1, Lu1, Ls1, [(I1, I2, R) | Abr]).
```

L'appel de genere permet de générer une nouvelle instance I_2, on va ensuite introduire l'instanciation $I_2 : C$ dans la Abox grâce à evolue. Il suffit maintenant de faire la résolution de cette nouvelle Abox.

4.2.4 Pour tout

On souhaite traiter une instanciation $I_1 : \forall R.C$.

```
deduction_all(Lie, [(I1, all(R, C)) | Lpt], Li, Lu, Ls, Abr) :-
        write('Utilisation de la règle \u2200 sur : '),affiche_Abi([(I1, all(R,C))]),nl,

        findall((I2, C),  member((I1, I2, R), Abr),  LC2),
        evolue_rec(LC2, Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1, Li1, Lu1, Ls1),
        affiche_evolution_Abox(Ls, Lie, [(I1, all(R, C)) | Lpt], Li, Lu, Abr,
            Ls1, Lie1, Lpt1, Li1, Lu1, Abr),
        resolution(Lie1, Lpt1, Li1, Lu1, Ls1, Abr).
```

On souhaite dans un premier temps trouver l'intégralité des I_2 tel que $< I_1, I_2 >: R$, pour cela nous allons utiliser le prédicat findall fournit par prolog. Nous parcourons dans l'ABox de relation les I_2 tel que $< I_1, I_2 >: R \in Abr$ avec l'aide de member. Avec cela nous créons une liste LI_2 directement sous la forme de (I_2, C). Ensuite il suffit d'ajouter les éléments de LI_2 dans l'Abox avec evolue_rec et résoudre la nouvelle ABox.

Nous n'utilisons pas setof car ce prédicat échoue si aucun élément n'est trouvé, or nous voulons que le programme continue même si aucun I_2 ne convient, LI_2 sera juste vide et rien ne sera ajouté à la ABox.