KRISNI Almehdi - 3800519
JDAY Achraf - 3802410

Rapport du projet de LRC

Ecriture en Prolog d'un démonstrateur basé sur l'algorithme des tableaux pour la logique de description *ALC*

Sommaire

KRISNI Almehdi - 3800519
JDAY Achraf - 3802410

Partie 1 - Préliminaires

La partie 1 est la plus courte des parties à expliquer puisqu'elle ne consiste qu'à la préparation à la démonstration, étant le but final du programme.

Afin de rassembler les informations liées à la TBox et à la ABox, on crée 3 listes allant contenir respectivement les instances (Abi), les rôles (Abr) et les équivalences (equiv).
Ces dernières seront amenées à évoluer au fur et à mesure que l'on soumettra des propositions à la démonstration.
Elles sont créées en utilisant le prédicat **setof** allant rassembler toutes les instances d'une certaine forme au sein d'une liste dont on passe le nom en paramètre.

Par exemple, **setof((X), chien(X), nomsChien)** renvoie la liste contenant tous les noms de chiens, en considérant que le prédicat **chien(X)** représente le fait que l'instance X soit un chien.

On utilise donc le prédicat **premiere_etape(Tbox, Abi, Abr)** allant renvoyer les différentes listes demandées.

```
premiere_etape(Tbox, Abi, Abr) :-
    setof((X, Y), equiv(X, Y), Tbox),

    setof((X, Y), inst(X, Y), Abi),

    setof((X, Y, Z), instR(X, Y, Z), Abr).
```

PS. Il est important de noter que dans le rapport, les corps de fonction seront représentés sans les appels de prédicats effectuant l'affichage et les commentaires afin de ne pas surcharger le rapport.

Une fois que les listes ont été créées, on peut désormais s'intéresser à la partie 2, c'est-à-dire aux types de propositions et comment les saisir afin de les soumettre à la démonstration.

KRISNI Almehdi - 3800519
JDAY Achraf - 3802410

Partie 2 - Saisie de proposition à démontrer

Proposition de type 1

Les propositions de type 1 sont celles représentant le fait qu'une instance **I** appartient à un concept **C**, soit de la forme syntaxique **I : C**. Une instance représente une personne ou un objet tandis qu'un concept est une classe à laquelle l'instance peut appartenir.

Exemple de proposition : **michelAnge : personne**

Proposition de type 2

Les propositions de type 2 sont celles représentant le fait que l'intersection entre deux concepts distincts **C1** et **C2** est négative, soit de la forme syntaxique : **C1** ⊓ **C2** ⊑ ⊥. Cela signifie que l'intersection entre les deux concepts est nulle, c'est-à-dire qu'il n'existe aucune instance dans les données allant appartenir à ces derniers.

Exemple de proposition : **(livre)** ⊓ **(parent)** ⊑ ⊥

Fonctionnement du prédicat de saisie

Le prédicat s'occupant d'initialiser la seconde partie de la démonstration est le prédicat **deuxieme_etape(Abi, Abi1, Tbox)** et elle utilise la Tbox et Abi initialisées lors de la précédente et renvoie la Abi1, soit la Abi ayant été modifiée par l'ajout de la proposition saisie.

```
deuxieme_etape(Abi,Abi1,Tbox) :-
    saisie_et_traitement_prop_a_demontrer(Abi,Abi1,Tbox).
```

Le prédicat **saisie_et_traitement_prop_a_demontrer** permet l'ouverture d'un menu interactif depuis lequel l'utilisateur peut choisir quel type de proposition il souhaite proposer à la démonstration.

```
saisie_et_traitement_prop_a_demontrer(Abi,Abi1,Tbox) :-
     nl,write('Entrer le numero du type de proposition que l"on
souhaite demontrer :'),
      nl,write('\tType 1 - Une instance donnee appartient a un
concept donne.'),
     nl,write('\tType 2 - Deux concepts n-elements en commun dont
l"intersection est vide (negation).'),
    nl,read(R),
    suite(R,Abi,Abi1,Tbox), !.
```

Il s'agit d'un prédicat fourni par l'énoncé, ainsi que le prédicat **suite**, allant permettre la redirection en fonction de l'option choisie par l'utilisateur (1 pour saisir une proposition de type 1 et 2 pour saisir une proposition de type 2).
On ne développe pas les explications sur le dernier prédicat puisqu'il est fourni dans le code de l'énoncé.

Si l'utilisateur a choisi l'option 1, alors on réalise l'appel au prédicat **acquisition_prop_type1** allant permettre cette fois-ci à l'utilisateur de saisir l'instance et le concept composant la proposition.
Il est de la forme :

```
acquisition_prop_type1(Abi,Abi1,Tbox) :-
    nl, write('Veuillez entrer l"instance :'),
    nl, read(I),verificationInstance(I),

    nl, write('Veuillez entrer le concept :'),
    nl, read(C),verificationConcept(C),

    replace(C, RC),
    nnf(not(RC),NRC),

    concat([((I,NRC))],Abi,Abi1).
```

Si l'utilisateur a choisi l'option 2, alors on réalise l'appel au prédicat **acquisition prop type2** allant permettre cette fois-ci à l'utilisateur de saisir les deux concepts composant la proposition.
Il est de la forme :

```
acquisition_prop_type2(Abi,Abi1,Tbox) :-
    nl, write('Veuillez entrer le concept 1 :'),
    nl, read(C1), verificationConcept(C1),

    nl, write('Veuillez entrer le concept 2 :'),
    nl, read(C2), verificationConcept(C2),

    replace(and(C1, C2), RC),
    nnf(RC, NRC),

    genere(Inst),
    concat([((Inst,NRC))], Abi, Abi1).
```

Les prédicats **nnf** (permettant de transformer la proposition en proposition sous forme normale négative), **genere** (permettant la génération de nouvelles instances) et **concat**, (permettant la concaténation de listes) sont fournis par l'énoncé et ne seront donc pas expliqués.

Lorsque l'utilisateur entre une instance **I** ou un concept **C**, il y a un appel au prédicat **verificationInstance(I)** ou **verificationConcept(C)**. Ces derniers vont permettre la vérification de l'existence des données saisies dans la Abox.
Si elles en font partie, alors on continue le programme. Sinon, on retourne au menu de sélection du type de proposition.
(On retrouve le code de ces prédicats à la ligne 290 du fichier *projet.pl*)

KRISNI Almehdi - 3800519
JDAY Achraf - 3802410

Partie 3 - Démonstration de la proposition

Différentes règles

Il existe différentes règles pouvant intervenir lors de la résolution basée sur la méthode des tableaux. Ces règles ont différentes propriétés et doivent donc être traitées de manière différente. On crée donc un prédicat allant permettre l'application de chacune de ces règles.

La règle ∃ - Si, dans Abe, on trouve une assertion de la forme a : **∃R.C**, alors on ajoute les assertions **<a, b> : R et b : C**, où b est un nouvel objet et l'on génère un nouveau noeud de l'arbre de résolution.
On crée alors le prédicat **complete_some** :

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

complete_some([(I1,some(R,C))|Lie],Lpt,Li,Lu,Ls,Abr) :-
     genere(I2),
     Abr1 = Abr,
     evolue((I2,C),Lie,Lpt,Li,Lu,Ls,Lie1,Lpt1,Li1,Lu1,Ls1),
     Abr2 = [(I1,I2,R)|Abr],
     resolution(Lie1,Lpt1,Li1,Lu1,Ls1,[(I1,I2,R)|Abr]).
```

La règle ∀ - Si, dans Abe, on trouve des assertions de la forme **a : ∀R.C** et **<a, b> : R**, alors on ajoute à Abe l'assertion **b : C** et l'on génère un nouveau noeud de l'arbre de résolution.
On crée alors le prédicat **deduction_all** :

```
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) :-
    member((I,B,R),Abr),
    evolue((B,C),Lie,Lpt,Li,Lu,Ls,Lie1,Lpt1,Li1,Lu1,Ls1),
    resolution(Lie1,Lpt1,Li1,Lu1,Ls1,Abr).
```

La règle ⊓ - Si, dans Abe, on trouve une assertion de la forme **a : C ⊓ D**, alors on ajoute à Abe les assertions **a : C** et **a : D** et l'on génère un nouveau noeud de l'arbre de résolution.
On crée alors le prédicat **transformation_and** :

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

transformation_and(Lie,Lpt,[(I,and(A,B))|Li],Lu,Ls,Abr) :-
    evolue((I,A),Lie,Lpt,Li,Lu,Ls,Lie1,Lpt1,Li1,Lu1,Ls1),
    evolue((I,B),Lie1,Lpt1,Li1,Lu1,Ls1,Lie2,Lpt2,Li2,Lu2,Ls2),
    resolution(Lie2,Lpt2,Li2,Lu2,Ls2,Abr).
```

La règle ⊔ - Si, dans Abe, on trouve une assertion de la forme **a : C ⊔ D**, alors on génère deux nouveaux noeuds frères de l'arbre de résolution. Dans l'un, on ajoute à Abe l'assertion **a : C** et dans l'autre, on ajoute à Abe l'assertion **a : D**. Ces deux noeuds sont les racines respectives de deux nouvelles branches de l'arbre de résolution.
On crée alors le prédicat **transformation_or** :

```
transformation_or(Lie,Lpt,Li,[(I,or(C,D))|Lu],Ls,Abr):-
    evolue((I,C),Lie,Lpt,Li,Lu,Ls,Lie1,Lpt1,Li1,Lu1,Ls1),
    evolue((I,D),Lie,Lpt,Li,Lu,Ls,Lie2,Lpt2,Li2,Lu2,Ls2),
    resolution(Lie1,Lpt1,Li1,Lu1,Ls1,Abr),
    resolution(Lie2,Lpt2,Li2,Lu2,Ls2,Abr).
```

Affichage et évolution des listes

Dans les prédicats de transformation de la sous-partie précédente, nous avons fait appel à **evolue** permettant la mise à jour de manière efficace des différentes listes lors de la résolution.
Le prédicat **evolue** est de la forme :

```
evolue((I,and(A,B)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, [(I,and(A,B))|Li], Lu, Ls).
evolue((I,all(R,C)), Lie, Lpt, Li, Lu, Ls, Lie, [(I,all(R,C))|Lpt], Li, Lu, Ls).
evolue((I,some(R,C)), Lie, Lpt, Li, Lu, Ls, [(I,some(R,C))|Lie], Lpt, Li, Lu, Ls).
evolue((I,or(C1,C2)), Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, [(I,or(C1,C2))|Lu], Ls).
evolue(Elem, Lie, Lpt, Li, Lu, Ls, Lie, Lpt, Li, Lu, [(Elem)|Ls]).
```

Afin de pouvoir observer les évolutions au sein des listes tout au long de la résolution, nous avons mis en place le prédicat **affiche_evolution_Abox** auquel nous ferons appel à chaque étape de la résolution après une modifications des éléments de résolution sur lesquels on cherche à résoudre la proposition.
Le prédicat **affiche_evolution_Abox** est de la forme :

```
affiche_evolution_Abox(Lie,Lpt,Li,Lu,Ls,Abr) :-
    nl, write('Liste Abi :'),
    affiche(Lie),
    affiche(Lpt),
    affiche(Li),
    affiche(Lu),
    affiche(Ls),
    nl, write('\nListe Abr :'),
    affiche(Abr), !.
```

Le prédicat affiche permet l'affichage d'une liste d'éléments et d'éléments d'une forme précise, il possède plusieurs formes différentes
(On retrouve le code de ces prédicats à la ligne 454 du fichier *projet.pl*)

KRISNI Almehdi - 3800519
JDAY Achraf - 3802410

Fonctionnement du prédicat de résolution

Après les deux premières étapes, il est temps de s'intéresser à comment la résolution sera effectuée. On doit dans un premier temps effectuer le tri dans la ABox en créant 5 nouvelles listes allant rassembler certains types de propositions précis.
Ces différents types et listes sont :

- la liste Lie des assertions du type **(I,some(R,C))**
- la liste Lpt des assertions du type **(I,all(R,C))**
- la liste Li des assertions du type **(I,and(C1,C2))**
- la liste Lu des assertions du type **(I,or(C1,C2))**
- la liste Ls des assertions restantes, à savoir les assertions du type **(I,C)** ou **(I,not(C))**, **C** étant un concept atomique.

On crée alors le prédicat **tri_Abox** :

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

Le prédicat **resolution** permet de commencer la résolution avec la méthode des tableaux tout en réalisant les tests de clash nécessaires, comme représenté dans la figure de l'énoncé '*Boucle de contrôle du processus de développement de l'arbre de démonstration*'.
On crée alors le prédicat **resolution** :

```
resolution([],[],[],[],Ls,Abr) :-
   not(verificationClash(Ls)), nl, !.

resolution(Lie,Lpt,Li,Lu,Ls,Abr) :-
   verificationClash(Ls), Lie\==[], complete_some(Lie,Lpt,Li,Lu,Ls,Abr).

resolution(Lie,Lpt,Li,Lu,Ls,Abr) :-
   verificationClash(Ls), Li\==[],transformation_and(Lie,Lpt,Li,Lu,Ls,Abr).

resolution(Lie,Lpt,Li,Lu,Ls,Abr) :-
   verificationClash(Ls), Lpt\==[], deduction_all(Lie,Lpt,Li,Lu,Ls,Abr).

resolution(Lie,Lpt,Li,Lu,Ls,Abr) :-
   verificationClash(Ls), Lu\==[], transformation_or(Lie,Lpt,Li,Lu,Ls,Abr).
```

Chacune des différentes formes du prédicat permet de gérer chacune des 4 règles évoquées plus haut dans le rapport (voir partie **Différentes règles**).

On crée également le prédicat **verificationClash** :

```
verificationClash([(I,C)|Ls]) :-
    nnf(not(C),NC),
    member((I,NC),Ls),
    write('\nOn trouve un clash avec : '),
    write(I), nl.

verificationClash([_|Ls]) :-
    verificationClash(Ls).
```

Un clash a lieu lorsque 2 propositions dans les listes sont contradictoires.
Prenons l'exemple où l'on a les propositions suivantes :

david : sculpture
david : not(sculpture)

Dans cet exemple, il y a une incohérence puisque David ne peut pas être une statue et ne pas être une statue en même temps (impossible que **A** et **not(A)** coexistent pour la même instance). Il y a donc un clash qui empêche la résolution de progresser.
On se situe donc sur une feuille fermée et on doit alors continuer la résolution sur les autres branches.

Un affichage est normalement toujours effectué lorsqu'un clash a lieu pendant l'exécution du programme.

KRISNI Almehdi - 3800519
JDAY Achraf - 3802410

Jeux de tests sur les différents prédicats

Dans cette partie, nous allons mettre en avant comment chacun des prédicats évoqués dans le rapport fonctionne à travers quelques tests basiques.

PS. Tous les tests sont réalisés avec les données de la TBox et de la ABox fournies avec le sujet.

acquisition_prop_type1(Abi,Abi1,Tbox)

Lors de l'appel au prédicat, il faut veiller à ne pas entrer de noms d'instances ou de concepts n'existant pas dans les données du fichier.
Si on entre l'instance 'michelAnge' et le concept 'sculpteur', on obtient **True**.
Si on entre l'instance 'michelAnge' et le concept 'livre', on obtient **False**.

acquisition_prop_type2(Abi,Abi1,Tbox)

Comme pour le prédicat précédent, il faut veiller à ne pas entrer de noms d'instances ou de concepts n'existant pas dans les données du fichier.
Si on entre les concepts 'livre' et 'sculpture', on obtient **True**.
Si on entre les concepts 'sculpteur' et 'auteur', on obtient **False**.

affiche_evolution_Abox

Il n'y a pas de cas d'exécution correcte ou incorrecte lors de l'appel à ce prédicat. Il effectue tout simplement l'affichage des différentes listes de propositions lors de l'exécution.

Lors de la résolution de la proposition de type 1 '*michelAnge : sculpteur*', on aura l'affichage initial suivant :

```
Liste Abi :
        michelAnge : not personne or  all.aCree.not sculpture
        david : sculpture
        joconde : objet
        michelAnge : personne
        socrate : personne
        sonnets : livre
        vinci : personne

Liste Abr :
        <michelAnge, david> : aCree
        <michelAnge, sonnets> : aEcrit
        <vinci, joconde> : aCree
```

KRISNI Almehdi - 3800519
JDAY Achraf - 3802410

Conclusion

Pour conclure, nous pensons qu'il est important de noter le fait que notre code ne renvoie pas toutes les réponses attendues.

Les propositions de type 1 sont, à quelques exceptions prêt, gérées par notre système sans aucune difficulté. Les propositions de type 2 quant à elles ne sont pas entièrement prises en charge par nos prédicats puisque si les plus basiques permettent d'obtenir des réponses correctes, il suffit d'utiliser des propositions à peine plus complexes pour que l'on obtienne des résultats erronés.

Le langage Prolog, aussi intéressant qu'il soit en raison du fait qu'il s'agisse d'un langage de programmation logique, nous a posé plusieurs problèmes lors de la création du code.
Ayant été habitués aux langages de programmation plus 'classiques' lors de nos années en licence d'Informatique, il arrive parfois que nous ayons du mal à utiliser des langages ne faisant pas partie de notre domaine d'expertise.

Mais cette expérience nous a appris à découvrir les bases d'un nouveau genre de langage et qu'à l'avenir, cette dernière nous permettra de mieux appréhender les futurs projets que nous aurons à réaliser.