Projet de LRC

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

Yanis Djermouni

Andy Torres

Année : 2023/2024

SORBONNE UNIVERSITÉ

Table des matières

1 Introduction

1.1 Découpage du code

Nous avons séparé le code en plusieurs fichiers, chacun ayant sa propre utlité :

— Boxs.pl : C'est dans ce fichier que l'utilisateur entre la TBox et la ABox initiale.

— main.pl : Ce fichier contient le prédicat `programme` qui appelle les différentes étapes permettant la résolution.

— pt1.pl : Contient les fonctions liées à la première partie décrite par le sujet.

— pt2.pl : Contient les fonctions liées à la deuxième partie décrite par le sujet.

— pt3.pl : Contient les fonctions liées à la troisième partie décrite par le sujet.

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 `intro.pl` à la racine du projet. On utilisera les prédicats suivants :

— 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 signalera toute erreur grammaticale ou syntaxique, ainsi que la détection d'une Tbox cyclique.

1.2.2 Exécution et saisie de la proposition à démontrer

Pour exécuter le programme, saisissez la commande suivante à la racine du projet : swipl -f main.pl. Ensuite, dans l'interpréteur Prolog, utilisez le prédicat `programme` pour démarrer le programme.

2 Étape préliminaire de contrôle et de mise en forme de la TBox et de la ABox

2.1 Correction syntaxique et sémantique

Dans cette phase initiale, nous débutons par vérifier la correction syntaxique et sémantique des deux boîtes. Pour ce faire, nous implémentons les prédicats concept, instance, role qui vérifient si des objets sont de ce type. Les cas de base sont les suivants : concept, instance, role qui vérifient si des objets sont de ce type. Les cas de base sont les suivants :

- concept(C) :- cnamea(C).
- concept(C) :- cnamena(C).
- instance(I) :- iname(I).
- role(R) :- rname(R).

Le prédicat concept requiert de la récursivité en raison des concepts non atomiques. Nous vérifions la grammaire avec les prédicats suivants :

- concept(not(C)) :- concept(C).
- concept(and(C, B)) :- concept(C), concept(B).
- concept(or(C, B)) :- concept(C), concept(B).
- concept(some(R, C)) :- role(R), concept(C).
- concept(all(R, C)) :- role(R), concept(C).

Nous utilisons ces prédicats pour définir le prédicat verifTbox qui prend en argument une TBox sous forme d'une liste d'équivalences et qui vérifie la correction syntaxique et sémantique :

- verifTbox([(C, D) | L]) :- cnamea(C), concept(D), verifTbox(L).
- verifTbox([]).

De même, nous avons le prédicat verifAboxFunc pour la ABox qui vérifie les instances et les rôles :

- verifAboxFunc([(I, C) | L], inst) :- instC(I, C), verifAboxFunc(L, inst).
- verifAboxFunc([(I, J, R) | L], role) :- instR(I, J, R), verifAboxFunc(L, role).
- verifAboxFunc([], _).

2.2 Vérification de l'auto-référencement

Nous souhaitons également nous assurer qu'il n'y a pas d'auto-référencement dans la définition des concepts non atomiques. Pour cela, nous utilisons le prédicat verifAutoref :

- verifAutoref([C | L]) :- equiv(C, E), (autoref(C, E, []); verifAutoref(L)).

La fonction autoref est définie comme suit :

- autoref(C, not(D), L) :- autoref(C, D, L).

— autoref(C, and(C, D), L) :- autoref(C, C, L) ; autoref(C, D, L).
— autoref(C, or(C, D), L) :- autoref(C, C, L), autoref(C, D, L).
— autoref(C, all(, D), L) :- autoref(C, D, L).
— autoref(C, some(, D), L) :- autoref(C, D, L).
— autoref(C, C, L) :- member(C, L) ; cnamena(C), concat(L, [C], M), equiv(C, Z), autoref(C, Z, M).
— pas_autoref(C, CGen, L) :- + autoref(C, CGen, L).

2.3 Mise sous forme

Une fois que nous sommes certains qu'il n'y a pas d'auto-référencement, nous pouvons développer les concepts non atomiques et les mettre sous forme normale négative (NNF). Nous implémentons les prédicats nnf pour convertir les concepts en NNF et atomF pour la conversion d'expressions de concept en atomes. Les prédicats traitement_Tbox et traitement_Abox sont utilisés pour transformer la TBox et la ABox respectivement :

— traitement_Tbox([], A, L) :- L = A.
— traitement_Tbox([(C, CGen) | L], A, N) :- pas_autoref(C, CGen, []), atomF(CGen, Y), nnf(Y, X), concat(A, [(C, Y)], M), traitement_Tbox(L, M, N).
— traitement_Abox([], A, L) :- L = A.
— traitement_Abox([(C, CGen) | L], A, N) :- pas_autoref(C, CGen, []), atomF(CGen, Y), nnf(Y, X), concat(A, [(C, Y)], M), traitement_Abox(L, M, N).

Le prédicat nnf est conçu pour gérer divers cas de concepts lors de la conversion en NNF :

— nnf(not(and(C, D)), or(CN, DN)) :- nnf(not(C), CN), nnf(not(D), DN), !.
— nnf(not(or(C, D)), and(CN, DN)) :- nnf(not(C), CN), nnf(not(D), DN), !.
— nnf(not(all(R, C)), some(R, CN)) :- nnf(not(C), CN), !.
— nnf(not(some(R, C)), all(R, CN)) :- nnf(not(C), CN), !.
— nnf(not(not(X)), Y) :- nnf(X, Y), !.
— nnf(not(X), not(X)) :- !.
— nnf(and(C, D), and(CN, DN)) :- nnf(C, CN), nnf(D, DN), !.
— nnf(or(C, D), or(CN, DN)) :- nnf(C, CN), nnf(D, DN), !.
— nnf(some(R, C), some(R, CN)) :- nnf(C, CN), !.
— nnf(all(R, C), all(R, CN)) :- nnf(C, CN), !.
— nnf(X, X).

3 Saisie de la proposition à montrer

Dans la seconde partie, l'utilisateur a la possibilité d'entrer deux types de propositions à démontrer :

— Une proposition de type $I : C$ qui sera prise en charge par le prédicat acquisition_prop_type1.

— Un proposition de type $C1 \sqcap C2 \sqsubseteq \bot$ qui sera prise en charge par le prédicat acquisition_prop_type2.

3.1 Proposition de type 1 : $I : C$

On commence par demander à l'utilisateur d'entrer Abi , $Abi1$ et $Tbox$ à l'aide du prédicat acquisition_prop_type1. C'est également à ce stade que l'on vérifie si l'entrée de l'utilisateur est correcte syntaxiquement avec le prédicat instC :

— acquisition_prop_type1(Abi, Abi1, Tbox) :-
write('I '), read(I),
write('C '), read(C),
nl, (instC(I, C) -> writef('write('Instance ou Concept erroné, réessayez.'), nl, fail), nl, atomF(not(C), Y), nnf(Y, X), concat(Abi, [(I, X)], Abi1).

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

De manière analogue, on utilise le prédicat acquisition_prop_type2 pour la proposition de type 2 :

— acquisition_prop_type2(Abi, Abi1, Tbox) :-
write('C '), read(C),
write('D '), read(D),
nl, (concept(C), concept(D) -> writef("write('Au moins un des concepts est non-déclaré, réessayer.'), nl, fail), nl, generer_random_Iname(RandI),
atomF(not(and(C, D)), Y), nnf(Y, X), concat(Abi, [(RandI, X)], Abi1).

4 Démonstration de la proposition

4.1 Algorithme de résolution

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

— resolution(Lie, Lpt, Li, Lu, Ls, Abr) :- test_clash(Ls, Ls).

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

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

— resolution(Lie, Lpt, Li, Lu, Ls, Abr) :- transformation_and(Lie, Lpt, Li, Lu, Ls, Abr).

— resolution(Lie, Lpt, Li, Lu, Ls, Abr) :- transformation_or(Lie, Lpt, Li, Lu, Ls, Abr).

— resolution(Lie, Lpt, Li, Lu, Ls, Abr) :- deduction_all(Lie, Lpt, Li, Lu, Ls, Abr).

À chaque étape, nous vérifions l'absence de clash avant de procéder au traitement.

4.1.1 Intersection

Pour traiter une instanciation $I : C_1 \sqcap C_2$, nous utilisons transformation_and pour diviser la proposition en ses composantes et les ajouter à la ABox.

— transformation_and(Lie, Lpt, [(A, and(C, D)) | Li], Lu, Ls, Abr) :- evolue((A, and(C, D)), Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1, Li1, Lu1, Ls1), resolution(Lie1, Lpt1, Li1, Lu1, Ls1, Abr).

4.1.2 Union

Pour $I : C_1 \sqcup C_2$, transformation_or est utilisé pour gérer les deux branches potentielles du tableau.

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

4.1.3 Existence

Pour $I : \exists R.C$, complete_some génère une nouvelle instance et l'ajoute à la ABox.

— complete_some([(A, some(R, C)) | Lie], Lpt, Li, Lu, Ls, Abr) :- generer_random_Iname(B), evolue((B, C), Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1, Li1, Lu1, Ls1), concat(Abr, [(A, B, R)], Abr1), resolution(Lie1, Lpt1, Li1, Lu1, Ls1, Abr1).

4.1.4 Universalité

Pour $I : \forall R.C$, deduction_all applique le concept C à toutes les instances liées par R.

— deduction_all(Lie, [(A, all(R, C)) | Lpt], Li, Lu, Ls, Abr) :- getAbr(A, R, Abr, [], Instances), bc(C, Instances, Lie, Lie1), resolution(Lie1, Lpt, Li, Lu, Ls, Abr).

Ces prédicats, combinés à la méthode des tableaux, permettent une résolution systématique des propositions de la ABox, en cherchant des feuilles ouvertes indiquant des solutions valides.