Logique et Représentation des Connaissances

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

Natacha Rivière - 28706745
Léa Movsessian - 28624266

Année : 2023/2024

SCIENCES SORBONNE UNIVERSITÉ

Table des matières

1 Introduction

L'objectif de ce projet est de construire en `Prolog` un démonstrateur de formule logique en se basant sur l'algorithme des tableaux pour la logique de description *ALC*.

Le projet est découpé en trois parties. Pour chaque partie, nous décrirons chacun des prédicats que nous avons défini.

1.1 Rappel des éléments fournis

Les données de la *Abox* et de la *Tbox* sont définies en utilisant les prédicats suivants:

- `equiv(ConceptNonAtom,ConceptGen)`
- `inst(Instance,ConceptGen)`
- `instR(Instance1, Instance2, role)`
- `cnamea(ConceptAtom)`
- `cnamena(ConceptNonAtom)`
- `iname(Instance)`
- `rname(Role)`

1.2 Fonctionnement du démonstrateur

Pour utiliser le démonstrateur, il faut utiliser un interpréteur prolog, ici nous utilisons *swipl*. Afin de lancer le programme, il faut tout d'abord charger un fichier dans lequel sont définis la *Abox* et *Tbox* dont nous allons nous servir. Les données fournies dans l'énoncé se trouvent dans le fichier `TBox_ABox_exo3_td4.pl`.

Voici un déroulé d'utilisation du démonstrateur :

- Lancer l'interpréteur avec `swipl -s .\TBox_ABox_exo3_td4.pl`.
- Charger le démonstrateur avec `[démonstrateur_riviere_movsessian].`
- Lancer le démonstrateur avec la commande `programme.`
- Suivez les instructions qui s'affichent.
- Le programme vous affiche `Youpiiiiii, on a demontre la proposition initiale !!! true.` si votre proposition est démontrée, et `false` sinon.

2 Partie 1 - Etape préliminaire de vérification et de mise en forme de la *Tbox* et de la *Abox*

L'objectif de cette prmière partie est de définir un analyseur sémantique et syntaxique des *Abox* et des *Tbox* qui seront fournies au démonstrateur.

2.1 Description des prédicats

2.1.1 `premiere_etape(Tbox, Abi, Abr)`

Ce prédicat est fait pour être utilisé avec trois inconnues en paramètre. Il récupère les données de la *Tbox* et de la *Abox* définies dans le fichier grâce au prédicat `setof`. Il vérifie ensuite la syntaxe et la sémantique de tous les éléments des listes qu'il extrait. Enfin, il renvoie ces listes dans lesquelles toutes les définitions sont développées au maximum et mises sous forme normale négative.

2.1.2 `concept(C)`

Vérifie que *C* est bien un concept défini dans la *Tbox* de notre démonstration. Ce prédicat ne peut être utilisé afin d'énumérer tous les concepts possibles car il y en a une infinité. On peut syntaxiquement construire un concept à l'infini en ajoutant des `not` devant un concept. Par conséquent, ce prédicat utilise le `cut` pour s'arrêter dès qu'une solution est trouvée. Cela permet d'éviter les boucles infinies.

2.1.3 `verification_Tbox(L)`

- `definition(C1, C2)`

  Vérifie qu'un couple correspond bien à la définition d'un concept simple (au sens de n'utilisant pas d'opérateur). Il renvoie vrai lorsque *C1* est un concept simple, et que *C2* est un concept.

Le prédicat `verification_Tbox` parcourt récursivement la liste *L* des définitions présentes dans la *Tbox* jusqu'à ce qu'elle soit vide. À chaque itération, le premier élément de la liste est vérifié avec `definition`.

2.1.4 `verification_Abox(AboxC, AboxR)`

Le prédicat `verification_Abox` prend en entrée deux listes, AboxC représente les assertions de concept et AboxR contient les assertions de rôle de la Abox à étudier. Il renvoie *true* si les deux listes contiennent des éléments sémantiquement corrects.

- `instanceC\2`

  Vérifie si les paramètres fournis correspondent bien à une instanciation de concept.

- `instanceR\3`

  Vérifie si les paramètres fournis correspondent bien à une instanciation de rôle.

- `verification_AboxC(L)`

  Vérifie récursivement que tous les éléments d'une liste sont des assertions de concept.

- `verification_AboxR(L)`

  Vérifie récursivement que tous les éléments d'une liste sont des assertions de rôle

2.1.5 `autoref(L)`

Prend en entrée une liste L représentant une Tbox. Renvoie *true* si elle ne contient pas de définition auto-référente.

- `pas_autoref(C, C1)`

  Développe les concepts non atomiques de *C1*, s'il y en a. Renvoie *false* si C apparaît dans la définition de *C1*. La récursion s'arrête lorsque *C1* est atomique.

2.1.6 `traitement_Tbox(L1, L2)` / `traitement_Abox(L1, L2)`

Ces deux prédicats fonctionnent de la même manière, et sont indépendants l'un de l'autre. Ils traitent récursivement une liste *L1* et créent la liste *L2*.

Pour chaque couple *(C, DefC)* de la liste *L1*, on vérifie que *DefC* est bien un concept avant de le décomposer au maximum afin de n'obtenir que des concepts atomiques. Ensuite on met l'expression trouvée sous forme normale négative. La liste *L2* est alors constituée des couples formés de C et de la forme normale négative de l'expression développée de *DefC*.

- `developpement_atomique(C, C1)`

  Prend en entrée C un concept, et constitue *C1*, l'expression développée de C ne contenant que des concepts atomiques. Renvoie *true* si *C1* est le développement atomique de C.

3 Partie 2 - Saisie de la proposition à démontrer

Cette partie a pour objectif d'acquérir la proposition que l'utilisateur souhaite démonter et de vérifier sa validité.

3.1 Description des prédicats

3.1.1 `acquisition_prop_type1(Abi, Abi1,Tbox)`

Permet de lire au clavier l'entrée de l'utilisateur du programme s'il décide d'entrer une proposition de la forme I : C. Si les données entrées par l'utilisateur ne sont pas valides, c'est à dire que ce ne sont pas un nom d'instance et de concepts connus, le prédicat renvoie *false*. Il prend en entrée *Abi* la liste des assertions de concept de la Abox et crée *Abi1*, cette même liste à laquelle est ajoutée la négation de l'assertion entrée par l'utilisateur. Le troisième paramètre n'est pas utilisé.

- `lecture_prop_1(I,C)`

  Lis l'entrée utilisateur au clavier et en vérifie la sémantique.

  - `verification_lecture(I,C)`

    Vérifie que I : C est bien une définition sémantiquement correcte d'instance. Échoue si ce n'est pas le cas, en affichant un message.

3.1.2 `acquisition_prop_type2(Abi, Abi1,Tbox)`

Permet de lire au clavier l'entrée de l'utilisateur du programme s'il décide d'entrer une proposition de la forme $C1 \sqcap C2 \sqsubseteq \bot$. Si les données entrées par l'utilisateur ne sont pas valides, c'est à dire que ce ne sont des concepts connus, le prédicat renvoie *false*. Il prend en entrée *Abi* la liste des assertions de concept de la Abox et crée *Abi1*, cette même liste à laquelle est ajoutée la négation de l'assertion entrée par l'utilisateur. Le troisième paramètre n'est pas utilisé.

- `lecture_prop_2(C1, C2)`

  Lis l'entrée utilisateur au clavier et en vérifie la sémantique.

  - `verification_lecture2(C1,C2)`

    Vérifie que $C1 \sqcap C2 \sqsubseteq \bot$ est bien une définition sémantiquement correcte. Échoue si ce n'est pas le cas, en affichant un message.

4 Partie 3 - Démonstration de la proposition

4.1 Description des prédicats

4.1.1 `tri_Abox(Abi, Lie, Lpt, Li, Lu, Ls)`

Répartit les éléments de *Abi*, la liste des assertions de concept de la Abox dans 5 listes différentes en fonction de leur structure.

- 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.

On consière que toutes les assertions sont sémantiquement et syntaxiquement correctes, donc si on parvient au cas traitant les "autres" assertions, c'est qu'elles sont de type `(I,C)` ou `(I,not(C))` avec C un concept atomique. De plus on utilise des `cut` pour empêcher prolog de considérer qu'il y a deux propositions possibles à tester.

4.1.2 `evolue(A, Lie1, Lpt1, Li1, Lu1, Ls1, Lie2, Lpt2, Li2, Lu2, Ls2)`

Intègre A à la liste lui correspondant selon les mêmes critère que `tri_Abox`. Ce prédicat n'ajoute pas A à une liste s'il y est déjà présent. Il y a donc deux cas pour chaque forme possible. Dans le code, on retrouve d'abord celle où A fait déjà partie de la liste où il devrait être rangé. Dans ce cas là, un test d'appartenance est fait. Dans l'autre cas, aucun test n'est fait car grâce à l'usage des `cut` on peut être certains que si on arrive au deuxième cas, c'est que le précédent n'est pas passé, donc A n'était pas déjà présent dans la Abox.

4.1.3 `complete_some(Lie, Lpt, Li, Lu, Ls, Abr)`

Extrait le premier élément de la liste Lie, de la forme `a : ∃R.C`, l'affiche, et applique la règle ∃. Ajoute donc `<a,b> : R` et `b: C` où b est un nouvel objet généré grâce au prédicat `genere\1`.

Crée ensuite un nouveau noeud de l'arbre avec un appel à `resolution` avec les nouvelles listes mises à jour par `evolue`.

4.1.4 `transformation_and(Lie, Lpt, Li, Lu, Ls, Abr)`

Extrait le premier élément de la liste Li, de la forme `a : C ⊓ D`, l'affiche, et applique la règle ⊓. Ajoute donc `a: C` et `a: D` aux assertions de concept.

Crée ensuite un nouveau noeud de l'arbre avec un appel à `resolution` avec les nouvelles listes mises à jour par `evolue`.

4.1.5 `deduction_all(Lie, Lpt, Li, Lu, Ls, Abr)`

Extrait le premier élément de la liste Lpt, de la forme `a: ∀R.C` l'affiche, et applique la règle ∀ autant de fois que possible en cherchant toutes les occurrences de relations de la forme `<a,b>: R`. Ajoute donc `b: C` aux assertions de concept. Tant qu'il est possible d'appliquer `a: ∀R.C`, les assertions de rôle traitées sont retirées mais pas l'assertion de concept.

Une fois toutes les assertions de rôle correspondantes traitées, `a: ∀R.C` est retiré des assertions de concept et on crée un nouveau noeud grâce à un appel à `résolution` avec les nouvelles listes mises à jour.

4.1.6 `transformation_or(Lie, Lpt, Li, Lu, Ls, Abr)`

Extrait le premier élément de la liste Lu, de la forme `a : C ⊔ D`, l'affiche, et applique la règle ⊔.

Cette règle crée deux nouveaux noeuds grâce à deux appels à `resolution`. Dans le premier, on ajoute l'assertion `a : C`, dans le deuxième `a : D`.

4.1.7 `resolution(Lie, Lpt, Li, Lu, Ls, Abr)`

Ce prédicat est essentiel au fonctionnement du démonstrateur. C'est grâce à lui que nous appliquons chaque règle de l'algorithme des tableaux. il prend en entrée 6 listes, les 5 premières correspondent aux listes triées par `tri_Abox` et la dernière aux assertions de rôles. Ce prédicat utilise les règles dans l'ordre défini dans le sujet, même si les cas sont exposés dans l'ordre inverse dans le code. En effet, si la liste des assertions construites avec ∃ n'est pas vide, tous les cas précédemment définis sont passés, car ils requièrent une liste vide en premier argument. Le même principe s'applique pour toutes les autres règles. Cet ordre permet d'éviter que prolog n'appelle toutes les règles lorsqu'il constate qu'il n'y a pas de clash dans la branche étudiée, il ne peut pas faire de backtracking.

- `no_clash(L)`

  Parcourt la liste L et renvoie *false* s'il y a un clash dans la liste. C'est à dire qu'il y a une assertion de la forme `I : C` et une autre de la forme `I : not(C)`. Un clash est détecté par le prédicat `clash(L)` qui calcule la forme normale négative de `not(C)` et et teste son appartenance à L.

  Si un clash est détecté, il y a alors un affichage pour le mettre en évidence.

4.1.8 `affiche_evolution_Abox\12`

Permet d'afficher le contenu de la Abox d'une étape sur l'autre. Les 6 premiers arguments sont les listes d'assertions dans le noeud précédent, les 6 suivants les listes mises à jour pour le noeud suivant.

Ce prédicat utilise plusieurs prédicats annexes qu'il ne semble pas utile de développer.