Alexandre XIA
Charline CURAUT
Groupe 5

RAPPORT PROJET LRC

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

SOMMAIRE

- **evolue** (ajoute l'assertion dans la liste correspondante à l'élément trouvé)
- **evolue_all** (prédicat spécifique à deduction_all qui ajoute tous les couples d'assertions possibles dans la liste correspondante)
- Les 4 règles de résolution : **complete_some, transformation_and, deduction_all, transformation_or**
- **affiche_evolution_Abox** (affiche l'évolution de la Abox après application d'une règle)
- **afficheAbox** (prédicat intermédiaire à affiche_evolution_Abox pour afficher les différents types d'éléments)

PARTIE 1 - Etape préliminaire de vérification et de mise en forme de la Tbox et de la Abox

concept (C)

Cette étape préliminaire consiste en outre à préparer la Abox et la Tbox. On doit vérifier tout d'abord à partir du prédicat **concept** la sémantique des concepts, c'est-à-dire si un concept atomique est bien un **cnamea** ou si un concept complexe est bien un **cnamena.** À partir de là, on peut agrandir la définition du prédicat **concept** et vérifier la correction syntaxique en traitant chaque cas possible de la grammaire des concepts et en utilisant les concepts atomiques déjà vérifiés. Le prédicat **concept** ne prend donc qu'un argument C qui est le concept qu'on veut vérifier.

Par exemple : **concept(personne) ou concept(and(personne,sculpture)),** renverront vrai (l'élément existe bien et c'est un concept atomique), **concept(auteur)** renverra aussi vrai (on vérifie s'il a une expression contenant que des identificateurs de rôles atomiques).

Remarque : Il nous est donné le prédicat **setof** qui n'est pas forcément nécessaire dans la vérification d'un concept (on peut avoir **cnamea(personne) => concept(personne).** directement).

role (R) et instance (I)

De la même façon, on vérifie la sémantique des rôles et des instances (en utilisant les prédicats donnés par le sujet **rname** et **iname** respectivement). Chacun de ces prédicats ne prend que le rôle R (resp. l'instance I) qu'il veut vérifier, en argument.

autoref (C) et autoref (C , Def)

Nous devons ensuite vérifier qu'un concept complexe C n'est pas autoréférent. Un concept est dit autoréférent lorsqu'il se définit lui-même. Nous allons donc parcourir les différents concepts attachés au concept C via sa définition Def que l'on récupère grâce au prédicat **equiv(C,Def)**. Et lorsqu'on rencontrera un concept complexe, on le remplacera par sa définition dans la TBox et on poursuivra l'analyse. Dès qu'un concept rencontré est identique au concept C, on a un concept autoréférent. Cependant, il faut aussi pouvoir gérer le cas où le concept autoréférent est interne (indirectement lié à C), on proposera dans notre cas d'utiliser une liste qui permettra avec le prédicat member de vérifier qu'un concept n'a pas déjà été rencontré (méthode vue en TME), si c'est le cas, on a un concept autoréférent.
On a créé deux prédicats :

- **autoref(C)** n'a qu'un argument et est défini comme suit : **equiv(X,Y)** et **autoref(X,Y) => autoref(C)**. On l'utilise par la suite dans le projet puisqu'il est plus simple de

n'avoir qu'un argument à gérer (sinon on doit appeler equiv pour récupérer Def dans tous les autres prédicats du projet).

- **autoref(C, Def)** : a deux arguments et renvoie vrai ou faux si le concept est autoréférent ou non. C'est lui qui parcourt et agit sur Def.

remplace_concept (C, R)

Pour finaliser cette étape préliminaire, il faut ensuite transformer la TBox et la ABox pour que chaque concept ou instance ne soit défini que par des concepts atomiques, et mis sous forme normale négative.

Le prédicat **remplace_concept** a donc pour rôle de préparer la TBox et la ABox en remplaçant les expressions contenant des concepts complexes par des expressions ne contenant plus que des concepts atomiques. On utilisera à nouveau **equiv** qui permet de traduire un concept complexe par une expression avec des identificateurs de concept atomique (et complexes qui seront aussi traités).
Il prend en argument le concept C à traiter et la réponse R qui sera rendue.

traitement_Tbox (C, R) et traitement_Abox (inst (I, Complex), inst (I, R))

Enfin, ces derniers prédicats qui sont très similaires vont finir de préparer la TBox en mettant les concepts sous forme normale négative (grâce au prédicat **nnf** fourni dans le sujet) après les avoir modifiés pour que leur définition ne contienne que des concepts atomiques (par un appel à **remplace_concept**). **traitement_TBox** prend donc deux arguments: le concept C à traiter ou vérifier, et le traitement R de C.

Le traitement de la Abox étant identique, il se fera par appel de **traitement_Tbox**. Le prédicat **traitement_ABox** prend aussi deux arguments (ici des instances inst(I,C)) de la même manière que pour **traitement_TBox** et appelle ce prédicat pour modifier le concept associé à l'instance I

Cela conclut donc la partie préliminaire de la mise en place des concepts ainsi que de la mise en forme des Abox et Tbox.

PARTIE 2 - Saisie de la proposition à démontrer

Dans cette partie, nous allons commencer à implémenter le programme qui permettra à l'utilisateur de fournir les propositions qu'il souhaite démontrer. Le prédicat **programme** appelle trois prédicats **premiere_etape**, **deuxieme_etape** et **troisieme_etape**, chacun correspondant aux différentes parties du sujet (i.e. étape préliminaire, étape de saisie des propositions, étape de résolution). Dans cette partie, nous n'implémentons que les deux premiers.

liste_autoref(L, LR) , is_autoref_TBox(LR) et is_autoref_TBox(LR, X)

Tout d'abord, lorsqu'on lance le programme, on doit vérifier si la TBox n'est pas cyclique car sinon, la TBox sera inexploitable par le démonstrateur et on devra écrire un message d'erreur à l'utilisateur. On a donc créer le prédicat **liste_autoref** qui parcourt une liste de concepts (premier argument) et donne en deuxième argument une autre liste remplie de 0 (si le concept i n'est pas autoréférent) ou 1 (sinon). On utilise ensuite un autre prédicat pour appliquer cette fonction à la TBox et renvoyer vrai s'il y a un 1 dans la liste de **liste_autoref** ou faux sinon. Il y a donc deux versions :

- **is_autoref_TBox(LR)** n'a qu'un argument qui est le résultat de **liste_autoref** appliqué à la liste des concepts non atomiques (on a utilisé **setof**).
- **is_autoref_TBox(LR,X)** a deux arguments : LR est le même que pour **is_autoref_TBox(LR)** puisqu'on a appelé ce prédicat, et X prend la valeur 0 (il n'y a aucun 1 dans LR) ou 1 (il y a au moins un 1 dans LR et donc au moins un concept autoréférent dans la TBox).

liste_nnf(L1, L2)

Une fois que nous aurons vérifié que la TBox n'est pas cyclique, il faudra la modifier (ainsi que la ABox) en appliquant **traitement_TBox** à chaque concept. C'est ce que va faire le prédicat **liste_nnf** en renvoyant le résultat du traitement de L1 (son premier argument) dans L2.

premiere_etape(TBox, Abi, Abr) et suite_etape1(X, TBox, Abi, Abr)

On arrive enfin à la première fonction qui sera appelée par le programme : **premiere_etape**. Ce prédicat va permettre de traiter le cas où il y a une définition cyclique dans la TBox. En fonction du résultat de l'appel à **is_autoref**, on récupère le résultat X et on l'utilise dans le prédicat **suite_etape1** comme condition. Si X = 1, alors on renvoie un message à l'utilisateur pour le prévenir qu'il y a un problème avec la TBox. Si X = 0, on transforme finalement la TBox puis la ABox via **liste_nnf (**et **setof** pour créer les listes).

acquisition_prop_type1(Abi, Abi1, TBox) et suite_saisie_type1(I, C, Abi, Abi1, TBox)

Une fois la première étape finalisée, le programme va appeler le predicat **deuxieme_etape** qui nous était fournie. Ce prédicat fait indirectement appel à **acquisition_prop_type1** ou **acquisition_prop_type2** en fonction des choix de l'utilisateur. Ces deux prédicats étaient à écrire. Nous rappelons donc ici que nous avons deux types de propositions : les propositions de type I et de type II.

Pour les propositions de type I, de la forme $I : C$ avec **I** une instance et **C** un concept, nous utiliserons le prédicat **acquisition_prop_type1** qui demandera à l'utilisateur de fournir tout d'abord un nom d'instance, puis un nom de concept qui seront tous deux récupérés en premiers arguments de **suite_saisie_type1**. Dans ce prédicat et pour que le programme ait un intérêt, nous vérifions que le nom d'instance et de concept existent dans les données, sinon la proposition ne peut pas être démontrée.

Par exemple pour "**michelAnge : sculpteur**", l'utilisateur devra entrer en premier "**michelAnge.**" puis "**sculpteur.**". Une fois la proposition vérifiée, on ajoute la négation " $I : \neg C$" dans la Abox après l'avoir traité avec traitement_Tbox. On obtient donc une nouvelle Abox nommée **Abi** stockée comme argument de **acquisition_prop_type1** et **suite_saisie_type1**.

acquisition_prop_type2(Abi, Abi1, TBox) et suite_saisie_type2(C1, C2, Abi, Abi1, TBox)

Pour les propositions de type II, de la forme : $C1 \sqcap C2 \sqsubseteq \perp$, nous utiliserons le prédicat **acquisition_prop_type2** qui réalisera des opérations similaire au premier en récupérant les deux concepts C1 et C2, et en les stockant aussi comme premiers arguments de **suite_saisie_type2**. Ce prédicat vérifiera l'existence des concepts C1 et C2 et créera une instance à partir du prédicat **genere** fourni en annexe, puis ajoutera la négation " $I : C1 \sqcap C2$" traitée à la Abox. Ce sera aussi Abi que l'on retrouve dans les autres prédicats.

Par exemple pour **editeur** ⊓ **auteur,** l'utilisateur devra entrer en premier "**editeur.**" puis "**auteur.**" (ou inversement).

Remarque 1: Le code pour lancer le programme est fourni dans cette partie, il faudra faire dans l'ordre les appels :

? - programme.
? - "1." ou "2."

Dans le cas 1. :
? - nomInstance.
? - nomConcept.

Dans le cas 2. :
? - nomConcept1.
? - nomConcept2.

Remarque 2 : Chaque prédicat qui contient un prédicat de type "**suite_…**" traite les cas d'erreurs en renvoyant un message à l'utilisateur pour lui donner la nature du problème, et lui demander de recommencer son entrée. Les "**suite_…**" rappelle donc les prédicats auxquels elles sont rattachés en cas d'erreur.

PARTIE 3 - Démonstration de la proposition

Cette partie consiste à faire la démonstration de la proposition donnée par l'utilisateur.

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

Il va tout d'abord falloir trier la Abox, on va parcourir la Abox et mettre chaque élément dans la liste respective qui correspond à son type d'assertion : une liste contenant toutes les assertions du type $I : \exists R.C$, une pour les assertions $I : \forall R.C$, une pour les assertions $I : C1 \sqcap C2$, une pour les assertions $I : C1 \sqcup C2$ et enfin la dernière qui contient que des assertions du type $I : C$ ou $I : \neg C$ ou C est un concept atomique (**on appellera respectivement les listes : Lie, Lpt, Li, Lu et Ls**).

La Abox utilisée ici est la suivante (pour les listes non vide):

$Li = [\,(steph : personne \sqcap \exists aEnfant.anything\ )\,]$
$Ls = [\,(david : sculpture)\,,\ (joconde : objet),\ (michelAnge : personne),$
$(sonnets : livre),\ (vinci : personne)\,]$

résolution(Lie, Lpt, Li, Lu, Ls, Abr) et isclash(Ls)

Le prédicat résolution est ici le point d'entrée de la démonstration, on utilise un prédicat intermédiaire **isclash(Ls)** qui permettra de vérifier si un élément et sa négation se trouve dans la Abox (la liste Ls). **isclash** fera un parcours de la liste **Ls** et vérifiera pour chaque élément si sa négation se trouve dans la liste en ayant retiré l'élément (on fait la négation de l'élément et on la met sous forme normale négative). Lorsqu'un clash est trouvé, la résolution de la branche est terminée, il faudra vérifier ensuite les autres branches s'il y en a d'autres (ce n'est que le cas lorsqu'on a fait des **transformation_or**). Le prédicat essayera d'appliquer l'une des quatres règles si on ne trouve pas de clash (d'où le ";", si le clash est vrai on ne va pas essayer d'aller plus loin), l'ordre choisi (dans l'énoncé) pour les règles est : **complete_some, transformation_and, deduction_all, transformation_or**.

complete_some(Lie, Lpt, Li, Lu, Ls, Abr),
transformation_and(Lie, Lpt, Li, Lu, Ls, Abr),
deduction_all(Lie, Lpt, Li, Lu, Ls, Abr),
transformation_or(Lie, Lpt, Li, Lu, Ls, Abr) ,
evolue(A, Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1, Li1, Lu1, Ls1) et
evolue_all(Tmp, Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1, Li1, Lu1, Ls1)

On notera ici que tous les ajouts d'éléments se feront par le prédicat **evolue** qui ajoutera selon le type d'élément, la nouvelle assertion dans l'une des listes Lie1, Lpt1, Li1, Lu1, Ls1 **(ou evolue_all spécifiquement pour deduction_all)**.

Si la liste Lie n'est pas vide, **complete_some** va générer une instance B à partir de **genere** et ajouter dans **Ls** l'élément $B : C$. Une nouvelle branche de résolution (appel à **resolution** avec les nouvelles listes) sera créée.

Sinon, on appellera **transformation_and**, qui si **Li** est non vide ajoutera dans **Ls** l'élément $I : C1$ et l'élément $I : C2$. Une nouvelle branche de résolution sera créée.

Sinon, on appellera **deduction_all** , qui si **Lpt** est non vide ajoutera dans Ls, tous les éléments $Y : C$ avec chaque Y qui vérifie $< A, Y > : R$. Ce prédicat appellera un autre prédicat intermédiaire qu'on a appelé **evolue_all** puisqu'on doit ajouter tous les éléments possibles afin de pouvoir retirer l'élément de la liste **Lpt**. Une nouvelle branche de résolution ici aussi sera créée (*Le Tmp de evolue_all correspond à la liste contenant tous les éléments du type $Y : C$ à ajouter en fonction du type d'élément trouvé tous les éléments seront ajoutés à sa liste respective*).

Sinon, on appellera **transformation_or**, qui va ajouter dans 2 listes différentes de **Ls**, respectivement les éléments $I : C1$ et $I : C2$, et créera 2 nouvelles branches de résolution. Il faudra donc que la branche 1 et 2 donne un clash pour démontrer la proposition.
Si aucune des règles n'est appliquée, alors on a soit réussi à démontrer la proposition (**isclash** a renvoyé vrai), soit pas réussi.

affiche_evolution_Abox(Ls1, Lie1, Lpt1, Li1, Lu1, Abr1, Ls2, Lie2, Lpt2, Li2, Lu2, Abr2), afficheAbox(L), afficheAbox(A)

On affichera aussi entre chaque changement des listes, l'évolution à partir du prédicat **affiche_evolution_Abox**, qui affichera les assertions avec des symboles mathématiques. Le prédicat intermédiaire **afficheAbox** permet de gérer l'affichage des différents éléments un à un (ce sera plus facile de cette manière), pour chaque type d'expression si c'est un "et" ou "ou", on décompose les 2 concepts qu'on appel aussi avec **afficheAbox** et on ajoute le symbole ⊓ ou ⊔, pour les "pour tout" et "il existe", on sépare le rôle du concept qu'on affiche tous les 2 avec **afficheAbox** en affichant au début ∃ ou ∀.

Enfin, pour pouvoir tester le démonstrateur, on peut utiliser des propositions comme $michelAnge : personne$ ou $michelAnge : sculpteur$ pour les propositions de type I et $auteur \sqcap editeur$ pour les propositions de type II (elles renvoient toutes vrai).

$michelAnge : personne$ sera démontrée par un clash avec : $michelAnge : \neg personne$.
$michelAnge : sculpteur$ sera démontrée par un clash avec :
$michelAnge : personne$ et sa négation ainsi que $david : sculpture$ et sa négation (on a ici une disjonction, il faut donc vérifier les 2 branches de résolution).
$auteur \sqcap editeur$ donnera un clash entre une instance: $livre$ et cette même instance: $\neg livre$ (n'est pas un livre).

Des propositions telles que $michelAnge : parent$ ou $personne \sqcap livre$ renvoie faux puisqu'elles ne sont pas démontrables (en développant le tout, aucun élément ne devrait être en clash).