Faculté des Sciences et Ingénierie - Sorbonne Université
Master Informatique parcours ANDROIDE

LRC - Logique et représentations des connaissances

Rapport de projet

Subsomptions en Prolog

Réalisé par :

PINHO FERNANDES Enzo

Décembre 2024

Table des matières

Introduction

Le but de ce projet est de programmer en Prolog un algorithme permettant d'inférer des subsomptions pour la logique de description FL^- décrite à la fin du sujet. On procède de façon progressive en ajoutant des règles pour gérer des cas de plus en plus complexes. On s'assure toujours de faire des inférences correctes : si on trouve que C subsume D selon nos règles, on a la garantie que C subsume bien D. L'inverse n'est pas forcément vrai, le programme, construit progressivement, pouvant être incomplet.

Le rendu du projet doit comporter deux fichiers :

— un fichier prolog, comportant en première ligne, en commentaire, les prénoms et noms des deux membres du binôme,

— un fichier pdf, comportant également en première ligne les prénoms et noms des deux membres du binôme et donnant les réponses aux questions de commentaire et logique.

Représentation

Exercice 1 : Représentation préfixe en prolog

On traduit les différents opérateurs de concepts et éléments des T-Box et A-Box de la façon suivante :

	$\mathcal{FL}^-$	prolog
Conjonction	$C \sqcap D$	`and(C, D)`
Quantification existentielle	$\exists R$	`some(R)`
Quantification universelle	$\forall R.C$	`all(R,C)`
Subsomption	$^T\boxed{C \sqsubseteq D}$	`subs(C,D)`
Equivalence	$^T\boxed{C \equiv D}$	`equiv(C,D)`

On travaille sur la T-Box correspondant aux connaissances sur les animaux suivant :

> Les chats sont des félins, comme les lions, alors que les chiens sont des canidés. Les seuls canidés considérés sont les chiens. Souris, félins et canidés sont des mammifères. Les mammifères sont des animaux, de même que les canaris. Les animaux sont des êtres vivants. On ne peut être à la fois animal et plante.
> Un animal qui a un maître est un animal de compagnie. Un animal de compagnie a forcément un maître, et toute entité qui a un maître ne peut avoir qu'un (ou plusieurs) maître(s) humain(s). Un chihuahua est à la fois un chien et un animal de compagnie.
> Un carnivore exclusif est défini comme une entité qui mange uniquement des animaux, de même, un herbivore exclusif est une entité qui mange uniquement des plantes. Le lion est un carnivore exclusif. On considère que tout carnivore exclusif est un prédateur. Tout animal se nourrit. On ne peut pas à la fois ne rien manger (ne manger que des choses qui n'existent pas) et manger quelque chose.

La traduction de ces connaissances en prolog est donnée dans le fichier LRC_donneesProjet.pl. Ouvrir ce fichier pour inspecter les données. Traduire en formules de FL^- les 6 lignes associées à l'indication **/* à commenter */** et identifier les phrases qu'elles traduisent.
Note : $\bot$ n'existant pas en FL^-, on traite ici **nothing** comme un concept atomique.

Réponse à l'exercice 1 - 1/1

Voici la traduction en FL^- des 6 lignes à commenter du fichier LRC_donneesProjet.pl ainsi que l'identification des phrases qu'elles traduisent dans la description des animaux vu précédemment.

Code prolog **Traduction** FL^- **Identification de la phrase**
`subs(chat,felin).` $chat \sqsubseteq felin$ "Les chats sont des félins"
`subs(chihuahua,and(chien,pet)).` $chihuahua \sqsubseteq (chien \sqcap pet)$ "Un chihuahua est à la fois un chien et un animal de compagnie."
`subs(and(animal,some(aMaitre)),pet).` $(animal \sqcap \exists aMaitre) \sqsubseteq pet$ "Un animal qui a un maître est un animal de compagnie."
`subs(some(aMaitre),all(aMaitre,personne)).` $\exists aMaitre \sqsubseteq \forall aMaitre.personne$ "Toute entité qui a un maître ne peut avoir qu'un (ou plusieurs) maître(s) humain(s)."
`subs(and(all(mange,nothing),some(mange)),nothing).` $(\forall mange.nothing \sqcap \exists mange) \sqsubseteq nothing$ "On ne peut pas à la fois ne rien manger (ne manger que des choses qui n'existent pas) et manger quelque chose."
`equiv(carnivoreExc,all(mange,animal)).` $carnivoreExc \equiv \forall mange.animal$ "Un carnivore exclusif est défini comme une entité qui mange uniquement des animaux."

Subsomption structurelle pour FL^-

Le but de cette section est d'écrire un ensemble de règles permettant de répondre à des questions du type "est-ce qu'on peut prouver que C est subsumé par D ?" (soit "est-ce que $C \sqsubseteq_s D$?")

Il faut souligner la différentre entre $\sqsubseteq$, qui correspond à des axiomes d'inclusion fournis explicitement dans la TBox, et $\sqsubseteq_s$, qui correspond à des subsomptions que l'on peut prouver, à partir des axiomes fournis dans la TBox.

Exercice 2 : Concepts atomiques

On suppose que dans un premier temps que la TBox ne contient que des expressions du type $A \sqsubseteq B$ où A et B sont des concepts atomiques et que C et D sont aussi atomiques. Pour vérifier si $C \sqsubseteq_s D$ dans ce contexte, on propose les règles suivantes, à écrire dans votre fichier prolog, dans lequel vous aurez préalablement copié-collé le contenu du fichier LRC_donneesProjet.pl.

```
subsS1(C,C).
subsS1(C,D) :- subs(C,D), C\==D.
subsS1(C,D) :- subs(C,E), subsS1(E,D).
```

2.1) Que traduisent ces règles ? Les tester sur les requêtes $canari \sqsubseteq_s animal$, $chat \sqsubseteq_s etreVivant$.

Réponse à la question 2.1 - 1/1

Traduction des règles :

- `subsS1(C,C)` :
  - Un concept est toujours subsumé par lui-même, on a donc $C \sqsubseteq_s C$.
- `subsS1(C,D) :- subs(C,D), C\==D` :
  - Si, dans la TBox, un concept C est subsumé par un concept D différent, alors on a $C \sqsubseteq_s D$.
- `subsS1(C,D) :- subs(C,E), subsS1(E,D)` :
  - On peut y voir la règle de la transitivité. Si $C \sqsubseteq E$, et $E \sqsubseteq_s D$, alors on a $C \sqsubseteq D$.

Les requêtes renvoient true, ce qui est logique pour la première étant donné qu'on a bien $canari \sqsubseteq animal$. Pour la deuxième, il existe une transition qui part de *chat* et qui arrive à $etreVivant$, la voici : $(chat \sqsubseteq felin) \rightarrow (felin \sqsubseteq mammifere) \rightarrow (mammifere \sqsubseteq animal) \rightarrow (animal \sqsubseteq etreVivant)$.

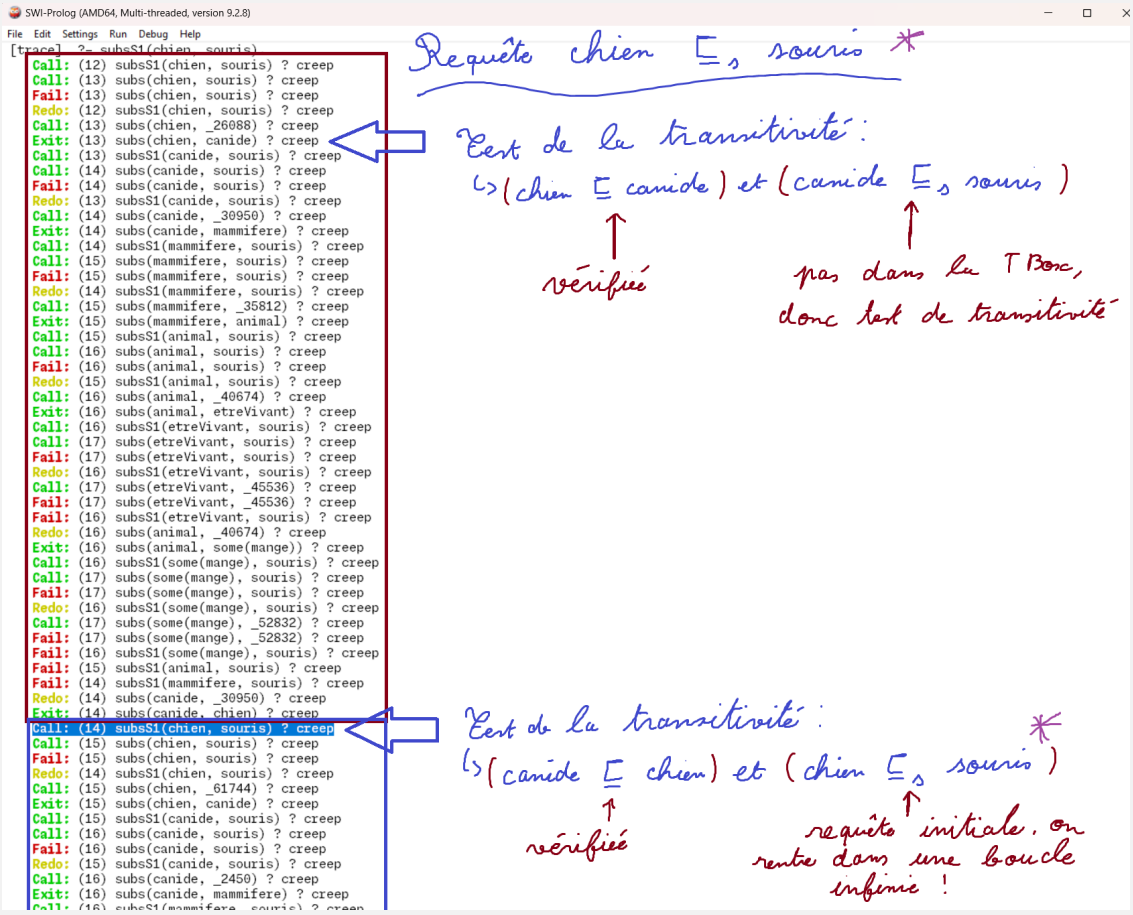
2.2) Tester la requête *chien* $\sqsubseteq_s$ *souris*. Quel problème pose cette requête ? Utiliser la trace pour en identifier la cause.

Réponse à la question 2.2 - 1/1

La requête *chien* $\sqsubseteq_s$ *souris* ne se termine jamais, dû à une boucle infinie au niveau de la règle de la transition. En effet, étant donné que *chien* et *souris* ne sont pas les mêmes concepts, et que nous avons pas *chien* $\sqsubseteq$ *souris*, alors le seul moyen est de trouver une transition de subsomption possible entre les deux.

Pour cela, il essayera de commencer par la subsomption *chien* $\sqsubseteq$ *canide* et cherchera donc à vérifier *canide* $\sqsubseteq$ *souris*. Même constat pour celui-ci. Il essayera plusieurs transitions qui rateront, jusqu'à qu'il fasse une transition *canide* $\sqsubseteq$ *chien*... Et il cherchera à vérifier la requête initiale *chien* $\sqsubseteq_s$ *souris*. Cela crée une boucle infinie.

Voici l'exécution de la requête avec trace, et quelques commentaires dessus pointant le problème.



Pour corriger ce problème, on introduit un troisième argument contenant la liste des subsomptions déjà faites dans une branche. On définit `subsS(C,D)`, qui doit être vérifié quand $C \sqsubseteq_s D$, avec le prédicat auxiliaire `subsS(C,D,L)` où L contient la liste des concepts utilisés dans la preuve de la subsomption. Pour éviter des preuves infinies, on s'interdit de réutiliser un concept déjà présent dans L.

On réécrit donc les règles sur `subsS1(C,D)` pour définir `subsS(C,D,L)`, en rajoutant avant tout appel récursif $E \sqsubseteq_s D$ la condition que E ne soit pas dans L et en ajoutant E à L dans l'appel récursif :

```
subsS(C,D) :- subsS(C,D,[C]).
subsS(C,C,_).
subsS(C,D,_) :- subs(C,D), C \== D
subsS(C,D,L) : -subs(C,E), not(member(E,L)), subsS(E,D,[E|L]), E \== D.
```

2.3) Tester ces nouvelles règles avec *chat* $\sqsubseteq_s$ *etreVivant*, *chien* $\sqsubseteq_s$ *canide* et *chien* $\sqsubseteq_s$ *chien* et vérifier que le résultat est conforme aux attentes. Tester ensuite la requête *chien* $\sqsubseteq_s$ *souris* qui doit échouer. Inspecter la trace de cette requête.

Réponse à la question 2.3 - 1/1

Les requêtes *chat* $\sqsubseteq_s$ *etreVivant*, *chien* $\sqsubseteq_s$ *canide* et *chien* $\sqsubseteq_s$ *chien* renvoient tous true, ce qui est conforme à nos attentes. En ce qui concerne la requête *chien* $\sqsubseteq_s$ *souris*, elle échoue bien comme attendu. Avec la trace, nous pouvons constater qu'à chaque transition testée, on vérifie le contenu de la liste des concepts utilisés. Et celui-ci empêche bien la boucle infinie dû à la transition de *canide* et *chien*.

Voici l'exécution d'un bout de requête avec trace, sur la solution du problème de la boucle infinie.

```
   Exit: (17) subs(animal, some(mange)) ? creep
^  Call: (17) not(member(some(mange), [animal, mammifere, canide, chien])) ? creep
^  Exit: (17) not(user:member(some(mange), [animal, mammifere, canide, chien])) ? creep
   Call: (17) subsS(some(mange), souris, [some(mange), animal, mammifere, canide, chien]) ? creep
   Call: (18) subs(some(mange), souris) ? creep
   Fail: (18) subs(some(mange), souris) ? creep
   Redo: (17) subsS(some(mange), souris, [some(mange), animal, mammifere, canide, chien]) ? creep
   Call: (18) subs(some(mange), _7816) ? creep
   Fail: (18) subs(some(mange), _7816) ? creep
   Fail: (17) subsS(some(mange), souris, [some(mange), animal, mammifere, canide, chien]) ? creep
   Fail: (16) subsS(animal, souris, [animal, mammifere, canide, chien]) ? creep
   Fail: (15) subsS(mammifere, souris, [mammifere, canide, chien]) ? creep
   Redo: (15) subs(canide, _116) ? creep
   Exit: (15) subs(canide, chien) ? creep
^  Call: (15) not(member(chien, [canide, chien])) ? creep
^  Fail: (15) not(user:member(chien, [canide, chien])) ? creep
   Fail: (14) subsS(canide, souris, [canide, chien]) ? creep
   Fail: (13) subsS(chien, souris, [chien]) ? creep
   Fail: (12) subsS(chien, souris) ? creep
false.
```

chien a déjà été utilisé, donc on coupe le test de la transition canide -> chien -> souris

2.4) Tester la requête $souris \sqsubseteq_s \exists mange$. Pourquoi cette requête réussit-elle bien que $\exists mange$ ne soit pas un concept atomique ?

Réponse à la question 2.4 - 1/1

Cette requête réussit bien que $\exists mange$ n'est pas un concept atomique parce qu'il existe techniquement une transition de *souris* à $some(mange)$ d'après notre TBox sur prolog. Ce dernier ne fait pas de "distinction" entre concept atomique et autre, et il trouve bien une valeur égale à $some(mange)$.

En effet, on peut suivre les étapes suivantes :

- `subsS(souris, some(mange)) :- subs(souris, mammifere), subsS(mammifere, some(mange))`
- `subs(souris, mammifere)` est vérifiée.
- `subsS(mammifere, some(mange)) :- subs(mammifere, animal), subsS(animal, some(mange))`
- `subs(mammifere, animal)` est vérifiée.
- `subsS(animal, some(mange))` est vérifiée grâce à la troisième règle, on trouve bien `subs(animal, some(mange))`.

2.5) Que devraient renvoyer les requêtes $chat \sqsubseteq_s X$ et $X \sqsubseteq_s mammifere$? Vérifier que c'est bien le cas (il n'est pas nécessaire d'éliminer les réponses doubles ou le false final).

Réponse à la question 2.5 - 1/1

La requête $chat \sqsubseteq_s X$ est sensée renvoyer "tout ce qu'un chat puisse être" par la variable X d'après notre TBox. Dans ce cas là, nous devrions avoir plusieurs résultats pour X. Un chat est un **chat** (subsumé par lui-même), un **félin**, un **mammifere**, un **animal**, un **être vivant** et pour la même raison qu'à la question 2.4, il devrait renvoyer **some(mange)**, parce qu'un animal mange toujours au moins une chose et que prolog le renvoie.

La requête $X \sqsubseteq_s mammifere$ est sensée renvoyer "toute entité étant une souris", dans notre TBox, il n'y a que la **souris** qui peut être renvoyée, parce que tout concept est subsumé par lui-même. On ne peut pas trouver une quelconque transition de subsomption entre X et *souris*.

Voici la vérification de notre réponse sur prolog, on a bien les mêmes résultats, ainsi qu'un false final, parce que Prolog teste d'autres possibilités même après avoir trouvé toutes les solutions réalisables.

```
?- subsS(chat, X).
X = chat ;
X = felin ;
X = mammifere ;
X = animal ;
X = etreVivant ;
X = some(mange) ;
false.
```

```
?- subsS(X, souris).
X = souris ;
false.
```

2.6) Ecrire des règles permettant de dériver $A \sqsubseteq B$ et $B \sqsubseteq A$ à partir de $A \equiv B$. Tester avant et après ajout des règles la requête $lion \sqsubseteq_s \forall mange.animal$.

Réponse à la question 2.6 - 1/1

Pour commencer, testons la requête avant l'ajout des règles. Prolog nous renvoie tout simplement **false**. En effet, nous avons aucune subsomption dans la TBox de type $subs(X, all(mange, animal))$, X étant une variable. On ne peut donc pas trouver de transition, et on ne peut pas utiliser l'équivalence $carnivoreExc \equiv \forall mange.animal$ dû à l'absence des règles.

Afin d'ajouter les règles, il faut savoir que $C \equiv D \Rightarrow (C \sqsubseteq D) \wedge (D \sqsubseteq C)$. Il suffit donc de rajouter deux nouvelles règles cherchant si $C \equiv D$ est dans la TBox, pour $C \sqsubseteq D$ et $D \sqsubseteq C$ séparémment. On a donc :

```
subsS(C, D, _) :- equiv(C, D).
subsS(D, C, _) :- equiv(C, D).
```

L'ajout se trouve aux lignes 61-63 du fichier LRC_donneesProjet.pl.

Avec les nouvelles règles, prolog nous renvoie **true**. En effet, il va en premier lieu tester plusieurs transitions de subsomption jusqu'à tomber sur celle allant de $lion$ à $carnivoreExc$. Ensuite, il essayera donc de prouver que $subsS(carnivoreExc, all(mange, animal))$ ce qu'il y arrivera grâce à l'équivalence $equiv(carnivoreExc, all(mange, animal))$.

Ensuite, il cherchera toujours d'autres chemins possible, sans résultat, renvoyant un false.