LRC - Projet M1 AI2D
Subsomptions en Prolog
Année 2025–2026

Yuxiang ZHANG Kenan ALSAFAD

Introduction

Le but de ce projet est de programmer en `Prolog` un algorithme permettant d'inférer des subsomptions, c'est-à-dire de déterminer si $C \sqsubseteq D$ où C et D sont des concepts, dans la logique de description FL^-. Cette dernière est une logique moins expressive que ALC et se caractérise par des concepts construits à partir de concepts atomiques A et de rôles atomiques R selon la grammaire suivante :

$$C ::= A \mid C \sqcap C \mid \exists R \mid \forall R.C$$

Sa sémantique repose sur une structure $\mathcal{M} = (\Delta^{\mathcal{M}}, \cdot^{\mathcal{M}})$, où les concepts sont interprétés ainsi :

$$\begin{cases} (C_1 \sqcap C_2)^{\mathcal{M}} = C_1^{\mathcal{M}} \cap C_2^{\mathcal{M}} \\ (\exists R)^{\mathcal{M}} = \{x \in \Delta^{\mathcal{M}} \mid \exists y \in \Delta^{\mathcal{M}}, (x,y) \in R^{\mathcal{M}}\} \\ (\forall R.C)^{\mathcal{M}} = \{x \in \Delta^{\mathcal{M}} \mid \forall y \in \Delta^{\mathcal{M}}, (x,y) \in R^{\mathcal{M}} \Rightarrow y \in C^{\mathcal{M}}\} \end{cases}$$

Note : $\bot$ n'existant pas dans FL^-, on traite ici `nothing` comme un concept atomique.

Nous développons notre programme de manière progressive en ajoutant des règles pour traiter des cas de plus en plus complexes. Nous veillons à garantir la correction des inférences : si notre algorithme affirme que C subsume D, alors cette subsomption est bien valide. En revanche, l'algorithme peut être incomplet, et ne pas détecter toutes les subsomptions possibles.

Représentation

Exercice 1 – Représentation préfixe en Prolog

Nous traduisons les différents opérateurs de concepts et éléments des *T-Box* et *A-Box* de la façon suivante :

FL$^-$	**Prolog**
Conjonction $C \sqcap D$	`and(C, D)`
Quantification existentielle $\exists R$	`some(R)`
Quantification universelle $\forall R.C$	`all(R, C)`
Subsomption $C \sqsubseteq D$	`subs(C, D)`
Equivalence $C \equiv D$	`equiv(C, D)`

Nous travaillons sur la *T-Box* correspondant aux connaissances sur les animaux suivantes :

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`.

Traduction des lignes « à commenter » en formules FL$^-$ et identification des phrases correspondantes

Les six lignes du fichier Prolog associées à l'indication `/* à commenter */` se traduisent en formules de la logique FL$^-$ comme suit :

— `subs(chat, felin).`

$$\text{chat} \sqsubseteq \text{félin}$$

Les chats sont des félins.

— `subs(chihuahua, and(chien, pet)).`

$$\text{chihuahua} \sqsubseteq \text{chien} \sqcap \text{pet}$$

Un chihuahua est à la fois un chien et un animal de compagnie.

— `subs(and(animal, some(aMaitre)), pet).`

$$\text{animal} \sqcap \exists aMaitre \sqsubseteq \text{pet}$$

Un animal qui a un maître est un animal de compagnie.

— `subs(some(aMaitre), all(aMaitre, personne)).`

$$\exists aMaitre \sqsubseteq \forall aMaitre.\text{personne}$$

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).`

$$\forall mange.\bot \sqcap \exists mange \sqsubseteq \bot$$

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)).`

$$\text{carnivoreExc} \equiv \forall mange.\text{animal}$$

Un carnivore exclusif est défini comme une entité qui mange uniquement des animaux.

Exercice 2 : Concepts atomiques

On suppose dans un premier temps que la T-Box ne contient que des expressions de la forme

$$A \sqsubseteq B$$

où A et B sont des concepts atomiques, et que C et D sont également atomiques. Pour vérifier si $C \sqsubseteq_s D$ dans ce contexte, on propose les règles suivantes :

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

Question 2.1 : Que traduisent ces règles ?

Ces règles définissent la relation réflexive et transitive `subsS1` à partir des faits `subs` contenus dans la T-Box.

- La première règle affirme que tout concept subsume lui-même (réflexivité).
- La deuxième règle dit que si `subs(C,D)` est vrai et $C \neq D$, alors `subsS1(C,D)` est vrai.
- La troisième règle permet d'enchaîner plusieurs subsomptions pour obtenir la transitivité : si C subsume E et E subsume D, alors C subsume D.

Question 2.2 : Testez-les sur les requêtes suivantes

```
?- subsS1(canari, animal).
true ;
true ;
false.

?- subsS1(canari, etreVivant).
true ;
true ;
false.
```

Les résultats des tests montrent que `canari` subsume bien `animal` et `etreVivant`, conformément aux faits de la T-Box.

Analyse de la requête `chien` $\sqsubseteq_s$ `souris`

La requête `?- subsS1(chien, souris).` pose problème car elle entre dans une boucle récursive infinie. Une analyse de la trace d'exécution révèle la cause exacte du problème.

Chemin d'inférence observé dans la trace :

```
subsS1(chien, souris)
  subs(chien, canide)
    subsS1(canide, souris)
      subs(canide, mammifere)
        subsS1(mammifere, souris)
          subs(mammifere, animal)
```

```
        subsS1(animal, souris)
          subs(animal, etreVivant)
            subsS1(etreVivant, souris)
              subs(etreVivant, _) → échec
            subs(animal, some(mange))
              subsS1(some(mange), souris) → échec
    subs(canide, chien)
      subsS1(chien, souris) [cycle]
```

On constate qu'à la fin de cette chaîne d'appels, le programme revient à la requête initiale `subsS1(chien, souris)`. Cela crée une boucle entre les concepts `chien` et `canide` :

$$\texttt{chien} \rightarrow \texttt{canide} \rightarrow \texttt{chien} \rightarrow \ldots$$

Cause du problème : Ce cycle est causé par la définition bidirectionnelle suivante dans la T-Box :

```
subs(chien, canide).
subs(canide, chien).
```

Cette double inclusion crée un cycle dans le graphe d'inclusion des concepts. La règle récursive :

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

entraîne alors une exploration sans fin du même chemin.

Solution avec prévention des cycles

Pour corriger le problème de boucle infinie, nous introduisons un troisième argument contenant la liste des concepts déjà visités dans la branche courante. Cette approche permet de détecter et éviter les cycles dans le graphe de subsomption.

Nouvelles règles avec détection de cycles

```
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.
```

Question 2.3 : Tests et résultats

Nous avons testé les nouvelles règles avec plusieurs requêtes pour vérifier leur comportement :

Test 1 : `chat` $\sqsubseteq_s$ `etreVivant`

```
?- subsS(chat, etreVivant).
true ;
false.
```

Résultat : SUCCÈS. La requête réussit car il existe un chemin valide sans cycle :

$$\text{chat} \rightarrow \text{félin} \rightarrow \text{mammifère} \rightarrow \text{animal} \rightarrow \text{êtreVivant}$$

Test 2 : `chien` $\sqsubseteq_s$ `canide`

```
?- subsS(chien, canide).
true ;
false.
```

Résultat : SUCCÈS. La subsomption directe est trouvée sans problème.

Test 3 : `chien` $\sqsubseteq_s$ `chien`

```
?- subsS(chien, chien).
true ;
true ;
false.
```

Résultat : SUCCÈS. La règle de réflexivité fonctionne correctement.

Test 4 : `chien` $\sqsubseteq_s$ `souris`

```
?- subsS(chien, souris).
false.
```

Résultat : ÉCHEC (comme attendu). La requête échoue proprement sans boucle infinie.

Test de la requête `chien` $\sqsubseteq_s$ `souris` et analyse de la trace

La requête `chien` $\sqsubseteq_s$ `souris` a été testée et a échoué comme attendu. L'analyse détaillée de la trace d'exécution révèle le fonctionnement du mécanisme de prévention des cycles :

```
Call: subsS(chien, souris, [chien])
  Fail: subs(chien, souris) → Échec direct
  Redo: Recherche d'un chemin indirect
  Exit: subs(chien, canide) → Trouvé
  Call: not(member(canide, [chien])) → Réussi
  Call: subsS(canide, souris, [canide, chien])
    Fail: subs(canide, souris) → Échec direct
    Redo: Recherche d'un chemin indirect
    Exit: subs(canide, mammifere) → Trouvé
    Call: not(member(mammifere, [canide, chien])) → Réussi
    Call: subsS(mammifere, souris, [mammifere, canide, chien])
      Fail: subs(mammifere, souris) → Échec direct
      Redo: Recherche d'un chemin indirect
      Exit: subs(mammifere, animal) → Trouvé
      Call: not(member(animal, [mammifere, canide, chien])) → Réussi
      Call: subsS(animal, souris, [animal, mammifere, canide, chien])
        Fail: subs(animal, souris) → Échec direct
        Redo: Recherche d'un chemin indirect
        Exit: subs(animal, etreVivant) → Trouvé
        Call: not(member(etreVivant, [animal, mammifere, canide, chien])) → Réus
        Call: subsS(etreVivant, souris, [etreVivant, animal, mammifere, canide,
          Fail: subs(etreVivant, souris) → Échec direct
          Redo: Recherche d'autres chemins depuis etreVivant
          Fail: Aucun autre chemin → Échec complet
        Exit: subs(animal, some(mange)) → Autre chemin trouvé
        Call: not(member(some(mange), [animal, mammifere, canide, chien])) → Réu
        Call: subsS(some(mange), souris, [some(mange), animal, mammifere, canide
          Fail: subs(some(mange), souris) → Échec direct
          Redo: Recherche d'autres chemins depuis some(mange)
          Fail: Aucun autre chemin → Échec complet
    Redo: Recherche d'autres chemins depuis canide
    Exit: subs(canide, chien) → Cycle potentiel détecté !
    Call: not(member(chien, [canide, chien])) → ÉCHEC ! Cycle bloqué
```

Points clés de l'analyse :

— **Exploration exhaustive :** L'algorithme explore systématiquement tous les chemins possibles depuis `chien` vers `souris` :
  — Chemin 1 : `chien → canide → mammifere → animal → etreVivant` (échec)
  — Chemin 2 : `chien → canide → mammifere → animal → some(mange)` (échec)
  — Chemin 3 : `chien → canide → chien` (cycle détecté et bloqué)
— **Prévention des cycles :** Le moment crucial se produit lorsque le programme tente d'explorer le chemin `canide → chien`. La condition `not(member(chien, [canide, chien]))` échoue car `chien` est déjà présent dans la liste des concepts visités, ce qui empêche la récursion infinie.
— **Échec propre :** Aucun des chemins valides ne mène à `souris`, et les cycles sont correctement détectés et évités. La requête échoue donc proprement sans boucle infinie.

Question 2.4 : Test de la requête `souris` $\sqsubseteq_s$ ∃`mange`

La requête `souris` $\sqsubseteq_s$ ∃`mange` a été testée et a réussi, ce qui peut sembler surprenant étant donné que ∃mange n'est pas un concept atomique.

Analyse du succès de la requête

1. Statut de ∃R dans FL⁻ Dans la logique de description FL⁻, la quantification existentielle ∃R est un **concept légitime** selon la grammaire des concepts :

$$C ::= A \mid C \sqcap C \mid \exists R \mid \forall R.C$$

Bien que ∃mange ne soit pas atomique, il s'agit d'un **concept bien formé** dans le langage FL⁻.

2. Chaîne de subsomption dans la T-Box Notre T-Box contient les axiomes suivants qui établissent un chemin de subsomption :

```
subs(souris, mammifere).
subs(mammifere, animal).
subs(animal, some(mange)).
```

Par transitivité, nous obtenons :

$$\text{souris} \sqsubseteq \text{mammifere} \sqsubseteq \text{animal} \sqsubseteq \exists\text{mange}$$

3. Traitement par notre implémentation Prolog Notre programme traite `some(mange)` comme un **concept unique** dans les règles de subsomption. Les règles établies précédemment :

```
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]).
```

s'appliquent parfaitement à ce cas, car `some(mange)` est traité comme un terme Prolog quelconque dans les faits `subs/2`.

Question 2.5 : Requêtes `chat` $\sqsubseteq_s$ `X` et `X` $\sqsubseteq_s$ `mammifere`

Analyse théorique des résultats attendus

Requête `chat` $\sqsubseteq_s$ `X` Cette requête recherche tous les concepts X qui sont subsumés par `chat`. Selon la T-Box et les règles de subsomption, les résultats attendus sont :

- `X = chat` (par réflexivité)
- `X = felin` (subsomption directe : `chat` $\sqsubseteq$ `felin`)
- `X = mammifere` (transitivité : `chat` $\sqsubseteq$ `felin` $\sqsubseteq$ `mammifere`)
- `X = animal` (transitivité : `chat` $\sqsubseteq$ `felin` $\sqsubseteq$ `mammifere` $\sqsubseteq$ `animal`)
- `X = etreVivant` (transitivité : `chat` $\sqsubseteq$ ... $\sqsubseteq$ `animal` $\sqsubseteq$ `etreVivant`)
- `X = some(mange)` (transitivité : `chat` $\sqsubseteq$ ... $\sqsubseteq$ `animal` $\sqsubseteq$ `some(mange)`)

Requête `X` $\sqsubseteq_s$ `mammifere` Cette requête recherche tous les concepts X qui subsument `mammifere`. Les résultats attendus sont :

- `X = mammifere` (réflexivité)
- `X = felin` (subsomption directe : `felin` $\sqsubseteq$ `mammifere`)
- `X = canide` (subsomption directe : `canide` $\sqsubseteq$ `mammifere`)
- `X = souris` (subsomption directe : `souris` $\sqsubseteq$ `mammifere`)
- `X = chat` (transitivité : `chat` $\sqsubseteq$ `felin` $\sqsubseteq$ `mammifere`)
- `X = chien` (transitivité : `chien` $\sqsubseteq$ `canide` $\sqsubseteq$ `mammifere`)
- `X = lion` (transitivité : `lion` $\sqsubseteq$ `felin` $\sqsubseteq$ `mammifere`)

Vérification expérimentale

Résultats de la requête `chat` $\sqsubseteq_s$ `X`

```
?- subsS(chat,X).
X = chat ;
```

```
X = felin ;
X = mammifere ;
X = animal ;
X = etreVivant ;
X = some(mange) ;
false.
```

Résultats de la requête `X` $\sqsubseteq_s$ `mammifere`

```
?- subsS(X,mammifere).
X = mammifere ;
X = felin ;
X = canide ;
X = souris ;
X = chat ;
X = chien ;
X = lion ;
false.
```

Analyse des résultats

Correspondance avec les attentes Les résultats expérimentaux correspondent parfaitement aux prédictions théoriques pour les deux requêtes :

- Tous les concepts attendus sont présents dans les résultats
- L'ordre d'apparition peut varier mais n'affecte pas la correction
- La présence de `false` à la fin est normale et indique qu'il n'y a pas d'autres solutions

Remarque sur `chihuahua` Le concept `chihuahua` n'apparaît pas dans les résultats de `X` $\sqsubseteq_s$ `mammifere` bien qu'on puisse s'attendre à ce qu'il subsume `mammifere` via la chaîne `chihuahua` $\sqsubseteq$ `chien` $\sqsubseteq$ `canide` $\sqsubseteq$ `mammifere`. Ceci s'explique par le fait que `chihuahua` est défini comme un concept composite (`and(chien, pet)`) et que nos règles actuelles ne permettent pas encore de déduire `chihuahua` $\sqsubseteq$ `chien` à partir de `chihuahua` $\sqsubseteq$ `and(chien, pet)`. Cette limitation sera résolue dans l'exercice suivant sur la gestion des conjonctions.

Question 2.6 : Gestion des équivalences

Règles ajoutées pour les équivalences

Pour gérer les expressions de la forme $A \equiv B$ dans la T-Box, nous avons ajouté les règles suivantes :

```
subs(C, D) :- equiv(C, D).
subs(D, C) :- equiv(C, D).
```

Ces règles permettent de déduire $A \sqsubseteq B$ et $B \sqsubseteq A$ à partir de $A \equiv B$.

Test de la requête `lion` $\sqsubseteq_s$ $\forall$`mange.animal`

Contexte : La T-Box contient les axiomes suivants :

```
subs(lion, carnivoreExc).
equiv(carnivoreExc, all(mange, animal)).
```

Avant l'ajout des règles d'équivalence : La requête `lion` $\sqsubseteq_s$ `all(mange, animal)` échouait car le système ne pouvait pas faire le lien entre `carnivoreExc` et `all(mange, animal)` via l'équivalence.

Après l'ajout des règles d'équivalence :

```
?- subsS(lion, all(mange, animal)).
true ;
false.
```

La requête réussit maintenant grâce à la chaîne de subsomption :

$$\text{lion} \sqsubseteq \text{carnivoreExc} \equiv \forall \text{mange.animal}$$

Les nouvelles règles permettent de déduire `subs(carnivoreExc, all(mange, animal))` à partir de l'équivalence, complétant ainsi le chemin de subsomption.

Exercice 3 - Gestion des conjonctions

Introduction

Dans cet exercice, nous étendons notre système pour gérer les requêtes contenant des conjonctions de concepts. L'objectif est d'ajouter des règles permettant de traiter les concepts de la forme $C \sqcap D$ tout en conservant le même prédicat `subsS` pour assurer la continuité avec les règles précédentes.

Implémentation des règles pour les conjonctions

Nous avons intégré les règles fournies pour la gestion des intersections dans notre fichier Prolog, à la suite des règles précédentes :

```
% Règles pour gérer l'intersection
subsS(C, and(D1, D2), L) :- D1 \= D2, subsS(C, D1, L), subsS(C, D2, L).
subsS(C, D, L) :- subs(and(D1, D2), D), E = and(D1, D2),
                  not(member(E, L)), subsS(C, E, [E | L]), E \== C.
subsS(and(C, C), D, L) :- nonvar(C), subsS(C, D, [C | L]).
subsS(and(C1, C2), D, L) :- C1 \= C2, subsS(C1, D, [C1 | L]).
subsS(and(C1, C2), D, L) :- C1 \= C2, subsS(C2, D, [C2 | L]).
subsS(and(C1, C2), D, L) :- subs(C1, E1), E = and(E1, C2),
                            not(member(E, L)), subsS(E, D, [E | L]), E \== D.
subsS(and(C1, C2), D, L) :- Cinv = and(C2, C1),
                            not(member(Cinv, L)), subsS(Cinv, D, [Cinv | L]).
```

Question 3.1 : Tests des requêtes avec conjonctions

Test 1 : `chihuahua` $\sqsubseteq_s$ `mammifere` $\sqcap$ $\exists$`aMaitre`

Résultat : SUCCÈS
Justification :
- La T-Box contient : `subs(chihuahua, and(chien, pet))`
- Nous pouvons déduire :
  - `chihuahua` $\sqsubseteq$ `chien` (première partie de la conjonction)
  - `chihuahua` $\sqsubseteq$ `pet` (deuxième partie de la conjonction)
- De `chien` $\sqsubseteq$ `canide` $\sqsubseteq$ `mammifere`, nous avons `chihuahua` $\sqsubseteq$ `mammifere`
- De `pet` $\sqsubseteq$ `some(aMaitre)`, nous avons `chihuahua` $\sqsubseteq$ `some(aMaitre)`
- Donc `chihuahua` $\sqsubseteq$ `mammifere` $\sqcap$ `some(aMaitre)`

Test 2 : `chien` $\sqcap$ $\exists$`aMaitre` $\sqsubseteq_s$ `pet`

Résultat : SUCCÈS
Justification :
- La T-Box contient : `subs(and(animal, some(aMaitre)), pet)`
- Nous avons `chien` $\sqsubseteq$ `animal` (via `chien` $\sqsubseteq$ `canide` $\sqsubseteq$ `mammifere` $\sqsubseteq$ `animal`)
- Donc `and(chien, some(aMaitre))` $\sqsubseteq$ `and(animal, some(aMaitre))` $\sqsubseteq$ `pet`
- La règle 2 permet d'utiliser cette subsomption existante

Test 3 : `chihuahua` $\sqsubseteq_s$ `pet` $\sqcap$ `chien`

Résultat : SUCCÈS
Justification :
— Directement de la T-Box : `subs(chihuahua, and(chien, pet))`
— Ce qui équivaut à `chihuahua` $\sqsubseteq$ `chien` $\sqcap$ `pet`
— Par commutativité de la conjonction : `chihuahua` $\sqsubseteq$ `pet` $\sqcap$ `chien`
— La règle 7 gère la commutativité des conjonctions

Observation sur les résultats multiples

Lors des tests, nous avons observé que toutes les trois requêtes retournent plusieurs fois `true` au lieu d'un seul `true` suivi de `false`. Ce phénomène s'explique par le mécanisme de **backtracking** de Prolog.

Par exemple, pour la requête `chihuahua` $\sqsubseteq_s$ `pet` $\sqcap$ `chien`, le système peut trouver plusieurs chemins de preuve :
— **Chemin direct** : Utilisation de `subs(chihuahua, and(chien, pet))`
— **Chemin par décomposition** : Preuve séparée de `chihuahua` $\sqsubseteq$ `pet` et `chihuahua` $\sqsubseteq$ `chien`
— **Chemin par commutativité** : Application de la règle d'échange des conjunctes

Chaque chemin constitue une preuve valide, donc Prolog retourne un `true` pour chaque preuve trouvée.

Question 3.2 : Analyse détaillée des règles de conjonction

Règle 1 : `subsS(C, and(D1, D2), L)`

Situation traitée : Vérifie si C est subsumé par D1 et D2 simultanément.
Exemple d'échec sans cette règle :
`chihuahua` $\sqsubseteq_s$ `pet` $\sqcap$ `chien` échouerait, car le système ne saurait pas décomposer la conjonction du côté droit.
Utilité : Cette règle capture la sémantique fondamentale de la conjonction : être subsumé par une conjonction signifie être subsumé par chacun de ses conjunctes.

Règle 2 : `subsS(C, D, L) :- subs(and(D1, D2), D), ...`

Situation traitée : Utilise une subsomption existante où la prémisse est une conjonction.
Exemple d'échec sans cette règle :

`and(chien, some(aMaitre))` $\sqsubseteq_s$ `pet` échouerait, car le système ne pourrait pas utiliser la connaissance `subs(and(animal, some(aMaitre)), pet)`.
Utilité : Permet d'exploiter les connaissances existantes de la T-Box qui impliquent des conjonctions.

Règle 3 : `subsS(and(C, C), D, L)`

Situation traitée : Gère le cas d'une conjonction idempotente (même concept des deux côtés).
Exemple d'échec sans cette règle :
Si nous avions `subs(and(animal, animal), etreVivant)` dans la T-Box, alors `and(animal, animal)` $\sqsubseteq_s$ `etreVivant` échouerait.
Utilité : Traite l'idempotence des conjonctions, une propriété fondamentale de l'opérateur $\sqcap$.

Règles 4 et 5 : `subsS(and(C1, C2), D, L)`

Situation traitée : Si une conjonction est subsumée par D, alors au moins un de ses conjunctes est subsumé par D.
Exemple d'échec sans ces règles :
`and(chien, some(aMaitre))` $\sqsubseteq_s$ `animal` échouerait, car bien que `chien` $\sqsubseteq$ `animal`, le système ne saurait pas extraire cette information de la conjonction.
Utilité : Ces règles capturent la propriété de "affaiblissement" (weakening) en logique : d'une conjonction, on peut déduire n'importe lequel de ses conjunctes.

Règle 6 : `subsS(and(C1, C2), D, L) :- subs(C1, E1), ...`

Situation traitée : Remplace un conjuncte par un concept plus général.
Exemple d'échec sans cette règle :
Si nous avions `subs(and(mammifere, some(aMaitre)), pet)` et `subs(chien, mammifere)`, alors `and(chien, some(aMaitre))` $\sqsubseteq_s$ `pet` échouerait.
Utilité : Permet de spécialiser progressivement les conjonctions en remplaçant un conjuncte par un concept plus spécifique.

Règle 7 : `subsS(and(C1, C2), D, L) :- Cinv = and(C2, C1), ...`

Situation traitée : Gère la commutativité des conjonctions.
Exemple d'échec sans cette règle :
`and(pet, chien)` $\sqsubseteq_s$ `and(chien, pet)` échouerait, bien que ces deux concepts soient logiquement équivalents.

Utilité : Capture la propriété de commutativité de l'opérateur $\sqcap$, essentielle pour une gestion complète des conjonctions.

Exercice 4 - Gestion des rôles

Question 4.1 : Règle pour les rôles qualifiés

Règle ajoutée

Nous avons ajouté la règle suivante pour gérer les subsomptions entre concepts de la forme $\forall R.C$ et $\forall R.D$:

```
subsS(all(R, C), all(R, D), L) :- subsS(C, D, L).
```

Justification sémantique

Cette règle repose sur la propriété fondamentale des quantificateurs universels en logique de description :

$$\text{Si } C \sqsubseteq D \text{ alors } \forall R.C \sqsubseteq \forall R.D$$

Cette propriété signifie que si un concept C est subsumé par un concept D, alors pour tout rôle R, le concept $\forall R.C$ (tous les R-successeurs sont dans C) est subsumé par $\forall R.D$ (tous les R-successeurs sont dans D).

Question 4.2 : Tests des requêtes avec rôles qualifiés

Test 1 : `lion` $\sqsubseteq_s$ $\forall$`mange.etreVivant`

Résultat obtenu :

```
?- subsS(lion, all(mange, etreVivant)).
true ;
false.
```

Analyse du succès :

— La T-Box contient les axiomes suivants :

```
subs(lion, carnivoreExc).
equiv(carnivoreExc, all(mange, animal)).
subs(animal, etreVivant).
```

— Chaîne de subsomption établie :

$$\text{lion} \sqsubseteq \text{carnivoreExc} \equiv \forall \text{mange.animal} \sqsubseteq \forall \text{mange.etreVivant}$$

— La règle pour les rôles qualifiés permet de déduire ∀mange.animal ⊑ ∀mange.etreVivant à partir de animal ⊑ etreVivant
— L'équivalence carnivoreExc ≡ ∀mange.animal est correctement exploitée via les règles d'équivalence

Interprétation sémantique : Ce résultat signifie que le lion, en tant que carnivore exclusif, ne mange que des êtres vivants, ce qui est sémantiquement correct.

Test 2 : ∀`mange.canari` $\sqsubseteq_s$ `carnivoreExc`

Résultat obtenu :

```
?- subsS(all(mange, canari), carnivoreExc).
false.
```

Analyse de l'échec :
Bien que la subsomption soit logiquement valide, notre implémentation actuelle ne parvient pas à la prouver. Examinons la chaîne de subsomption attendue :

$$\forall \text{mange.canari} \sqsubseteq \forall \text{mange.animal} \equiv \text{carnivoreExc}$$

— **Étape réussie :** La règle pour les rôles qualifiés permet de prouver ∀mange.canari ⊑ ∀mange.animal grâce à canari ⊑ animal
— **Étape problématique :** Le système ne parvient pas à établir le lien entre ∀mange.animal et carnivoreExc malgré l'équivalence carnivoreExc ≡ ∀mange.animal
— **Cause technique :** Notre implémentation utilise l'équivalence pour générer des subsomptions dans les deux directions (carnivoreExc ⊑ ∀mange.animal et ∀mange.animal ⊑ carnivoreExc), mais ces subsomptions sont ajoutées comme des faits `subs/2` directs. Le mécanisme de transitivité ne permet pas de combiner :
  — Une subsomption dérivée (entre concepts universellement quantifiés)
  — Avec une subsomption issue d'une équivalence

Solution potentielle : Pour résoudre cette limitation, nous pourrions ajouter des règles spécifiques gérant les interactions entre concepts quantifiés et concepts atomiques équivalents :

```
% Règles supplémentaires pour gérer les équivalences avec concepts quantifiés
subsS(all(R, C), Atomic, L) :-
    equiv(Atomic, all(R, D)),
```

```
    subsS(C, D, L).

subsS(Atomic, all(R, C), L) :-
    equiv(Atomic, all(R, D)),
    subsS(D, C, L).
```

Nouveaux résultats des tests

Test 1 : `lion` $\sqsubseteq_s$ $\forall$`mange.etreVivant`

```
?- subsS(lion, all(mange, etreVivant)).
true ;
true ;
false.
```

Analyse :

- La requête réussit maintenant avec plusieurs preuves, démontrant la richesse des capacités d'inférence
- Chaîne de subsomption principale :

$$\text{lion} \sqsubseteq \text{carnivoreExc} \equiv \forall \text{mange.animal} \sqsubseteq \forall \text{mange.etreVivant}$$

- Les résultats multiples indiquent que le système trouve plusieurs chemins de preuve valides

Test 2 : $\forall$`mange.canari` $\sqsubseteq_s$ `carnivoreExc`

```
?- subsS(all(mange, canari), carnivoreExc).
true ;
false.
```

Analyse :

- La requête réussit maintenant grâce aux nouvelles règles
- Chaîne de subsomption établie :

$$\forall \text{mange.canari} \sqsubseteq \forall \text{mange.animal} \equiv \text{carnivoreExc}$$

- Les nouvelles règles permettent de faire le lien entre le concept quantifié et le concept atomique équivalent

Explication du fonctionnement des nouvelles règles

Règle 1 : `subsS(all(R, C), Atomic, L)`
- **Situation traitée :** Subsomption d'un concept universellement quantifié vers un concept atomique équivalent
- **Application dans le test 2 :**
  - `all(mange, canari)` vers `carnivoreExc`
  - `equiv(carnivoreExc, all(mange, animal))` permet de faire le lien
  - `subsS(canari, animal, L)` prouve la relation entre les concepts intérieurs

Règle 2 : `subsS(Atomic, all(R, C), L)`
- **Situation traitée :** Subsomption d'un concept atomique équivalent vers un concept universellement quantifié
- **Utilité générale :** Permet de prouver des subsomptions dans l'autre direction

Question 4.3 : Tests des requêtes avec contradictions

Contexte logique

Avant de procéder aux tests, rappelons les définitions importantes de notre T-Box :
- `carnivoreExc` $\equiv$ $\forall$`mange.animal`
- `herbivoreExc` $\equiv$ $\forall$`mange.plante`
- `subs(and(animal,plante), nothing)` (animal et plante sont disjoints)
- `subs(and(all(mange,nothing), some(mange)), nothing)` (contradiction entre "ne rien manger" et "manger quelque chose")
- `subs(animal, some(mange))` (tout animal mange quelque chose)

Résultats initiaux sans règle supplémentaire

Avant l'ajout de la règle pour les restrictions universelles :

```
?- subsS(and(carnivoreExc, herbivoreExc), all(mange, nothing)).
false.

?- subsS(and(and(carnivoreExc, herbivoreExc), animal), nothing).
false.
```

Sans règle spécifique, les deux requêtes échouent car notre système ne parvient pas à déduire automatiquement que la conjonction de deux restrictions universelles sur le même rôle équivaut à une restriction universelle sur la conjonction des concepts.

Règle ajoutée et résultats après correction

Règle ajoutée :

```
% Règle pour la conjonction de restrictions universelles
subsS(and(all(R,C), all(R,D)), all(R, E), L) :-
    subsS(and(C,D), E, L).
```

Résultats après ajout de la règle :

```
?- subsS(and(carnivoreExc, herbivoreExc), all(mange, nothing)).
true ;
...

?- subsS(and(and(carnivoreExc, herbivoreExc), animal), nothing).
true ;
...
```

Les deux requêtes réussissent maintenant, produisant de multiples résultats `true`, ce qui indique que le système trouve plusieurs chemins de preuve valides pour chaque subsomption.

Explication du fonctionnement de la règle

La règle ajoutée capture la propriété logique fondamentale :

$$\forall R.C \sqcap \forall R.D \sqsubseteq \forall R.(C \sqcap D)$$

Chaîne de raisonnement pour le premier test :

1. Expansion des équivalences :

$$\text{carnivoreExc} \sqcap \text{herbivoreExc} \equiv \forall \text{mange.animal} \sqcap \forall \text{mange.plante}$$

2. Application de la nouvelle règle :

$$\forall \text{mange.animal} \sqcap \forall \text{mange.plante} \sqsubseteq \forall \text{mange.}(\text{animal} \sqcap \text{plante})$$

3. Utilisation de la T-Box :

$$\text{animal} \sqcap \text{plante} \sqsubseteq \text{nothing}$$

4. Application de la règle pour les rôles qualifiés :

$$\forall \text{mange}.(\text{animal} \sqcap \text{plante}) \sqsubseteq \forall \text{mange}.\text{nothing}$$

Chaîne de raisonnement pour le deuxième test :

1. D'après le premier test : `carnivoreExc` $\sqcap$ `herbivoreExc` $\sqsubseteq$ $\forall$`mange.nothing`
2. Propriété des conjonctions :

   $$(\text{carnivoreExc} \sqcap \text{herbivoreExc}) \sqcap \text{animal} \sqsubseteq \forall \text{mange}.\text{nothing} \sqcap \text{animal}$$

3. De la T-Box : `animal` $\sqsubseteq$ $\exists$`mange`
4. Donc :

   $$\forall \text{mange}.\text{nothing} \sqcap \text{animal} \sqsubseteq \forall \text{mange}.\text{nothing} \sqcap \exists \text{mange}$$

5. Utilisation de la T-Box :

   $$\forall \text{mange}.\text{nothing} \sqcap \exists \text{mange} \sqsubseteq \text{nothing}$$

Test 3 : `(carnivoreExc` $\sqcap$ `animal)` $\sqcap$ `herbivoreExc` $\sqsubseteq_s$ `nothing`

Résultat observé :

```
?- subsS(and(and(carnivoreExc, animal), herbivoreExc), nothing).
false.
```

Analyse de l'échec :

Cette requête échoue bien qu'elle soit logiquement équivalente à la requête précédente. La différence réside dans la structure syntaxique des conjonctions :

- **Structure de la requête 2 :** `and(and(carnivoreExc, herbivoreExc), animal)`
- **Structure de la requête 3 :** `and(and(carnivoreExc, animal), herbivoreExc)`

Notre règle ajoutée ne s'applique que lorsque les deux concepts `all(R,C)` et `all(R,D)` apparaissent directement au premier niveau de la conjonction. Dans la requête 3, la structure `and(and(carnivoreExc, animal), herbivoreExc)` empêche cette règle de s'appliquer directement.

Question 4.4 : Nécessité de règles pour les concepts de la forme $\exists R$?

Réponse

Non, il n'est pas nécessaire d'écrire des règles similaires pour les concepts de la forme $\exists R$.

Justification

1. **Différence fondamentale dans FL$^-$** :
   — Dans FL$^-$, $\exists R$ est un **concept de base** (concept atomique)
   — Alors que $\forall R.C$ est un **concept composite** (construction complexe)
2. **Traitement dans notre implémentation** :
   — Les concepts $\exists R$ sont représentés par `some(R)` et traités comme des concepts atomiques
   — Ils apparaissent directement dans la T-Box sous forme de faits `subs/2`
   — Exemple : `subs(animal, some(mange))`
3. **Gestion par les règles existantes** :
   — Les subsomptions impliquant $\exists R$ sont déjà gérées par nos règles de base :

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

   — Aucune règle spéciale n'est nécessaire car $\exists R$ n'a pas de "composant interne" à manipuler
4. **Contraste avec** $\forall R.C$:
   — $\forall R.C$ a une structure interne (le concept C) qui nécessite un traitement spécial
   — Notre règle `subsS(all(R,C), all(R,D), L) :- subsS(C,D,L)` permet de raisonner sur cette structure interne
   — $\exists R$ n'a pas cette complexité structurelle

Question 4.5 : Requêtes `lion` $\sqsubseteq_s$ `X` et `X` $\sqsubseteq_s$ `predateur`

Analyse théorique des résultats attendus

Requête `lion` $\sqsubseteq_s$ `X` Cette requête recherche tous les concepts X qui sont subsumés par `lion`. Selon la T-Box et les règles de subsomption, les résultats attendus incluent :

— `X = lion` (par réflexivité)
— `X = felin` (subsomption directe)
— `X = carnivoreExc` (subsomption directe)
— `X = mammifere` (transitivité via `felin`)
— `X = animal` (transitivité via `mammifere`)
— `X = etreVivant` (transitivité via `animal`)
— `X = some(mange)` (transitivité via `animal`)

— `X = predateur` (transitivité via `carnivoreExc`)

Résultats expérimentaux obtenus

Requête `lion` $\sqsubseteq_s$ `X`

```
?- subsS(lion,X).
X = lion ;
X = felin ;
X = carnivoreExc ;
X = mammifere ;
X = animal ;
X = etreVivant ;
X = some(mange) ;
X = predateur ;
X = all(mange, animal) ;
X = all(mange, etreVivant) ;
X = all(mange, some(mange)) ;
X = all(mange, animal) ;
X = all(mange, etreVivant) ;
X = all(mange, some(mange)) ;
false.
```

Requête `X` $\sqsubseteq_s$ `predateur`

```
?- subsS(X,predateur).
X = predateur ;
X = carnivoreExc ;
X = lion ;
X = all(mange, animal) ;
X = and(lion, carnivoreExc) ;
X = and(lion, predateur) ;
X = and(lion, all(mange, animal)) ;
X = and(carnivoreExc, predateur) ;
X = and(carnivoreExc, all(mange, animal)) ;
X = and(all(mange, animal), carnivoreExc) ;
X = and(all(mange, animal), predateur) ;
false.
```

Analyse des résultats obtenus

Correspondance avec les attentes de base

— Tous les concepts atomiques attendus sont présents dans les résultats
— La réflexivité, la transitivité et les subsomptions directes sont correctement gérées

Résultats supplémentaires observés **Pour** `lion` $\sqsubseteq_s$ `X` :
— Apparition de concepts de la forme `all(mange, ...)` :
  — Ces résultats proviennent de l'équivalence `carnivoreExc` $\equiv$ `all(mange, animal)`
  — Notre système exploite cette équivalence pour générer des concepts supplémentaires
  — La duplication des résultats est due au mécanisme de backtracking de Prolog

Pour `X` $\sqsubseteq_s$ `predateur` :
— Apparition de nombreuses conjonctions :
  — Ces résultats démontrent la puissance de nos règles pour les conjonctions
  — Le système génère toutes les conjonctions possibles qui subsument `predateur`
  — Exemple : `and(lion, carnivoreExc)` subsume `predateur` car les deux conjunctes subsument individuellement `predateur`

Explication des duplications Les duplications observées (comme `all(mange, animal)` apparaissant deux fois) sont dues à :
— Le mécanisme de backtracking de Prolog qui trouve plusieurs chemins de preuve
— L'application de différentes séquences de règles aboutissant aux mêmes conclusions
— Ce comportement est normal et attendu dans les systèmes d'inférence Prolog

Interprétation des résultats complexes

Validité des résultats supplémentaires
— Tous les concepts retournés sont corrects d'un point de vue logique
— `lion` $\sqsubseteq$ `all(mange, animal)` car `lion` $\sqsubseteq$ `carnivoreExc` $\equiv$ `all(mange, animal)`
— `and(lion, carnivoreExc)` $\sqsubseteq$ `predateur` car `carnivoreExc` $\sqsubseteq$ `predateur`

Richeses démontrées par ces résultats
— Notre système est capable de découvrir des relations de subsomption non triviales

— La gestion des équivalences et des conjonctions fonctionne de manière extensive
— Le système explore automatiquement les conséquences logiques des axiomes de la T-Box

Exercice 5 - Complétude

Définition de la complétude

Un ensemble de règles est dit **complet** pour un langage donné s'il peut prouver toute subsomption $C \sqsubseteq D$ valide sémantiquement, à condition que tous les termes de la T-Box et de la requête appartiennent au langage en question.

Analyse de la complétude de notre système pour $\mathbf{FL}^-$

Évaluation de la complétude

Notre système n'est pas complet pour $\mathbf{FL}^-$.

Preuves de l'incomplétude

Plusieurs limitations observées démontrent l'incomplétude de notre système :

1. **Limitations avec les structures de conjonctions** :
   — La requête $(\texttt{carnivoreExc} \sqcap \texttt{animal}) \sqcap \texttt{herbivoreExc} \sqsubseteq_s \texttt{nothing}$ échoue
   — Pourtant, cette subsomption est logiquement valide
   — L'échec provient de l'incapacité du système à reconnaître l'équivalence entre différentes structures arborescentes de conjonctions
2. **Gestion partielle de la commutativité et de l'associativité** :
   — Notre système ne reconnaît pas toutes les réorganisations possibles des conjunctes
   — Seule une forme limitée de commutativité est implémentée
3. **Limitations dans les interactions entre règles** :
   — Certaines combinaisons de règles ne sont pas capturées
   — L'ordre d'application des règles peut empêcher certaines inférences
4. **Absence de normalisation des concepts** :
   — Notre système travaille directement sur la syntaxe des concepts
   — Aucune normalisation n'est effectuée pour uniformiser les structures équivalentes

Exemple concret d'incomplétude

Considérons les trois requêtes suivantes, logiquement équivalentes :

1. `(carnivoreExc` $\sqcap$ `herbivoreExc)` $\sqcap$ `animal` $\sqsubseteq_s$ `nothing` **(réussite)**
2. `(carnivoreExc` $\sqcap$ `animal)` $\sqcap$ `herbivoreExc` $\sqsubseteq_s$ `nothing` **(échec)**
3. `(herbivoreExc` $\sqcap$ `animal)` $\sqcap$ `carnivoreExc` $\sqsubseteq_s$ `nothing` **(échec)**

Seule la première structure syntaxique est reconnue par notre système, bien que les trois soient sémantiquement équivalentes.

Cas de complétude partielle

Notre système est cependant complet pour certains sous-ensembles de FL^- :

- **Concepts atomiques** : Complétude pour les subsomptions entre concepts atomiques
- **Conjonctions simples** : Complétude pour les cas où les structures de conjonctions correspondent exactement aux règles implémentées
- **Rôles qualifiés** : Complétude pour les subsomptions directes entre concepts universellement quantifiés partageant le même rôle

Compromis entre correction et complétude

Il est important de noter que notre système respecte le compromis fondamental :

- **Correction** : **Garantie** - Si le système prouve $C \sqsubseteq_s D$, alors $C \sqsubseteq D$ est sémantiquement valide
- **Complétude** : **Non garantie** - Certaines subsomptions valides ne sont pas prouvées par le système

Ce compromis est tout à fait acceptable dans le cadre de ce projet, comme précisé dans l'introduction : *"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."*

Conclusion

Ce projet nous a permis de développer un système d'inférence progressif pour la logique de description FL^- en Prolog, en suivant une approche modulaire et incrémentale. Notre travail démontre la faisabilité de la mise en œuvre d'un raisonneur pour un fragment de logique de description, tout en

respectant les contraintes de correction et en acceptant certaines limitations en termes de complétude.

Bilan des réalisations

- **Représentation efficace** : Nous avons établi une correspondance claire entre les constructions FL^- et leur représentation Prolog, permettant une manipulation naturelle des concepts et des relations de subsomption.
- **Gestion robuste des concepts atomiques** : L'implémentation initiale a été renforcée par un mécanisme de prévention des cycles, garantissant la terminaison des requêtes même en présence de circularités dans la T-Box.
- **Extension progressive** : L'ajout successif des règles pour les équivalences, les conjonctions et les rôles qualifiés a permis d'étendre progressivement les capacités d'inférence du système tout en maintenant sa correction.
- **Tests exhaustifs** : Les nombreux tests effectués ont validé le comportement du système sur une variété de cas, des subsomptions simples aux inférences complexes impliquant des contradictions.

Limitations et perspectives

Notre système présente certaines limitations, principalement liées à son incomplétude :

- **Sensibilité à la structure syntaxique** : Le système échoue sur certaines requêtes logiquement valides lorsque la structure des conjonctions ne correspond pas exactement aux motifs attendus par les règles.
- **Absence de normalisation** : L'absence de phase de normalisation des concepts limite la capacité du système à reconnaître des structures équivalentes mais syntaxiquement différentes.
- **Interactions complexes** : Certaines combinaisons de règles ne sont pas capturées, conduisant à des échecs sur des inférences pourtant valides.

Ces limitations ouvrent des perspectives d'amélioration intéressantes :

- **Normalisation des concepts** : L'ajout d'une phase de normalisation permettrait de uniformiser la représentation des concepts et d'améliorer la complétude.
- **Règles supplémentaires** : L'enrichissement du système avec des règles pour gérer d'autres constructions logiques ou des interactions plus complexes.

— **Optimisations** : L'amélioration des performances grâce à des techniques de mémoïsation ou d'indexation.

Apports pédagogiques

Ce projet a constitué une expérience formatrice à plusieurs niveaux :

— **Compréhension des logiques de description** : L'implémentation concrète a renforcé la compréhension des concepts fondamentaux de FL^- et de leur sémantique.

— **Maîtrise de Prolog** : Le développement a permis d'approfondir la maîtrise de la programmation logique, notamment la gestion de la récursivité, du backtracking et des structures de données.

— **Approche méthodologique** : La construction progressive du système a illustré l'importance d'une approche incrémentale dans le développement de systèmes complexes.

— **Analyse critique** : L'évaluation de la correction et de la complétude a développé notre capacité à analyser les limites d'un système et à identifier les axes d'amélioration.

Conclusion générale

Notre système d'inférence pour FL^- représente une implémentation fonctionnelle et correcte, capable de traiter une large gamme de requêtes de subsomption. Bien qu'incomplet, il démontre la faisabilité de la mise en œuvre d'un raisonneur pour les logiques de description et fournit une base solide pour des extensions futures.

Ce projet illustre parfaitement le compromis entre correction et complétude qui caractérise de nombreux systèmes d'inférence pratiques. Il souligne également l'importance de bien comprendre les limitations d'un système pour l'utiliser de manière appropriée dans des applications réelles.

Le code produit, associé à cette analyse réflexive, constitue une contribution significative à l'apprentissage des logiques de description et de leur implémentation, tout en ouvrant des perspectives stimulantes pour des travaux futurs dans ce domaine.