Recherche des contre-exemples aux syllogismes à l’aide des diagrammes de Venn

Bou Zeid Elie
Rayan Boughanem

Réponses aux questions

1. Expliquer comment implanter l'extension d'un diagramme d en fonction d'un diagramme c incompatible en Ocaml.

On a introduit une fonction **"contrainte"** qui prend en argument deux diagrammes d1 et d2, et renvoie un diagramme contenant les contraintes de vacuité de d2 qui ne sont pas dans d1, et si il n'en existe pas elle renvoie un diagramme vide (Diag.empty)

Cette fonction a été utilisé dans **"extend_contre_ex"**, il faut que le diagramme de prémisse soit passer en premier puis celui de la conclusion:

```
extend_contre_ex dpremisse conclusion;;
```

On vérifie en premier si les deux diagrammes mis en argument son compatible avec **"est_compatible_diag_diag"** alors on renvoie **dpremisse** sans changement, sinon on fait appel à la fonction **"contrainte"** et si cette dérnière renvoie un diagramme vide on renvoie **dpremisse** sans changement sinon on ajoute la première contrainte qu'on a trouvé dans dpremisse en transformant le diagramme en liste pour extracter la zone du diagramme qu'on a besoin et en lui mettant une contrainte de non vacuité.

On aura à la fin un diagramme **dpremisse** étendu en fonction des contraintes de **conclusion.**

2. Expliquer comment implanter le calcul des ensembles correspondant à une numérotation de croix d'un diagramme de Venn en Ocaml.

Ce calcul est effectué par la fonction "**diag_to_predicate_def"**, qui prend en paramètre un diagramme, cette fonction est composé de 3 fonctions récursives terminales, en premier on verifie si le diagramme contient des croix en transformant le diagramme en liste et vérifiant s'il y a des contraintes de non vacuité avec **"List.exist"**, alors on filtre la liste pour avoir que des contraintes de non vacuité, sinon on renvoie le couple None, **Predicate_def.empty**.

On parcours cette liste et à chaque itération on ajoute dans une liste initialement vide le **Predicate_set** qui est la zone où se situe la croix comme premier élément avec le numéro d'itération qui commence à 0 qui correspond au numéro de la croix comme deuxième élément des couples.
Prenons le diagramme D1 de l'énoncé,
Si on transforme les set en liste avec **"List.map"** on aura alors une liste de la forme:

```
[(["b"], [2]); (["a"; "b"], [1]); (["a"], [0])]
```

Mais ce n'est pas la forme qu'on souhaite avoir, on veut avoir chaque ensemble seul avec les croix qui sont dans cet ensemble, donc une croix peut être dans plusieurs ensemble en même temps, c'est pour cela on a besoin d'une deuxième fonction récursive pour parcourir la liste qu'on a obtenu et une troisième pour parcourir chaque **Predicate_set** dans la liste.
Donc en parcourant chaque **Predicate_Set** qui est un set de string en le transformant en liste de string, on a un accumulateur initialisé à **Predicate_def.empty**, on vérifie si chaque string du set existe dans l'accumulateur, s'il existe déjà il a alors une valeur qui lui est associé, on récupère cette

valeur et on y ajoute la valeur associé à ce string dans la liste qu'on parcourt, sinon on ajoute directement le string dans l'accumulateur avec "**Predicate_def.add"**.
À la fin on aura un un **predicate_def** bien construit:

Predicate_def transformé en liste

```
[("a", "{0, 1}"); ("b", "{1, 2}")]
```

3. Expliquer comment implanter l'évaluation d'une formule d'un syllogisme sous une interprétation donnée en Ocaml.

La fonction qui évalue la formule d'un syllogisme est **"eval_syllogisme"** qui est une fonction d'un foncteur qui la construit sur des types énumérables.

Les types énumérables reviennent au module qui représente les entiers de 0 à n, donc ce que fait la fonction est une itération recursive sur les valeurs du module en utilisant la fonction '**_.values'** qui renvoie une séquence puis on transforme cette séquence en liste pour la parcourir avec **"List.of_seq"**, si la formule correspond à un quantificateur existensielle on s'arrête dès qu'on trouve une valeur qui satisfait l'interprétation donnée, si on arrive à la fin de la liste alors aucune valeur ne la vérifie, et si la formule correspond à un quantificateur universelle, il faut que toute les valeurs satisfaient l'interprétation, sinon on s'arrête dès qu'on trouve une valeur qui ne satisfait pas l'interprétation.

Conclusion

Pour conclure, ce rapport explore l'analyse des syllogismes à travers les diagrammes de Venn en OCaml, automatisant leur extension, leur numérotation et leur évaluation. On a expliqué comment on a implanté chaque fonction, les structures de données qu'on a utilisé comme les listes et des **predicate_def** qui correspondent à des Map etc.. En plus du mauvais résultat auquel on a fait face dans "**diag_to_predicate_def"**, et comment on a utilisé ce résultat pour arriver à la bonne solution