BURDIN Kevin 11507706

INAYA Victor 11510092

Compte Rendu Projet BDD

**Archive du projet**

L’archive du projet contient :

Ce present rapport, expliquant le projet dans les grandes lignes, le code source du projet, dans BDD\_Projet\_2020\_2021/ BDD\_Projet\_2020\_2021. Le dossier src présente le code Java (main/java), ainsi que les exemples pouvant être testés avec le projet (main/ressources), et enfin un dossier « DES\_programs » qui contient les différents datasets en datalog utilisés pour comparer l’exécution DES l’exécution de l’interpréteur. Pour compiler : mvn clean / mvn package, pour executer : mvn exec :java (ou alors, lancer l’exécutable target/mif37-dex-1.0-SNAPSHOT-jar-with-dependencies.jar

**Top-Down Query evaluation**

Un interpréteur Datalog standard fonctionne par rondes : il va s’appuyer sur une liste de faits et de règles et, à chaque ronde, va évaluer l’ensemble des règles en se basant sur ce qu’il a calculé à la ronde précédente (cf. figure 1 dans les Annexes).

Cependant, force est de constater que procéder d’une telle façon n’est pas optimale lorsqu’il s’agit seulement de connaître la réponse à une requête donnée. L’heuristique Top-Down Query evaluation va permettre de calculer directement la requête, sans passer par ce système de rondes.

En effet, lors de la Top-Down Query evaluation, nous allons directement chercher la règle de la requête qui nous intéresse. Par la suite, lorsque l’on tombe sur un fait de type « IDB », donc se basant sur des règles, nous allons injecter les constantes que nous cherchons dans lesdites règles du programme (Top-Down passing information), les faires passer dans le corps de la règle et l’évaluer afin de résoudre l’IDB de la requête (Side-ways passing information).

Le projet se base sur cette heuristique. Nous allons expliquer les subtilités du code source de notre interpréteur.

**Explications du code**

L’interpréteur se base sur l’attribution d’un ornement pour chaque atome présent dans le corps d’une règle. Il s’agit de définir si une variable est liée (si une ou plusieurs constantes lui sont attribuées) ou libre (donc à calculer).

Après avoir attribué un ornement à chaque atome de la règle, l’interpréteur va chercher, pour chaque variable libre, une valeur qui lui correspond dans une autre règle (une sous-requête). Une fois tous les ornements calculés, la sous-requête qui leur correspond peut être calculée, afin de résoudre une requête antérieure.

Le code comporte plusieurs classes Java qui méritent notre attention :

Une classe Mapping qui se charge de la conversion d’un fichier texte vers les différents objets Java dont l’interpréteur aura besoin ; une classe Atom, afin de stocker les atomes des règles, et leurs arguments ; une classe AdornedAtom, qui, va représenter un atome orné, par une liste de booléens en se basant sur les positions des arguments de l’atome ; une classe AdornedTgd qui va représenter la règle ornée dans son ensemble ; et enfin, une classe RecursiveQsqEngine qui va comporter notre interpréteur. Cette classe étant la plus complexe, c’est celle que nous allons expliciter.

Cette classe comprend une classe interne : QSQRState, qui va stocker l’ensemble des règles (unadornedRules), des règles ornées (adornedRules), des réponses aux sous-requêtes (ans), ainsi que des valeurs prises par chaque variable de chaque règle (inputByRules).

La requête principale est calculée par la fonction query. On va lui donner en paramètre l’entête de la requête que l’on souhaite évaluer et elle va nous retourner une liste de 3 objets : en premier, le résultat de la requête, puis le résultat de toutes les sous-requête résolues pour trouver la réponse, et enfin, tout le détail de l’exécution, donc le contenu de « QSQRState » (cf. figure 2). Cette fonction va avoir la responsabilité d’orner le corps de la requête passée en paramètre, d’appeler une fonction qui va l’évaluer, puis de renvoyer le résultat. Elle va appeler la fonction QSQSubroutine.

Cette dernière a la responsabilité d’évaluer une sous-requête déjà ornée : elle va, d’abord, vérifier que toutes les variables sont liées. Si tel est le cas, elle sera capable de fournir une réponse à la sous-requête qu’elle évalue : toutes les variables ont une valeur. Dans le cas contraire, elle va appeler une autre fonction : qsqr.

Qsqr va s’occuper de trouver les sous-requêtes nécessaires à l’évaluation de la requête, à travers les règles non ornées, que l’on peut assimiler à la « Top-down passing information ». Il va transmettre les éventuelles constantes au corps de chaque règle ainsi trouvée (Sideways passing information), les orner en conséquence, puis les renvoyer à la fonction d’évaluation.

Ces trois fonctions gèrent l’interprétation de la requête. Elles possèdent également quelques particularités notables :

La fonction QSQSubroutines se base sur les Inputs qu’on lui transmet pour donner un résultat. Chaque nouvelle réponse est donc convertie en input après les appels à qsqr. De plus, elle gère les récursions et vérifie si de nouvelles réponses sont ajoutées après leur évaluation (cas d’arrêt).

La fonction qsqr peut aussi mettre en lumière des contradictions. Par exemple, si l’on a un edb sans résultat.

**Difficultés rencontrées**

Durant ce projet, plusieurs difficultés ont été rencontrés :

* Tout d’abord, la première difficulté a été la compréhension du concept de Top-Down evaluation. En effet, cette façon d’aborder Datalog est nouveau et complètement différente du système Datalog standard. Nous avons été habitués au système de rondes à travers les cours et les TPs et il nous a fallu penser autrement.
* Ensuite, Nous avons reçu un code source qui nous était inconnu. Il nous a donc fallu le comprendre et nous y imprégner afin de pouvoir nous y appuyer efficacement.
* Enfin, le Datalog est un langage pour gérer des données. L’interpréter signifie avoir recours à de très nombreuses structures de données que nous propose Java, ainsi que penser à tous les cas de figures possibles de Datalog.

**Mesure de temps d’exécution**

La mesure du temps d’exécution de notre code a été fait grâce à la fonction Java « CurrentTimeMillis ». Voir figure 4.

**Programmes Datalog**

Pour nous assurer du bon fonctionnement de notre programme, nous avons réalisé une série de tests sur DES et sur notre interpréteur. Voir la section « Annexes – programmes Datalog ». Les codes Datalog utilisés sur notre interpréteur sont dans *src/main/java/ressources*. Il suffit de changer le fichier dans « App » afin de l’interpréter.

**Annexes**

Une image contenant texte

Description générée automatiquement

Figure 1 - Déroulement d'un programme Datalog

Une image contenant texte

Description générée automatiquement

Figure 2-La classe "QSQRState"

Une image contenant texte

Description générée automatiquement

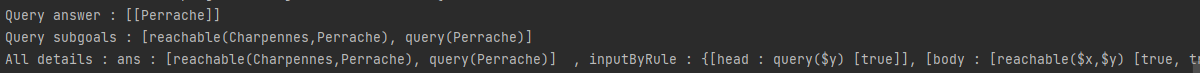


Figure 3-Détails de la réponse

Une image contenant texte

Description générée automatiquement

Figure 4- Temps d'execution

**Annexes – programmes Datalog**

Datalog Dataset :

Dataset 1 :

Une image contenant table

Description générée automatiquement

Dataset 1 – bis

Une image contenant table

Description générée automatiquement

Dataset 2 :

Une image contenant table

Description générée automatiquement

Nous allons tester notre interpréteur Datalog, fonctionnalité par fonctionnalité :

Exemple 1 : Requête sur un EDB, avec projection :

Sur DES, avec le Dataset 1 :

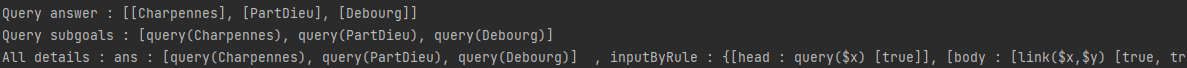
Une image contenant texte

Description générée automatiquement

Sur notre interpréteur :

Une image contenant texte

Description générée automatiquement



Exemple 2 : Requête sur un EDB avec sélection

Sur DES avec le Dataset 1 :

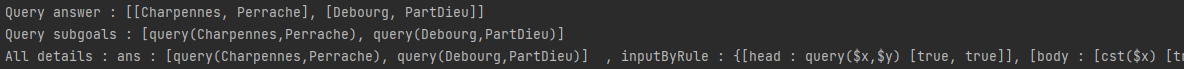
Une image contenant texte

Description générée automatiquement

Sur notre interpréteur :

Une image contenant texte

Description générée automatiquement



Exemple 3 : Requête sur un EDB avec sélection et projection

Sur DES avec le Dataset 1 :

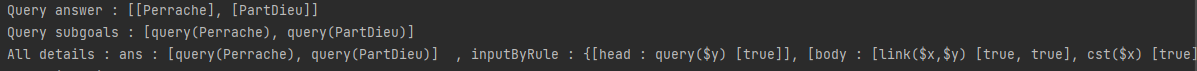
Une image contenant texte

Description générée automatiquement

Sur notre interpréteur :

Une image contenant texte

Description générée automatiquement



Exemple 4 : Requête avec une sous-requête

Sur DES avec le Dataset 1 :

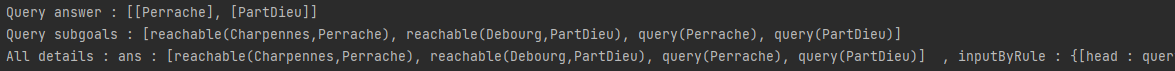
Une image contenant texte

Description générée automatiquement

Sur notre interpréteur :

Une image contenant texte

Description générée automatiquement



Exemple 5 :

Sur DES avec le Dataset 1 bis :

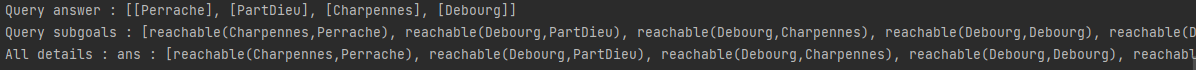
Une image contenant texte

Description générée automatiquement

Sur notre interpréteur :

Une image contenant texte

Description générée automatiquement



Exemple 6 :

Sur DES avec le Dataset 1-bis sans le cst(‘Debourg’) :

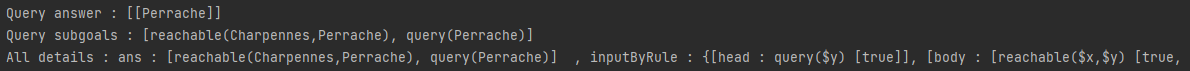
Une image contenant texte

Description générée automatiquement

Sur notre interpréteur :

Une image contenant texte

Description générée automatiquement



Exemple 7 :

Sur DES avec le Dataset 2 :

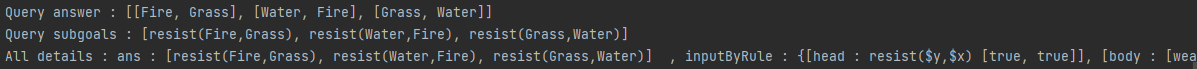
Une image contenant texte

Description générée automatiquement

Sur notre interpréteur :

Une image contenant texte

Description générée automatiquement



Exemple 8 :

Sur DES avec le Dataset 2 :

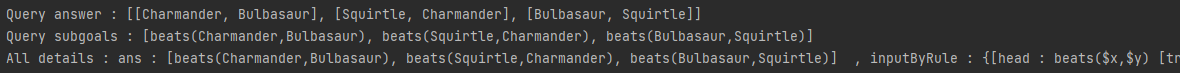
Une image contenant texte

Description générée automatiquement

Sur notre interpréteur :

Une image contenant texte

Description générée automatiquement



Exemple 9 :

Sur DES avec le Dataset 2 :

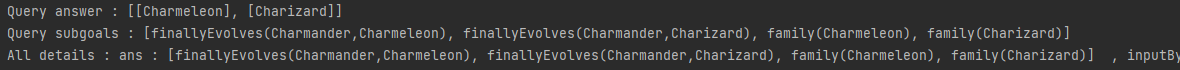
Une image contenant texte

Description générée automatiquement

Sur notre interpréteur :

Une image contenant texte

Description générée automatiquement



Exemple 10 :

Sur DES avec le Dataset 2 :

Une image contenant texte

Description générée automatiquement

Sur notre interpréteur :

Une image contenant texte

Description générée automatiquement

