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.

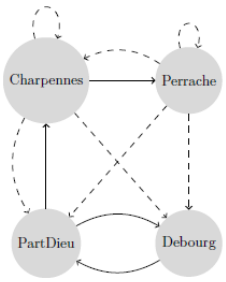
Programmes Datalog

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

Example 1.1 (Reachability Queries). Consider the following Datalog program. It consists of a set of **facts**, denoting that there is a *link* — marked by an arrow — between two stations, and of a set of **rules** indicating the *metro* names and the *reachability* between two stations (via multiple links) — marked by a dotted arrow. The program **query** computes all *metro* stations that are *reachable* from *Charpennes*.

```
link(Charpennes, Perrache)
link(PartDieu, Charpennes)
link(Debourg, PartDieu)
link(PartDieu, Debourg)
link(X,Y)→metro(X).
link(X,Y)→metro(Y).
link(X,Y)→reachable(X,Y).
link(X,Z), reachable(Z,Y)→reachable(X,Y).
reachable(Charpennes, Y) → query(Y).
```



The *bottom-up* evaluation of the program above, proceeds as follows to compute its fixpoint model:

$T_P^0(\emptyset) = \{link(Charpennes, Perrache), link(PartDieu, Charpennes), link(Debourg, PartDieu), link(PartDieu, Debourg)\}$

$T_P^1(T_P^0) = T_P^0 \cup \{metro(Charpennes), metro(Perrache), metro(PartDieu), metro(Debourg), reachable(Charpennes, Perrache), reachable(PartDieu, Charpennes), reachable(Debourg, PartDieu), reachable(PartDieu, Debourg)\}$

$T_P^2(T_P^1) = T_P^1 \cup \{reachable(PartDieu, Perrache), reachable(Debourg, Charpennes), reachable(Debourg, Debourg), reachable(PartDieu, PartDieu), query(Perrache)\}$

$T_P^3(T_P^2) = T_P^2 \cup \{reachable(Debourg, Perrache)\}$

Figure 1 - Déroulement d'un programme Datalog

```java
private class QSQRState {
    /** Tracks the answer tuples generated for each adorned predicate. */
    private Collection<fr.univlyon1.mif37.dex.mapping.Relation> ans;// Datas for each IDB, Map<String,List<String[]>> -> ans[IDB] = {couple of answers}
    /** Tracks which input tuples have been used for each rule. */
    private Map<AdornedTgd, Map<String,Map<String,List<String>>>> inputByRule;//Map<AdornedTgd, Map<String,Map<String,List<String>>>   > -> inputByRule[
    /** Holds all the adorned rules for a given adorned predicate. */
    private Map<String, List<AdornedTgd>> adornedRules;//Map<String, List<AdornedTgd>> -> adorned rules for a predicate
    /** Holds all the unadorned rules for a given predicate. */
    private final Map<String, List<Tgd>>  unadornedRules; // Map<String, List<Tgd> -> all rules for each IDB


    /** Initializes state with a set of all unadorned rules for the program. ...*/
    public QSQRState(Map<String, List<Tgd>> unadornedRules) {
        this.ans = new ArrayList<>();
        this.inputByRule = new LinkedHashMap<>();
        this.adornedRules = new LinkedHashMap<>();
        this.unadornedRules = unadornedRules;

    }

    @Override
    public String toString() {
        return "ans : " + ans + "  , inputByRule : " + inputByRule + " , adornedRules : " + adornedRules + "  , unardornedRules : " + unadornedRules;
    }
```

Figure 2-La classe "QSQRState"

```
System.out.println(
        "Query answer : " +
        answer.get(0) +
                "\nQuery subgoals : " +
                answer.get(1) +
                "\nAll details : " +
                answer.get(2)

);
```

```
Query answer : [[Perrache]]
Query subgoals : [reachable(Charpennes,Perrache), query(Perrache)]
All details : ans : [reachable(Charpennes,Perrache), query(Perrache)]  , inputByRule : {[head : query($y) [true]], [body : [reachable($x,$y) [true, t
```

Figure 3-Détails de la réponse

```
public static void main(String[] args) throws Exception {
    long initialTime = System.currentTimeMillis();
    MappingParser mp = new MappingParser(App.class.getResourceAsStream( name: "/exemple1.txt"));
    Mapping mapping = mp.mapping();
    RecursiveQsqEngine engine = new RecursiveQsqEngine(mapping);
    List<Object> answer = engine.query(((Tgd)mapping.getTgds().toArray()[mapping.getTgds().size()-1]).getRi
    long executionTime = System.currentTimeMillis()-initialTime;
    System.out.println(
            "Query answer : " +
            answer.get(0) +
                    "\nQuery subgoals : " +
                    answer.get(1) +
                    "\nAll details : " +
                    answer.get(2)

    );
    System.out.println("execution time : " + executionTime/1000.0 + " seconds");
}
```

```
1 sec, 394 ms
[INFO] Building mif37-dex 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @ mif37-dex ---
Query answer : [[Perrache]]
Query subgoals : [reachable(Charpennes,Perrache), query(Perrache)]
All details : ans : [reachable(Charpennes,Perrache), query(Perrache)]  , inputByR
execution time : 0.008 seconds
[INFO] ------------------------------------------------------------------------
```

Figure 4- Temps d'execution

Annexes – programmes Datalog

Datalog Dataset :

Dataset 1 :

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                      Datalog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%          METRO DATABASE
%%
%%  link(from, to)
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Optionally declare types
:- type(cst(test:string)).
cst('Charpennes').
cst('Debourg').
:-type(link(from:string,to:string)).

link('Charpennes','Perrache').
link('PartDieu','Charpennes').
link('Debourg','PartDieu').
link('PartDieu','Debourg').

metro(X) :- link(X,Y).
metro(Y) :- link(X,Y).

reachable(X,Y) :- link(X,Y).
```

Dataset 1 – bis

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                      Datalog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%          METRO DATABASE
%%
%%  link(from, to)
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Optionally declare types
:- type(cst(test:string)).
cst('Charpennes').
cst('Debourg').
:-type(link(from:string,to:string)).

link('Charpennes','Perrache').
link('PartDieu','Charpennes').
link('Debourg','PartDieu').
link('PartDieu','Debourg').

metro(X) :- link(X,Y).
metro(Y) :- link(X,Y).

reachable(X,Y) :- link(X,Y).
reachable(X,Y) :- reachable(X,Z), link(Z,Y).
```

Dataset 2 :

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                      Datalog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%          POKEMON DATABASE
%%
%%
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Optionally declare types
:- type(cst(Cname:string)).
cst('Bulbasaur').
:- type(type(typename:string)).
type('fire').
type('water').
type('grass').
:-type(weak(type:string,against:string)).
weak('fire','water').
weak('water','grass').
weak('grass','fire').

:-type(hasType(monster:string,type:string)).
hasType('Squirtle','water').
hasType('Bulbasaur','grass').
hasType('Charmander','fire').

:-type(evolves(monster1:string,monster2:string)).
evolves('Charmander','Charmeleon').
evolves('Charmeleon','Charizard').
evolves('Bulbasaur','Ivysaur').
evolves('Ivysaur','Venusaur').
evolves('Squirtle','Wartortle').
evolves('Wartortle','Blastoise').

finallyEvolves(X,Y) :- evolves(X,Y).
finallyEvolves(X,Y) :- finallyEvolves(X,Z),evolves(Z,Y).
```

Nous allons tester notre interpréteur Datalog, fonctionnalité par fonctionnalité :

Exemple 1 : Requête sur un EDB, avec projection :

Sur DES, avec le Dataset 1 :

```
DES> query(X) :- link(X,Y).

Warning: This view has a singleton variable: [Y]
Info: Processing:
  query(X) :-
    link(X,Y).
{
  query('Charpennes'),
  query('Debourg'),
  query('PartDieu')
}
Info: 3 tuples computed.
```

Sur notre interpréteur :

```
EDB
link(Charpennes,Perrache)
link(PartDieu,Charpennes)
link(Debourg,PartDieu)
link(PartDieu,Debourg)
cst(Charpennes)

IDB
metro($x)
reachable($x,$y)
query($y)

MAPPING
link($x,$y) -> metro($x) .
link($x,$y) -> metro($y) .
link($x,$y) -> reachable($x,$y) .
link($z,$y) , reachable($x,$z) -> reachable($x,$y) .
link($x,$y) -> query($x) .
```

```
Query answer : [[Charpennes], [PartDieu], [Debourg]]
Query subgoals : [query(Charpennes), query(PartDieu), query(Debourg)]
All details : ans : [query(Charpennes), query(PartDieu), query(Debourg)]  , inputByRule : {[head : query($x) [true]], [body : [link($x,$y) [true, tr
```

Exemple 2 : Requête sur un EDB avec sélection

Sur DES avec le Dataset 1 :

```
DES> query(X,Y) :- cst(X),link(X,Y).

Info: Processing:
  query(X,Y) :-
    cst(X),
    link(X,Y).
{
  query('Charpennes','Perrache'),
  query('Debourg','PartDieu')
}
Info: 2 tuples computed.
```

Sur notre interpréteur :

```
EDB
link(Charpennes,Perrache)
link(PartDieu,Charpennes)
link(Debourg,PartDieu)
link(PartDieu,Debourg)
cst(Charpennes)
cst(Debourg)

IDB
metro($x)
reachable($x,$y)
query($y)

MAPPING
link($x,$y) -> metro($x) .
link($x,$y) -> metro($y) .
link($x,$y) -> reachable($x,$y) .
link($z,$y) , reachable($x,$z) -> reachable($x,$y) .
cst($x) ,link($x,$y) -> query($x,$y) .
```

```
Query answer : [[Charpennes, Perrache], [Debourg, PartDieu]]
Query subgoals : [query(Charpennes,Perrache), query(Debourg,PartDieu)]
All details : ans : [query(Charpennes,Perrache), query(Debourg,PartDieu)]  , inputByRule : {[head : query($x,$y) [true, true]], [body : [cst($x) [t
```

Exemple 3 : Requête sur un EDB avec sélection et projection

Sur DES avec le Dataset 1 :

```
DES> query(Y) :- cst(X),link(X,Y).

Info: Processing:
  query(Y) :-
    cst(X),
    link(X,Y).
{
  query('PartDieu'),
  query('Perrache')
}
Info: 2 tuples computed.
```

Sur notre interpréteur :

```
EDB
link(Charpennes,Perrache)
link(PartDieu,Charpennes)
link(Debourg,PartDieu)
link(PartDieu,Debourg)
cst(Charpennes)
cst(Debourg)

IDB
metro($x)
reachable($x,$y)
query($y)

MAPPING
link($x,$y) -> metro($x) .
link($x,$y) -> metro($y) .
link($x,$y) -> reachable($x,$y) .
link($z,$y) , reachable($x,$z) -> reachable($x,$y) .
cst($x),link($x,$y) -> query($y) .
```

```
Query answer : [[Perrache], [PartDieu]]
Query subgoals : [query(Perrache), query(PartDieu)]
All details : ans : [query(Perrache), query(PartDieu)]  , inputByRule : {[head : query($y) [true]], [body : [link($x,$y) [true, true], cst($x) [true]
```

Exemple 4 : Requête avec une sous-requête

Sur DES avec le Dataset 1 :

```
DES> query(Y) :- cst(X), reachable(X,Y).

Info: Processing:
  query(Y) :-
    cst(X),
    reachable(X,Y).
{
  query('PartDieu'),
  query('Perrache')
}
Info: 2 tuples computed.
```

Sur notre interpréteur :

```
EDB
link(Charpennes,Perrache)
link(PartDieu,Charpennes)
link(Debourg,PartDieu)
link(PartDieu,Debourg)
cst(Charpennes)
cst(Debourg)

IDB
metro($x)
reachable($x,$y)
query($y)

MAPPING
link($x,$y) -> metro($x) .
link($x,$y) -> metro($y) .
link($x,$y) -> reachable($x,$y) .
cst($x) , reachable($x,$y) -> query($y) .
```

```
Query answer : [[Perrache], [PartDieu]]
Query subgoals : [reachable(Charpennes,Perrache), reachable(Debourg,PartDieu), query(Perrache), query(PartDieu)]
All details : ans : [reachable(Charpennes,Perrache), reachable(Debourg,PartDieu), query(Perrache), query(PartDieu)]  , inputByRule : {[head : quer
```

Exemple 5 :

Sur DES avec le Dataset 1 bis :

```
DES> query(Y) :- cst(X), reachable(X,Y).

Info: Processing:
  query(Y) :-
    cst(X),
    reachable(X,Y).
{
  query('Charpennes'),
  query('Debourg'),
  query('PartDieu'),
  query('Perrache')
}
Info: 4 tuples computed.
```

Sur notre interpréteur :

```
EDB
link(Charpennes,Perrache)
link(PartDieu,Charpennes)
link(Debourg,PartDieu)
link(PartDieu,Debourg)
cst(Charpennes)
cst(Debourg)

IDB
metro($x)
reachable($x,$y)
query($y)

MAPPING
link($x,$y) -> metro($x) .
link($x,$y) -> metro($y) .
link($x,$y) -> reachable($x,$y) .
link($z,$y) , reachable($x,$z) -> reachable($x,$y) .
cst($x) , reachable($x,$y) -> query($y) .
```

```
Query answer : [[Perrache], [PartDieu], [Charpennes], [Debourg]]
Query subgoals : [reachable(Charpennes,Perrache), reachable(Debourg,PartDieu), reachable(Debourg,Charpennes), reachable(Debourg,Debourg), reachable(D
All details : ans : [reachable(Charpennes,Perrache), reachable(Debourg,PartDieu), reachable(Debourg,Charpennes), reachable(Debourg,Debourg), reachabl
```

Exemple 6 :

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

```
DES> query(Y) :- cst(X),reachable(X,Y).

Info: Processing:
  query(Y) :-
    cst(X),
    reachable(X,Y).
{
  query('Perrache')
}
Info: 1 tuple computed.
```

Sur notre interpréteur :

```
EDB
link(Charpennes,Perrache)
link(PartDieu,Charpennes)
link(Debourg,PartDieu)
link(PartDieu,Debourg)
cst(Charpennes)

IDB
metro($x)
reachable($x,$y)
query($y)

MAPPING
link($x,$y) -> metro($x) .
link($x,$y) -> metro($y) .
link($x,$y) -> reachable($x,$y) .
link($z,$y) , reachable($x,$z) -> reachable($x,$y) .
cst($x) , reachable($x,$y) -> query($y) .
```

```
Query answer : [[Perrache]]
Query subgoals : [reachable(Charpennes,Perrache), query(Perrache)]
All details : ans : [reachable(Charpennes,Perrache), query(Perrache)]  , inputByRule : {[head : query($y) [true]], [body : [reachable($x,$y) [true,
```

Exemple 7 :

Sur DES avec le Dataset 2 :

```
DES> resists(X,Y) :- weak(Y,X).

Info: Processing:
  resists(X,Y) :-
    weak(Y,X).
{
  resists(fire,grass),
  resists(grass,water),
  resists(water,fire)
}
Info: 3 tuples computed.
```

Sur notre interpréteur :

```
EDB
type(Grass)
type(Water)
type(Fire)
weak(Grass,Fire)
weak(Fire,Water)
weak(water,Grass)

IDB
resist($x,$y)

MAPPING
weak($x,$y) -> resist($y,$x).
```

```
Query answer : [[Fire, Grass], [Water, Fire], [Grass, Water]]
Query subgoals : [resist(Fire,Grass), resist(Water,Fire), resist(Grass,Water)]
All details : ans : [resist(Fire,Grass), resist(Water,Fire), resist(Grass,Water)]  , inputByRule : {[head : resist($y,$x) [true, true]], [body : [wea
```

Exemple 8 :

Sur DES avec le Dataset 2 :

```
DES> beats(X,Y) :- hasType(X,Z),weak(W,Z),hasType(Y,W).

Info: Processing:
  beats(X,Y) :-
    hasType(X,Z),
    weak(W,Z),
    hasType(Y,W).
{
  beats('Bulbasaur','Squirtle'),
  beats('Charmander','Bulbasaur'),
  beats('Squirtle','Charmander')
}
Info: 3 tuples computed.
```

Sur notre interpréteur :

```
EDB
type(Grass)
type(Water)
type(Fire)
weak(Grass,Fire)
weak(Fire,Water)
weak(Water,Grass)
hasType(Squirtle,Water)
hasType(Bulbasaur,Grass)
hasType(Charmander,Fire)

IDB
resist($x,$y)
beats($x,$y)

MAPPING
weak($x,$y) -> resist($y,$x).
hasType($x,$z), weak($w,$z), hasType($y,$w)-> beats($x,$y).
```

```
Query answer : [[Charmander, Bulbasaur], [Squirtle, Charmander], [Bulbasaur, Squirtle]]
Query subgoals : [beats(Charmander,Bulbasaur), beats(Squirtle,Charmander), beats(Bulbasaur,Squirtle)]
All details : ans : [beats(Charmander,Bulbasaur), beats(Squirtle,Charmander), beats(Bulbasaur,Squirtle)]  , inputByRule : {[head : beats($x,$y) [tr
```

Exemple 9 :

Sur DES avec le Dataset 2 :

```
DES> family(Y) :- cst(X), finallyEvolves(X,Y).

Info: Processing:
  family(Y) :-
    cst(X),
    finallyEvolves(X,Y).
{
  family('Charizard'),
  family('Charmeleon')
}
Info: 2 tuples computed.
```

Sur notre interpréteur :

```
weak(Water,Grass)
hasType(Squirtle,Water)
hasType(Bulbasaur,Grass)
hasType(Charmander,Fire)
evolves(Charmander,Charmeleon)
evolves(Charmeleon,Charizard)
evolves(Bulbasaur,Ivysaur)
evolves(Ivysaur,Venusaur)
evolves(Squirtle,Wartortle)
evolves(Wartortle,Blastoise)
cst(Charmander)

IDB
resist($x,$y)
beats($x,$y)
finallyEvolves($x,$y)
family($x,$y)

MAPPING
weak($x,$y) -> resist($y,$x).
hasType($x,$z), weak($w,$z), hasType($y,$w)-> beats($x,$y).
evolves($x,$y) -> finallyEvolves($x,$y).
evolves($z,$y),finallyEvolves($x,$z) -> finallyEvolves($x,$y).

cst($x), finallyEvolves($x,$y) -> family($y).
```

```
Query answer : [[Charmeleon], [Charizard]]
Query subgoals : [finallyEvolves(Charmander,Charmeleon), finallyEvolves(Charmander,Charizard), family(Charmeleon), family(Charizard)]
All details : ans : [finallyEvolves(Charmander,Charmeleon), finallyEvolves(Charmander,Charizard), family(Charmeleon), family(Charizard)]  , input
```

Exemple 10 :

Sur DES avec le Dataset 2 :

```
DES> typeEvolution(Y,Z) :- cst(X),finallyEvolves(X,Y),hasType(X,Z).

Info: Processing:
  typeEvolution(Y,Z) :-
    cst(X),
    finallyEvolves(X,Y),
    hasType(X,Z).
{
  typeEvolution('Ivysaur',grass),
  typeEvolution('Venusaur',grass)
}
Info: 2 tuples computed.
```

Sur notre interpréteur :

```
evolves(Charmander,Charmeleon)
evolves(Charmeleon,Charizard)
evolves(Bulbasaur,Ivysaur)
evolves(Ivysaur,Venusaur)
evolves(Squirtle,Wartortle)
evolves(Wartortle,Blastoise)
cst(Bulbasaur)

IDB
resist($x,$y)
beats($x,$y)
finallyEvolves($x,$y)
isFrom($x,$y)
typeEvolution($x,$y)

MAPPING
weak($x,$y) -> resist($y,$x).
hasType($x,$z), weak($w,$z), hasType($y,$w)-> beats($x,$y).
evolves($x,$y) -> finallyEvolves($x,$y).
evolves($z,$y),finallyEvolves($x,$z) -> finallyEvolves($x,$y).


cst($x),finallyEvolves($x,$y),hasType($x,$z)->typeEvolution($y,$z).
```

```
Query answer : [[Ivysaur, Grass], [Venusaur, Grass]]
Query subgoals : [finallyEvolves(Bulbasaur,Ivysaur), finallyEvolves(Bulbasaur,Venusaur), typeEvolution(Ivysaur,Grass), typeEvolution(Venusaur,Grass)]
All details : ans : [finallyEvolves(Bulbasaur,Ivysaur), finallyEvolves(Bulbasaur,Venusaur), typeEvolution(Ivysaur,Grass), typeEvolution(Venusaur,Gras
```