Projet annuel: Résolution de problèmes en logique en C++ sans unification

Matthias LAUNAY, Victor LELU--RIBAIMONT, Jérôme LENG, Timotei MALASHCHYTSKI

Enseignant référent : Jean-Michel RICHER

Master 2 Informatique - Année universitaire 2020/2021

Sommaire

I - Présentation

Objectif initial

L'objectif initial de ce projet était de développer un parser en C++ pour traduire un fichier en langage Prolog en un fichier C++ exploitable qui serait la représentation de la base de connaissances du fichier Prolog. Il se divise alors en deux grandes parties à savoir la lecture du fichier Prolog et l'écriture d'un fichier en C++.

Pour ce faire nous avons écris un fichier "*readFile.cpp*" contenant toutes les méthodes nécessaires à la lecture et au stockage des données du fichier Prolog, et un second fichier d'écriture appelé "*ecriture.cpp*". Ce fichier contient les différentes méthodes pour représenter les faits et les règles sous forme de classes et de méthodes. Lors de l'exécution d'un appel aux méthodes de ces fichiers, un fichier C++ est créé, il est nommé de la même manière que le fichier Prolog. Une fois ce fichier compilé et exécuté, il affiche la base de connaissances du fichier Prolog ainsi que les déductions qu'il réalise à partir des règles et faits du fichier Prolog.

Améliorations apportées

Ce projet a été traduit en une librairie qui est facilement importable dans n'importe quel projet C++, cette librairie permet:

- d'importer un fichier prolog pour le décrypter en liste de vecteur de prédicats et de règles (base de faits et de règles)
- manipuler les Bases de Faits et de Règles (**BdF&R**)
- ajouter ou modifier du contenu dans la base de connaissances
- de lire et modifier les BdF&R afin d'interagir avec le code Prolog traduit
- de générer le fichier C++ à partir des BdF&R
- de récupérer directement les solutions assimilés aux BdF&R

Ces nouvelles structures de données, les BdF&R , repensées pour la librairie, sont des améliorations de ce qui avait été fait pour l'objectif initial, elles sont plus adaptées pour le stockage des règles et des faits (prédicats) et permettent également une interaction plus facile et intuitive lors de l'utilisation de la librairie.

La librairie est constituée de "*readFile.cpp*" et "*ecriture.cpp*" pour ce qui est des fonctionnalités initiales, ensuite "*libprolog.cpp*" permet la traduction des anciennes structures de données utilisées en nouvelles définie dans "*libprolog.h*". A partir de là, le fichier contient les méthodes associées à toutes les fonctionnalités présentées plus tôt.

Ainsi les deux premières semaines de développement ont été consacrées au parsing du fichier Prolog et enfin les deux dernières semaines ont été consacrées au déploiement de la librairie.

II - Implémentation

Première approche

Le programme s'exécute via un script Bash qui prend en argument le fichier Prolog source. On exécute la commande "*./shell.exe fichier.pl"*. Ainsi nous obtenons les fichiers suivants à partir de la ligne de commande:

Entrée	Sorties
fichier.pl : Fichier source en Prolog	- *fichier.cpp* : Fichier C++ généré représentant la traduction du Prolog - *fichier.exe* : Fichier exécutable résultant de la compilation de "*fichier.cpp"* réalisant les déductions - Affichage : Exécution de "*fichier.exe"* affichant les faits et les règles déduites

Lors de la lecture à partir du fichier "*readFile.cpp*", les faits sont stockés dans un conteneur et les règles sont ordonnées en fonction de leurs dépendances puis stockées dans un autre conteneur. Les deux structures de données sont transmises au fichier d'écriture "*ecriture.cpp*" qui génère le code C++ en fonction de données reçues.

Exemple 1

Entrée	Déductions réalisées par le programme
animal(X):- homme(X). mortel(X):- animal(X). meurt(X):- mortel(X), empoisonne(X). empoisonne(X):- boit(X,Y),poison(Y). homme(socrate). homme(platon). ami(socrate,platon). ami(platon,socrate). boit(socrate,cigue). poison(cigue).	animal = (socrate) (platon) mortel = (socrate) (platon) empoisonne = (socrate) meurt = (socrate)

Dans l'exemple suivant, notre fichier contient les données suivantes:

- faits: homme, ami, boit, poison;
- règles: animal, mortel, empoisonne, meurt;
- constantes: socrate, platon, cigue.

Notre fichier *readFile.cpp* stockera dans un premier temps tous nos faits et nos règles dans des vecteurs distincts qui constitueront nos conteneurs par la suite. Nos constantes seront représentées par la classe *Object*.
Nos conteneurs correspondent donc ici à nos faits et à nos règles. Si on prend le fait "ami" d'arité 2 correspondant au nombre de paramètres dans notre prédicat, le conteneur *ami* contiendra une liste d'objets *Tuple2* qui possède un tableau d'*Object* de taille 2. Si le programme détecte un nouveau prédicat avec une arité différente, un nouvel objet *TupleX* sera créé avec *X* correspondant à l'arité du prédicat. La classe *TupleX* contiendra un tableau d'*Object* de taille X.

Nos règles utilisent le même principe et la même structure. Pour chaque règle, une méthode *regle_deduce()* est implémentée qui se basera sur le nombre de prédicats du corps de la règle. Par conséquent, si une règle est répétée plusieurs fois, il y aura autant de méthodes que de règles et seront numérotées dans l'ordre où elles ont été trouvées. Cette méthode recherche les correspondances entre les différentes variables au sein du corps de la règle, par exemple si on trouve plusieurs fois la variable Y comme dans la règle "empoisonne" de l'exemple 1 on vérifie l'égalité entre ces deux valeurs. Lorsque toutes les conditions sont respectées on ajoute alors le nouveau fait à la base de faits.
La dernière méthode générée est *deduction()* qui remplit les conteneurs représentant la base de faits. C'est également dans cette méthode que les appels aux méthodes *regle_deduce()* seront effectués ainsi que l'affichage de la base de connaissance à la fin de l'éxecution une fois les déductions réalisées.

Dans l'exemple 1, lors de la lecture le programme inverse l'ordre des règles "meurt" et "empoisonne" car "meurt" dépend de "empoisonne" qui est définie après.

Code généré pour la règle grand_pere(X,Z):- pere(X,Y), pere(Y,Z).	**Passage en librairie**

`void grand_pere_deduce1(){`
`  for(auto t1 : pere){`
`  for(auto t2 : pere){`
`  if(t1.x2() != t2.x1()) continue;`
`  grand_pere.push_back(Tuple2(t1[0],t2[1]));`
`  }`
`  }`
`}`
	Code dynamique et utilisation de la récursivité pour générer toutes les affectations possibles pour chaque règle.

Ce code teste toutes les possibilités d'affectation via des boucles imbriquées (1 par prédicat dans le corps de règle) et vérifie la condition d'égalité entre les 2 affectations de la valeur Y. Ce code est spécifique à une règle et non adaptable. Chaque règle à une fonction *deduce* qui lui correspond.	Une fonction unique capable de réaliser les déductions pour toutes les règles en simulant des boucles imbriquées grâce à la récursivité. Le résultat est donné directement au lieu de générer un nouveau code à exécuter.

Exemple 2

Exemple d'exécution du programme sur une de Base de connaissances plus conséquente.

Entrée	Sortie
pere(roger,michel). pere(michel,jean). pere(michel,mathieu). pere(jean,sarah). pere(jean,xavier). pere(xavier,thomas). mere(marie,michel). mere(elise,jean). mere(elise,mathieu). frere(roger,stephane). frere(jean,mathieu). soeur(sarah,xavier). soeur(anna,roger). grand_pere(X,Z):- pere(X,Y), pere(Y,Z). grand_pere(X,Z):- pere(X,Y), mere(Y,Z). oncle(Y,Z):- frere(X,Y), pere(X,Z). oncle(Y,Z):- frere(X,Y), mere(X,Z). tante(X,Z):- soeur(X,Y), pere(Y,Z). tante(X,Z):- soeur(X,Y), mere(Y,Z).	Listes des prédicats : pere = (roger,michel) (michel,jean) (michel,mathieu) (jean,sarah) (jean,xavier) (xavier,thomas) mere = (marie,michel) (elise,jean) (elise,mathieu) frere = (roger,stephane) (jean,mathieu) soeur = (sarah,xavier) (anna,roger) Déductions pour toutes les règles : grand_pere = (roger,jean) (roger,mathieu) (michel,sarah) (michel,xavier) (jean,thomas) oncle = (stephane,michel) (mathieu,sarah) (mathieu,xavier) tante = (sarah,thomas) (anna,michel)

Passage en librairie

La librairie se décompose en deux fichiers. Le premier fichier *“libprolog.h”* sera inclus dans les programmes utilisés par les utilisateurs. Il permet de faire appel aux différentes fonctions de la librairie. Le second fichier *“libprolog.a”* est une archive qui est générée à partir du fichier objet de *“libprolog.cpp”*. Ce dernier contient le code des fonctions de la librairie mais il n’est pas accessible à l’utilisateur. En effet seul *“libprolog.h”* et *“libprolog.a”* sont nécessaires pour pouvoir utiliser la librairie.

Pour faciliter la manipulation des règles et des faits d’un fichier prolog en C++ des structures ont été créés pour la librairie dans *“liprolog.h”*.

La structure **Predicate** permet de représenter un prédicat (que ce soit utilisé pour un fait ou dans une règle). Elle est composée du nom du prédicat et d’une liste de constantes.
Si on reprend le fait pere(roger,michel) de l’exemple 2 le nom sera pere et la liste de constantes sera [roger,michel] (**Predicate**(pere,[roger,michel])).
La structure **Rule** représente une règle de la BdR. Elle est composée d’un nom, d’une liste de variables et d’une liste de prédicats.
grand_pere(X,Z):- pere(X,Y), mere(Y,Z) de l’exemple 2 sera décomposé ainsi :

nom = grand_pere
variables = [X,Z]
prédicats = [**Predicate**(pere,[X,Y]), **Predicate**(mere,[Y,Z])]

Pour chacune des structures des getteurs et setteurs sont définis afin de modifier les différents arguments les composant. De plus, l’opérateur de sortie standard << a été redéfini afin d’afficher les règles et prédicats sous forme Prolog.
On peut créer un prédicat ou une règle en utilisant les constructeurs des structures ou en utilisant les méthodes *createPredicate* et *createRule* de la librairie.

Une instance de libprolog contient deux vecteurs correspondant respectivement à la base de faits et la base de règle Prolog : _predicats et _rules. Ces vecteurs constituent la base de connaissances et sont ceux que l’utilisateur va manipuler grâce aux fonctions de la librairie.

Utilisation

```cpp
#include "../lib/libprolog.h"
using namespace std;

int main(int argc, char **argv)
{
  libprolog l;
  l.readPl("example.pl"); //lit le fichier Prolog source
  //la base de connaissances est stockee dans des vector de Predicat et Regle

  l.generateCPP(); //genere le fichier cpp representant la base de connaissances

  l.solvePl(); //realise les deductions a partir de la base de connaissances

  //ajoute le fait pere(michel, jean) a la base de connaissances
  l.addPredicate("pere", 2, "michel", "jean");

  //exporte la base de connaissances actuelle sous forme Prolog
  l.exportPl("sortie.pl");

  return 0;
}
```

Compilation:

Il ne faut pas oublier de compiler avec l'archive "*libprolog.a".*

g++ -std=c++14 -c -o demo.o demo.cpp

g++ -std=c++14 -o demo.exe demo.o **../lib/libprolog.a**

Liste des fonctions

readPl(string filename)	Charge le fichier Prolog afin de remplir les BdF&R
generateCPP()	Génère le fichier CPP équivalent au prolog, basé sur les BdF&R
solvePl()	Réalise les déductions à partir de BdF&R, les stocke et les affiche
exportPl(string filename)	Génère un fichier prolog à partir de la BdF&R
vector<Predicate> getPredicates()	Retourne dans un vecteur les faits de la BdF
vector<Predicate> findPredicates(string nom)	A partir d'un string donné en argument, récupère dans un vecteur tous les faits du même nom
setPredicats(vector<Predicate> predicats)	Définit une nouvelle BdF
addPredicate(Predicate p)	Ajoute un fait à la BdF avec une structure Predicate en paramètre
addPredicate(string nom, int n, ...)	Ajoute un fait à la BdF avec son nom, le nombre de constantes qui compose le fait et les constantes séparées par des virgules en paramètre
Predicate createPredicate(string nom, int n, ...)	Retourne un prédicat avec créé à partir d'un nom, du nombre constantes qui compose le prédicat, et des constantes séparées par des virgules en paramètre
Predicate createPredicate(string nom, vector<string> constantes)	Retourne un prédicat créé à partir d'un nom et vecteur contenant les constantes qui le compose
vector<Rule> getRules()	Retourne dans un vecteur les règles de la BdR
setRules(vector<Rule> rules)	Définit une nouvelle BdR
addRule(Rule r)	Ajoute une règle à la BdR avec une structure Rule en paramètre
addRule(string nom, vector<Predicate> predicats, int n, ...)	Ajoute une règle à la BdR avec son nom, un vecteur de Predicate, le nombre de variables qui compose la règle et les variables séparées par des virgules en paramètre
Rule createRule(string nom, vector<Predicate> predicats, int n, ...)	Retourne une règle créée à partir d'un nom, d'un vecteur de Predicate, du nombre de variables qui compose la règle et des variables séparées par des virgules en paramètre
Rule createRule(string nom, vector<string> var, vector<Predicate> predicats)	Retourne une règle créée à partir d'un nom, d'un vecteur contenant les variables qui compose la règle et d'un vecteur de Predicate en paramètre

Fonction SolvePl

La fonction *solvePl* est celle qui résout le problème Prolog (réalise les déductions à partir de la base de connaissance). Cette fonction réalise les mêmes tests que les différentes fonctions *regle_deduce* que nous générons dans la première partie du projet. La différence est que maintenant il faut exécuter le code pour faire les déductions et non écrire une fonction C++ dans un nouveau fichier comme précédemment. Pour mettre en place cette fonction nous avons repris le même principe utilisé auparavant mais l'avons implémenté dynamiquement de sorte que la fonction puisse traiter tous les problèmes contrairement à la première approche qui écrivait une fonction de résolution par règle.
Pour ce faire, nous avons développé une fonction récursive qui permet de tester toutes les combinaisons possibles d'affectations sur une règle et nous utilisons une map pour associer les variables de la règle à leur position dans le corps de règle.

Exemple

Base de connaissances	**Combinaisons à tester pour la règle grandpere**
pere(michel,jean) pere(patrick,michel) pere(jacques, sophie) grand_pere(X,Z):-pere(X,Y), pere(Y,Z)	pere(michel,jean),pere(michel,jean) pere(patrick,michel),pere(michel,jean) pere(jacques,sophie),pere(michel,jean) pere(michel,jean),pere(patrick,michel) pere(patrick,michel),pere(patrick,michel) pere(jacques,sophie),pere(patrick,michel) pere(michel,jean),pere(jacques,sophie) pere(patrick,michel),pere(jacques,sophie) pere(jacques,sophie),pere(jacques,sophie)

Cet exemple donne une déduction : grand_pere(patrick,jean) qui correspond à la deuxième possibilité du tableau ci-dessus.
Pour arriver à ce résultat nous faisons correspondre la position des variables X, Y, Z avec leur nom dans une map. Une position est de la forme x[Y], avec x l'indice du prédicat dans le corps de règle et Y l'indice de la variable dans le prédicat x.

Variable	**Position dans le corps de règle grand_pere**
X	1[0]
Y	1[1], 2[0]
Z	2[1]

On trouve la variable Y à deux positions dans le corps de règle donc on teste l'égalité entre les 2 affectations faites par la fonction récursive. Si toutes les conditions d'égalité de la règle sont vérifiées les affectations valides sont reportées à leur position dans la tête de règle, on peut faire une déduction et ajouter le nouveau fait à la base de faits.

III - Conclusion

En l'état, il manque au programme quelques fonctionnalités du Prolog. En effet, notre programme s'applique seulement sur des opérations simples sans unification. On pourrait aller plus loin dans la réflexion en intégrant notamment:

- la gestion de la négation
- la gestion des listes
- la gestion de l'opérateur "_"
- la gestion des fonctions intégrées à prolog
- la gestion d'opérations plus complexes comme par exemple : c(f(a),g(a,b))

Les autres améliorations possibles sont moins importantes. Il est possible d'optimiser les temps d'exécution avec de la parallélisation par exemple. Il est également possible de convertir les structures de données utilisées dans "*readFile.cpp*" et "*ecriture.cpp*" en nouvelles définies dans la librairie, afin d'avoir plus de cohérence entre les différentes méthodes dans différentes classes.
On peut également imaginer une interface graphique permettant de manipuler le programme sans avoir à faire du code.

Pour conclure nous pouvons dire que nous avons réussi à mener ce projet à son terme car nous avons répondu à toutes les demandes et avons produit une librairie pouvant s'intégrer à tout type de projet. De plus la phase de développement nous a permis de confirmer et améliorer nos compétences en C++, on peut citer notamment le développement d'une fonction récursive comme point technique intéressant.