Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

ProloGraal 2
Rapport de projet

Projet de semestre 6
I-3
Printemps 2020

Tony Licata
tony.licata@edu.hefr.ch

Responsable:
Frédéric Bapst
frederic.bapst@hefr.ch

8 mai 2020

Table des matières

1 Introduction

Ce chapitre consiste à introduire le projet. Des explications sur le contexte, les objectifs et le but du projet y sont détaillées.

1.1 Contexte

GraalVM est une machine virtuelle basée sur la Java Virtual Machine permettant d'exécuter du code dans différent langages lors d'une même exécution, celle-ci promettant de potentiels gros gains de performance comparé à un contexte d'exécution classique. Cela nécessite cependant qu'une implémentation spécifique pour chaque langage exécuté soit effectuée. Il existe alors des utilitaires pour faciliter l'implémentation d'un tel langage, Truffle est l'un de ces outils.
Alors que de nombreux langages populaires sont déjà implémentés pour GraalVM (C, Python, JS, Ruby, Java, etc...), aucune tentative publiée n'a été faite pour le langage de programmation logique Prolog. Bien que le fonctionnement du langage Prolog est fondamentalement bien différent d'un langage de programmation impératif tel Java ou C, il est tout à fait possible d'implémenter un tel langage logique pour GraalVM.
Lors d'un précédent projet de semestre, réalisé en automne 2019-2020 par Monsieur Spoto, un interpréteur Prolog pour GraalVM a été implémenté avec Truffle. L'interpréteur réalisé par M. Spoto permet alors d'exécuter du code Prolog au sein de GraalVM. Cet interpréteur est nommé ProloGraal, et c'est ainsi qu'il sera désigné dans le reste du document.
Alors que ProloGraal nous permet de compiler des règles Prolog et ensuite de questionner un interpréteur par rapport à ces règles, comportement classique de la console originale GNU-Prolog, il ne profite pas des outils d'optimisations et d'interopérabilité offerts par GraalVM.
Rendre ce langage interopérable signifie qu'il serait possible d'aisément profiter des avantages du langage Prolog au sein d'un programme polyglot tournant sur GraalVM. L'implémentation d'une telle fonctionnalité serait un avancement majeur pour le langage ProloGraal, le rendant potentiellement utile pour certaines tâches concrètes.

1.2 Objectifs

L'objectif est alors de rendre ProloGraal interopérable. Pour l'instant, ProloGraal ne permet que de questionner l'interpréteur sur les règles chargées au démarage du programme. Une réctification de certains mécanismes du langage est alors à effectuer pour permettre au langage d'exécuter des buts prologs en dehors de cet intepréteur. Il sera également nécessaire de modifier les objets propres au langage ProloGraal pour les rendre interopérables et utilisables dans d'autres langages GraalVM.
Ne sachant alors le temps de travail que cela représente, il a été décidé d'également implémenter certaines fonctionnalités du langage Prolog non-présentes dans ProloGraal. La possibilité d'effectuer un calcul (prédicat is/2) était par exemple absente, pourtant très utile pour réaliser un grand nombre de comportements voulu pour un programme Prolog. L'implémentation du calcul

permettrait également de tester les potentiels gains de performances promis par GraalVM dans le futur. Voici d'autres fonctionnalités désignées pour l'implémentation :

- prédicat trace/0 et notrace/0
- prédicat consult, permettant de charger des règles en cours d'exécution
- prédicat useinterpreter/0, prédicat additionnel permettant de pouvoir lancer l'interpréteur, ceci n'étant plus le comportement de base du langage
- modification du comportement des prédicats built-in, ne permettant actuellement pas de provoquer des échecs Prolog.

1.3 Déroulement

Ce projet est planifié sur deux différentes phases :

- Une première phase consiste à prendre en main les différentes technologies du projet, comprenant alors l'implémentation des prédicats trace/0 et is/2. Le site web de GraalVM met également à disposition un projet Java + Maven, nommé SimpleLanguage, expliquant à travers le code comment implémenter un nouveau langage dans Graalvm via Truffle. Ce projet sera également étudié durant cette phase pour mieux comprendre comment implémenter l'interopérabilité d'un langage avec Truffle.
- La seconde phase se concentre alors sur l'implémentation de l'intéropérabilité de ProloGraal. Une modification considérable de l'infrastructure de ProloGraal sera nécessaire pour permettre de le rendre interopérable.

1.4 Gestion du projet

Ce projet est supervisé par le Professeur Frédéric Bapst. Tout au long du projet, des séances hebdomadaires sont organisées pour communiquer les avancées mais également pour demander conseil sur les différents choix d'implémentation.
Le repository GitLab du projet créé par M. Spoto a été forké pour la continuation du projet. La même arborescence a été conservée pour les documents administratifs. Le lien [1] vers nouveau repository GitLab est présent en note de bas de page.
Un cahier des charges a été fourni en début de projet pour clairement fixer les objectifs à atteindre. Ce document est accompagné d'un planning sous forme de diagramme de Gantt. Ces documents sont présents sous `./docs/specs/semestre6/`. Le planning est disponible sous deux versions différentes, et ne représentent pas le travail effectué lors des semaines P10-P12. Lors de ces semaines, le travail effectué se concentrait encore sur l'interopérabilité.
Des procès verbaux sont réalisés à chaque séance pour garder trace de ce qui est dit. Ils sont soumis au contrôle pour assurer l'exactitude des informations y figurant. Ces documents sont présents sous `./docs/pv/semestre6/`.

1. Martin Spoto & Tony Licata. *ProloGraal Repository.* URL : `https://gitlab.forge.hefr.ch/tony.licata/prolog-truffle/`.

1.5 Structure du rapport

Le rapport se concentre sur les différents mécanismes et fonctionnalités implémentés dans ProloGraal. Les chapitres expliquent comment ont été implémentées les différentes nouvelles fonctionnalités, excepté le chapitre d'analyse de SimpleLanguage est lui plus axé théorique, avec des explications sur différents aspects de l'interopérabilité au sein de SimpleLanguage.

2 Technologies du projet

Cette section décrit les différentes technologies utilisées au sein de ce projet.

2.1 GraalVM

GraalVM, comme énoncé dans le contexte, est une machine virtuelle Java modifiée permettant de compiler du code polyglotte. Elle permet d'exécuter des programmes de différentes manières, comme par exemple exécuter des applications compilées en code natif [2]. Nous allons plutôt utiliser le SDK fourni par l'équipe GraalVM, plus précisément le SDK 19.2.0.1, pour l'intégrer au projet ProloGraal et Truffle, étant dépendant de celui-ci [3].

2.2 Truffle

Truffle est un utilitaire nous assistant dans la création d'un langage pour GraalVM. Dans le cadre de ce projet, Truffle est chargé en tant que librairie via Maven, grâce au pom.xml du projet.
Truffle est l'API qui permettra de rendre ProloGraal interopérable. Grâce à des mécanismes tel que l'InteropLibrary, il permet alors de rendre des objets spécifique à un langage d'être générique du point de vue de GraalVM. L'étude de ses mécanismes est alors un point crucial du projet.

2. GraalVM Team. *Getting started with GraalVM.* URL : `https://www.graalvm.org/getting-started/`.
3. GraalVM Team. *Github release of JVM 19.2.0.1.* URL : `https://github.com/oracle/graal/releases/tag/vm-19.2.0.1/`.

3 Prise en main

Ce chapitre met en avant les différentes prédicats intégrées à ProloGraal qui n'influent pas directement sur l'interopérabilité. Bien que ces prédicats ne résolvent pas l'objectif principal du projet, leurs implémentations permettent une familiarisation avec l'environnement ProloGraal.

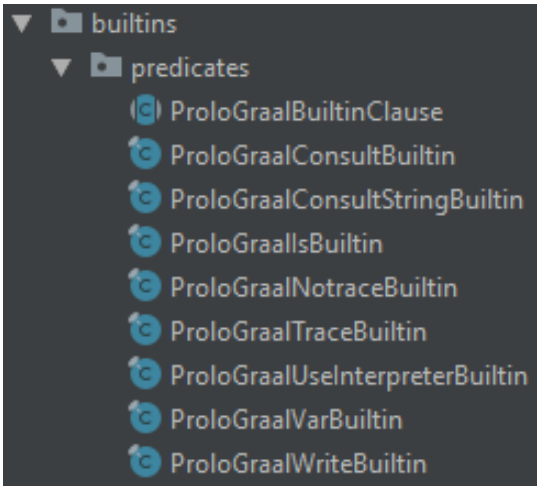
3.1 Besoins

Voici les besoins nécessaires pour cette phase :

- Mise en place du SDK GraalVM 19.2.0.1
- Installation de ProloGraal

3.2 Implémentation d'un prédicat built-in

Les deux prédicats implémentés dans cette section de prise en main sont des prédicats built-in. Le prototype fourni de ProloGraal possède alors une démarche pour simplement implémenter un nouveau prédicat built-in, et nécessite alors la compréhension de certains mécanismes de ProloGraal.
Voici les classes java représentant un prédicat built-in au sein de ProloGraal :



On y voit déjà les futurs prédicats à implémenter, cette représentation étant l'état final des prédicats après réalisation du projet. Cette image nous montre également la classe abstraite ProloGraalBuiltinClause, classe que chaque nouveau prédicat implémenté se devra d'étendre au sens Java.

3.2.1 ProloGraalBuiltinClause

Voyons le contenu de cette classe :

```
public abstract class ProloGraalBuiltinClause extends
   ProloGraalClause {
   /** Overridable execute method for built-ins.
    *  May have side effects, like writing or opening a file.
    */
   public ProloGraalBoolean execute(){
      //default behaviour is to output a success
      return new ProloGraalSuccess(getVariables());
   };

   //...
}
```

La méthode non-présente (représentée via le commentaire `//...`) est une méthode permettant de s'assurer du bon fonctionnement du prédicat mais n'est pas au coeur du fonctionnement d'une classe built-in.
La méthode `execute()` est la méthode qui s'exécute lorsque l'on questionne ProloGraal avec un prédicat built-in. On peut voir que le type de retour est un `ProloGraalBoolean`, représentant un succès ou échec Prolog. ce type de retour a du être modifié, car précédemment void (ce qui empèchait aux prédicats built-in de pouvoir provoquer des échecs, et qui sera par exemple nécessaire avec l'implémentation du prédicat is/2).

3.2.2 Modification du ProloGraalRuntime

Lors de la résolution d'une question Prolog, ProloGraal s'aide alors d'un objet Runtime contenant les règles actuellement chargées pour la résolution d'un but. Vu qu'un prédicat built-in n'a pas besoin d'être spécifié en tant que règle valide dans un environnement Prolog, il faut que ce prédicat soit chargé dans la liste des règles à la création du Runtime. Cela nécessite donc de modifier la classe `ProloGraalRuntime`, classe représentant le Runtime de ProloGraal.
Voyons le contenu de la classe `ProloGraalRuntime` nous intéressant pour l'implémentation des nouveaux prédicats :

```
public final class ProloGraalRuntime {
    //...

   /**Install built-ins predicate. Every new built-in should be
      added here.
    */
   private void installBuiltins() {
      //...
```

```
        ProloGraalClause traceBuiltin = new
           ProloGraalTraceBuiltin(context);
        clauses.put(traceBuiltin.getHead(),
           Collections.singletonList(traceBuiltin));

        ProloGraalClause notraceBuiltin = new /*...*/;
        clauses.put(/*...*/);

        ProloGraalClause isBuiltin = new /*...*/;
        clauses.put(/*...*/);

        ProloGraalClause useinterpreterBuiltin = new /*...*/;
        clauses.put(/*...*/);

        ProloGraalClause consultStringBuiltin = new /*...*/;
        clauses.put(/*...*/);

        ProloGraalClause consultBuiltin = new /*...*/;
        clauses.put(/*...*/);
    }
}
```

On peut alors voir tous les nouveaux prédicats built-in ajoutés lors de la réalisation de ce projet. Les spécificités d'implémentation de chaque prédicat sont décrites dans leurs sections dédiées dans la suite du rapport. Cette méthode est automatiquement exécutée dans le constructeur du `ProloGraalRuntime`.
D'autres modifications ont été effectuées sur le comportement du `ProloGraalRuntime`, notamment pour pouvoir ajouter des règles à la volée. Quelques précisions sur ces changements sont disponibles dans la section `Restructuration de ProloGraal` du rapport.

3.3 Implémentation du prédicat trace

Le prédicat trace/0 en Prolog est un prédicat activant une sorte de debugger. Ce debugger verbeux va alors écrire sur le terminal le chemin emprunté par l'interpréteur Prolog dans l'arbre de preuve pour résoudre notre question Prolog. Cela nécessitera alors d'également modifier certains compostants déjà mis en place, comme par exemple l'arbre de preuve ProloGraal. Le prédicat notrace/0 sera également implémenté pour désactiver le debugger.

3.3.1 Mise en place

Le prédicat trace/0 est implémenté en tant que prédicat built-in dans ProloGraal. L'implémentation d'un tel prédicat suit une certaine procédure déjà mise en place lors du précédant projet. Cette procédure est décrite dans la section `Implémentation d'un prédicat built-in`

présente précédemment.
Il est également nécessaire d'implémenter le constructeur de la classe. Voici le contenu de la classe `ProloGraalTraceBuiltin` spécifique au constructeur :

```
public final class ProloGraalTraceBuiltin extends
   ProloGraalBuiltinClause {

   private final PrintWriter writer; // used for outputting
   private final ProloGraalContext context; // we keep the
      context to use it later
   private static final String TRACE_ON_TEXT = /*...*/;

   public ProloGraalTraceBuiltin(ProloGraalContext context) {
      super();
      // get printer from context
      this.writer = new PrintWriter(context.getOutput(), true);
      this.context = context;

      // create the head of this clause
      // since we do not need unification, an atom is enough
      ProloGraalAtom head = new
         ProloGraalAtom(getVariables(),"trace");

      setHead(head);
   }

   //...
}
```

On initialise premièrement les valeurs des attributs writer et context qui seront utilisées plus tard par le prédicat. Par la suite, on set la tête de notre règle (via la méthode héritée `setHead()`) comme `ProloGraalAtom` nommé `"trace"`. Cette action permet de concrêtement créer le prédicat, ce qui nous permettra par la suite de l'interpréter comme un fait Prolog (tête de règle mais pas de but), et donc d'unifier les prédicats trace/0 parsés avec ce prédicat présent dans les règles admises du Runtime.

3.3.2 Méthode execute

Lors d'une requête trace/0, le prédicat builtin va être exécuté. En interne, la méthode `execute()` de la classe `ProloGraalTraceBuiltin` sera exécutée. Voici le contenu de cette méthode ainsi que les attributs qu'elle utilise :

```
public final class ProloGraalTraceBuiltin extends
   ProloGraalBuiltinClause {

   private final PrintWriter writer; // used for outputting
   private final ProloGraalContext context; // we keep the
      context to use it later
   private static final String TRACE_ON_TEXT = "The debugger
      will first creep -- showing everything (trace)";

   //...

   /**Execute the trace, even if the trace is already ON (like
      usual Prolog behaviour).
    */
   @Override
   public ProloGraalBoolean execute() {
      context.setTraceFlag(true);
      writer.print(TRACE_ON_TEXT);
      writer.flush();
      return new ProloGraalSuccess(getVariables());
   }

   //...
}
```

Le context ProloGraal est le context actuel d'exécution, contenant entre autre la liste des règles (Runtime) mais également d'autres éléments importants du langage. Ce context est présent dans la totalité des classes, et il semble alors une bonne stratégie de le modifier pour que l'exécution change de comportement (debugger verbeux).
Il contient alors un nouveau boolean faisant office de traceflag, que l'on va set à `true`. On output par la suite le texte usuellement lisible lorsque l'on requête trace/0 dans la console Prolog officielle, et retournons un succès ProloGraal (l'exécution du prédicat trace/0 ne peut usellement pas causer d'échec).

3.3.3 Modification de l'arbre de preuve

Bien que le traceflag a été set à true, le comportement permettant de simuler un trace usuel n'est pas implémenté. En effet, lors du parcours de l'arbre de preuve, les décisions effectuées par l'interpréteur prolog doivent être affichées à l'écran. Nous allons donc modifier le comportement de l'actuel `ProloGraalProofTreeNode`.
La construction de l'arbre de preuve s'effectue via le noeud racine, il n'y a donc pas besoin d'objet arbre, mais simplement d'instancier le noeud racine avec le bon but à réaliser. Plutôt

que de s'intéresser à l'initialisation du noeud racine, regardons directement la méthode execute du `ProloGraalProofTreeNode`, méthode permettant le parcours et la création de l'arbre :

```
public ProloGraalObject execute(Deque<Integer> branches) throws
   ProloGraalExistenceError {
   /*
     initialisation of currentGoal
   */
   if (traceFlag) {
      System.out.println(depth + (start+1) + TRACE_CALL_TEXT +
         currentGoal.toString() +  TRACE_QUESTION_MARK);
   }
   /*
     initialisation of unifiableClauses
   */
   for (int i = start; i < unifiableClauses.size(); i++) {
      // no need to copy here since it is already one
      ProloGraalClause unifiableClause = unifiableClauses.get(i);
      /*
        initialisation of newGoals
      */
      ProloGraalObject result = new
         ProloGraalProofTreeNode(clauses, newGoals, depth+1,
         traceFlag).execute(branches);
      if (result instanceof ProloGraalBoolean &&
         ((ProloGraalBoolean) result).asBoolean()) {
         if (traceFlag) {
            System.out.println(depth  + (start+1) +
               TRACE_EXIT_TEXT + currentGoal  +
               TRACE_QUESTION_MARK);
         }
         return new ProloGraalSuccess();
      }
   }
   return new ProloGraalFailure();
}
```

On voit alors dans ce code que lorsqu'un nouveau but est exécuté et que le traceFlag est à true, on annonce alors qu'on appel ce but. lorsqu'on rencontre un succès, on annonce que l'on sort du but courrant. Cette manière de faire provoque effectivement l'effet d'un trace Prolog, les variables `static final TRACE_...` permettent d'obtenir un résultat semblable au traceur traditionnel. Les variables depth et start permettent de mémoriser et afficher la profondeur

actuelle de l'arbre de preuve ainsi que la branche actuellement visitée.

3.3.4 prédicat notrace/0

Il est alors nécessaire d'également implémenter le prédicat notrace/0 pour pouvoir désactiver le debugging. On passe l'étape de sa mise en place, car identique à celle du trace/0.
Penchons-nous simplement sur la méthode `execute()` permettant de set le traceflag à false :

```
private static final String TRACE_OFF_TEXT = "The debugger is
   switched off";
public ProloGraalBoolean execute() {
    context.setTraceFlag(false);
    writer.print(TRACE_OFF_TEXT);
    writer.flush();
    return new ProloGraalSuccess(getVariables());
}
```

L'arbre de preuve arrête alors de détailler sa résolution.

3.3.5 Résultats

Voici un exemple d'interaction avec l'interpréteur ProloGraal utilisant les nouveau prédicats trace/0 et notrace/0 :

```
"C:\Program Files\Java\graalvm-ce-19.2.0.1\bin\java.exe" ...
?- trace.

The debugger will first creep -- showing everything (trace)

yes
?- t(A).

    1   1 Call: t/1(A) ?
    2   1 Call: test/1(X = A) ?
    2   1 Exit: test/1(X = 8) ?
    1   1 Exit: t/1(A = 8) ?

A = 8

yes
?- notrace.

    1   1 Call: notrace ?
The debugger is switched off
    1   1 Exit: notrace ?


yes
?- t(A).


A = 8

yes
```

On peut voir que le debugger est encore activé lors de la résolution du prédicat notrace/0, mais n'est plus activé pour la requête qui suit. La profondeur actuelle est correctement affichée, mais le système de branche ne fonctionne actuellement pas (causé par la manière dont le noeud de l'arbre de preuve résoud ses buts).

3.4 Implémentation du prédicat is/2

Le prédicat is/2 en Prolog nous permet d'effectuer des calculs et de stocker le résultat dans une variable non-assignée. Il permet également d'assurer l'égalité entre un nombre/variable bindée en nombre et un calcul. La section `Méthode execute` décrit l'implémentation permettant de mimer ce comportement.

3.4.1 Mise en place

Le prédicat is/2 est implémenté en tant que prédicat built-in dans ProloGraal. L'implémentation d'un tel prédicat suit une certaine procédure déjà mise en place lors du précédant projet.

Cette procédure est décrite dans la section `Implémentation d'un prédicat built-in` présente précédemment.
Le constructeur du prédicat is/2 se doit de préparer les arguments qu'il recevra. Ces arguments sont comme décrit précédemment une variable/nombre et un calcul. voici le constructeur du prédicat :

```
private ProloGraalVariable arg; // the variable A in is(A,B). We
   keep it to use it in the execute method
private ProloGraalVariable arg2; // the variable B in is(A,B).
   We keep it to use it in the execute method

private final Map<ProloGraalAtom, BiFunction<ProloGraalNumber,
   ProloGraalNumber, ProloGraalNumber>> operations;

public ProloGraalIsBuiltin(ProloGraalContext context) {
   super();

   // we fill the operations map to check the operator used by
      the user later in the execute method
   operations = new HashMap<>();
   operations.put(new ProloGraalAtom(getVariables() , "'+'"),
      (a,b) -> new ProloGraalDoubleNumber(getVariables() ,
      a.asDouble() + b.asDouble()));
   operations.put(/*'-'*/,/*minusLambda*/);
   operations.put(/*'*'*/,/*productLambda*/);
   operations.put(/*'/'*/,/*divisionLambda*/);
   operations.put(/*'**'*/,/*powerLambda*/);
   operations.put(/*'mod'*/,/*modLambda*/);

   // create the head of this clause
   // since we do not need custom unification, a simple
      structure is enough
   ProloGraalStructure head = new
      ProloGraalStructure(getVariables());
   head.setFunctor(new ProloGraalAtom(getVariables(), "is"));

   // we create and store the variables to access them more
      easily later in the execute method
   arg = new ProloGraalVariable(getVariables(), "_");
   head.addSubterm(arg);
```

```
    arg2 = new ProloGraalVariable(getVariables(), "op");
    head.addSubterm(arg2);

    setHead(head);
}
```

Grâce à l'initialisation des arguments (attributs Java) en tant que variable et a l'assignation de ces variables en tant qu'arguments de la tête de règle (via la méthode `addSubterm()`), il sera possible dans la méthode `execute()` d'accéder aux objets ProloGraal avec lesquels les arguments de la tête de règle se seront unifiés (et donc accessible via les attributs `arg` et `arg2`). On voit également dans le code que les opérations valides pour le prédicat is/2 sont ajoutées à un `HashMap` nommé `operations`. il est alors possible d'ajouter un nouvel opérateur pour autant que celui prenne deux arguments. Ce défaut de conception temporaire est du à l'utilisation de la `BiFunction` utilisée en tant que valeur du `HashMap`. Pour ajouter un nouvel opérateur, il faut alors définir, par exemple, une fonction lambda qui exécutera l'opération désirée. Cette fonction possédera également deux arguments.

3.4.2 Méthode execute

La méthode `execute()` permet d'initialiser le fonctionnement du prédicat is/2. Voici le code relatif à la méthode `execute()`

```
private ProloGraalVariable arg; // already bound
private ProloGraalVariable arg2; // already bound
public ProloGraalBoolean execute() {
    ProloGraalTerm leftTerm = arg.getRootValue();
    ProloGraalTerm rightTerm = arg2.getRootValue();
    if(leftTerm instanceof ProloGraalVariable || leftTerm
       instanceof ProloGraalNumber){
        ProloGraalNumber result = consultTerm(rightTerm);
        if(result==null){
            writer.print("op nok");
            writer.flush();
            return new ProloGraalFailure();
        }
        if(leftTerm.unify(result)){
            return new ProloGraalSuccess(getVariables());
        } else {
            return new ProloGraalFailure();
        }
    }else{
        writer.print("left term nok");
```

```
        writer.flush();
        return new ProloGraalFailure();
    }
}
```

On voit qu'elle s'assure premièrement que le terme de gauche du prédicat is/2 est bien un nombre ou une variable. Si ce n'est pas le cas, elle signale à l'interpréteur que le terme de gauche n'est pas valide et retourne une nouvelle `ProloGraalFailure`. Sinon, la méthode consulte le terme de droit via une méthode `consultTerm`, car celui-ci peut autant être un nombre qu'un calcul (un calcul effectué n'est jamais qu'un nombre).
Cette méthode nous retournera donc un `ProloGraalNumber` si le terme de droit est valide, sinon null. Le prédicat informe également l'utilisateur si le terme de droit est invalide et retourne une `ProloGraalFailure`.

3.4.3 Méthode consultTerm

Le calcul qui peut être fournis dois posséder un format strict non énoncé jusqu'à présent. Le calcul doit alors être présenté sous la forme d'un prédicat à deux arguments possédant comme foncteur une opérande. Les arguments peuvent à leur tour être un nouveau calcul, ou un nombre. Le calcul 3 + 5 * 3 devra alors s'écrire '+'(3,'*'(5,3)). Il faut alors effectuer un tri pour chacun des éléments, à commencer par le prédicat parent '+'/2 qui sera déterminé en tant qu'opérateur. Le code suivant montre ce comportement :

```
private ProloGraalNumber consultTerm(ProloGraalTerm term){
    ProloGraalNumber number = null;
    if(term instanceof ProloGraalNumber){
        number = (ProloGraalNumber)term;
    }else if(term instanceof ProloGraalVariable &&
        ((ProloGraalVariable) term).isBound() &&
        term.getRootValue() instanceof ProloGraalNumber){
        number =
            (ProloGraalNumber)((ProloGraalVariable)term).getRootValue();
    }else if(term instanceof ProloGraalStructure &&
        ((ProloGraalStructure) term).getArguments().size()==2){
        number = consultOperation((ProloGraalStructure)term);
    }
    return number;
}
```

On voit que les deux premières conditions permettent de retourner le terme lui-même si celui-ci est un nombre, ou sa valeur bindée si celui-ci est une variable unifiée à un nombre. La condition `term instanceof ProloGraalStructure && ...` s'assure que term est un prédicat, et que celui-ci possède deux arguments. Si cela est vérifié, on délègue alors le traitement de

l'opération à une autre méthode `consultOperation()`. Celle-ci appellera deux fois la méthode `consultTerm()` pour décréter comment traiter les deux arguments de l'opération.

3.4.4 Méthode consultOperation

Cette méthode privée du prédicat is/2 est exécutée lorsque la méthode `consultTerm()` détermine qu'elle consulte une opération. On sait alors que l'objet ProloGraal que l'on reçoit est une `ProloGraalStructure`, soit un prédicat.
Voici le contenu de la méthode `execute()` :

```
private ProloGraalNumber consultOperation(ProloGraalStructure
   operation){
    ProloGraalTerm leftTerm = operation.getArguments().get(0);
    ProloGraalTerm rightTerm = operation.getArguments().get(1);

    ProloGraalNumber leftNumber = consultTerm(leftTerm);
    ProloGraalNumber rightNumber = consultTerm(rightTerm);
    if(leftNumber == null || rightNumber == null) return null;

    return computeOperation(operation.getFunctor(), leftNumber,
       rightNumber);
}

private ProloGraalNumber computeOperation(ProloGraalAtom functor,
                              ProloGraalNumber leftNumber,
                                 ProloGraalNumber rightNumber){
    if(operations.containsKey(functor)){
       return operations.get(functor).apply(leftNumber,
          rightNumber);
    }
    return null;
}
```

On voit alors que les deux arguments de l'opérateur sont récupérés, et sont ensuite consultés à leur tour pour en récupérer des nombres, car ces arguments peuvent initialement aussi bien être des nombres ProloGraal que des opérations. Si ces arguments évalués ne sont pas égaux à null, cela signifie qu'ils possédaient une valeur valide, on peut donc exécuter l'opération désirée sur les deux arguments ainsi transformés en `ProloGraalNumber`.
La méthode privée compute Operation est donc appelée avec le foncteur de l'opération courante, et va retourner les paramètres leftNumber et rightNumber avec l'opération voulue. Pour déterminer l'opération à effectuer, il suffit de demander au `HashMap operations` de nous retourner la `BiFunction` assossiée au foncteur passé en paramètre, foncteur de l'opération courrante.

Si la `BiFunction` est trouvée, la valeur calculée est donc retournée, sinon, null sera retourné.

3.4.5 Résultats

Voici un exemple d'interaction avec l'interpréteur ProloGraal utilisant le nouveau prédicat is/2 :

```
?- is(A,'+'(3,'*'('/'(4,2),7))), is(B,'^'(A,3)).


A = 17.0
B = 4913.0

yes
?- is(3,'+'(2,2)).



no
```

On constate alors que le prédicat gère effectivement les opérations imbriquées grâce à sa méthode récursive. Lors de la première requête, on observe que la variable A prend la valeur 17. Elle est également réutilisée dans le second calcul en lélevant au cube, opération valide car on lui a déjà attribué une valeur. La variable B prend donc la valeur 17^3.
On observe également que l'opérateur is/2 permet de vérifier si un calcul est correct. Ici, la question Prolog `is(3,'+'(2,2))`, qui peut être traduite par $3 \stackrel{?}{=} 2+2$, retourne un échec.

4 Interopérabilité

Cette section regroupe les apports faits à ProloGraal au niveau de l'interopérabilité. Alors que la section `Analyse du SimpleLanguage` est plutôt théorique et apporte des éclaircissements sur les mécanismes d'interopérabilité lié offerts par Truffle, les autres sections seront concentrés sur les apports interopérables à ProloGraal.

4.1 Analyse du SimpleLanguage

SimpleLanguage est un projet Java+Maven réalisé par l'équipe GraalVM. Le but du projet est d'expliquer comment créer un langage pour GraalVM via Truffle. De nombreuses explications commentées sont présentes à travers le code pour guider le lecteur dans sa compréhension de l'utilisation de Truffle.

4.1.1 Installation

SimpleLanguage est displonible sur GitHub [4]. Après avoir cloné le repository, il suffit de l'ouvrir sur un IDE support les projets Java+Maven pour pouvoir le modifier et le compiler comme bon nous semble. Il sera cependant nécessaire de supprimer cette ligne du fichier pom.xml si l'on compile le projet sur Windows, cette feature n'étant actuellement pas supportée :

```
<project <!-- namespace infos -->>
  <!-- ... -->
  <modules>
    <!-- ... -->
    <!-- <module>native</module> --> <!-- comment this line -->
    <!-- ... -->
  </modules>
</project>
```

L'IDE utilisé pour l'implémentation des nouvelles fonctionnalités est l'IDE IntelliJ [5] IDEA 2019.1.1. La version exacte utilisée semble absente, mais d'autres versions très proches sont disponibles.

4.1.2 Export d'objets SimpleLanguage

Les types primitifs de SimpleLanguage ainsi que sa définition des fonctions sont des objets générique vus par Truffle, car ils implémentent correctement les mécanismes les rendant interopérables. Voyons de plus près la classe `SLBigNumber`, classe représentant les nombres SimpleLanguage :

4. GraalVM Team. *Simple Language Repository.* URL : `https://github.com/graalvm/simplelanguage`.
5. JetBrains. *Download IntelliJ.* URL : `https://www.jetbrains.com/idea/download/other.html`.

```
@ExportLibrary(InteropLibrary.class)
public final class SLBigNumber implements TruffleObject,
   Comparable<SLBigNumber> {
    /* methods used to check if value fit in primitive type */
    private final BigInteger value;
    public SLBigNumber(BigInteger value) {
        this.value = value;
    }

    @ExportMessage
    boolean isNumber() {
        return fitsInLong();
    }

    @ExportMessage
    @TruffleBoundary
    boolean fitsInInt() {
        return value.bitLength() < 32;
    }

    @ExportMessage
    @TruffleBoundary
    int asInt() throws UnsupportedMessageException {
        if (fitsInInt()) {
            return value.intValue();
        } else {
            throw UnsupportedMessageException.create();
        }
    }
}
```

On peut y voir plusieurs mots-clés usuellement absents du language Java, dans leur ordre d'apparition :

- `@ExportLibrary(X)` : à placer au dessus d'une classe, permet de spécifier que notre classe doit s'exporter pour la librairie X.
- `InteropLibrary.class`[6] : librairie incluse dans Truffle permettant l'interopérabilité des objets spécifiques à chaque langage.
- `@ExportMessage` : à placer au dessus d'une méthode à exporter pour la librairie. Il est nécessaire que la méthode possède le même nom, les même paramètres ainsi que le même

6. GraalVM Team. *InteropLibrary JavaDoc*. URL : `https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/interop/InteropLibrary.html`.

type de retour que la méthode générique présente dans l'InteropLibrary.

— `@TruffleBoundary` [7] : annotation Truffle à placer judicieusement au dessus de certaines méthodes pour optimiser la compilation, en rapport avec l'inlining.

Les mots clés que nous retiendrons pour l'interopérabilité sont les trois premiers, car le mot clé `@TruffleBoundary` nécessite une application réfléchie, et bien qu'en lisant de la documentation à son sujet, son impact et comportement exact lors de la compilation est encore inconnu. Il est néanmoins important de garder cette annotation pour une amélioration future de ProloGraal. L'objectif sera alors d'implémenter ces mots clés actuellement absents du langage ProloGraal. Grâce à cela, on espère alors que les `ProloGraalObjects` exportés pourront répondre à des messages Truffle génériques.

4.1.3 Comportements interopérables du langage

D'autres comportements interopérables sont observables au sein de SimpleLanguage. Dans ce chapitre, différentes classes de SimpleLanguage sont analysées pour mieux comprendre par quels outils Truffle peut rendre un language interopérable.

4.1.3.1 SLFunction

On peut par exemple observer ceci dans la classe `SLFunction` :

```
@ExportLibrary(InteropLibrary.class)
public final class SLFunction implements TruffleObject {
    /** The name of the function. */
    private final String name;

    public static final int INLINE_CACHE_SIZE = 2;
    /**{@link SLFunction} instances are always visible as
       executable to other languages.
     */
    @SuppressWarnings("static-method")
    @ExportMessage
    boolean isExecutable() {
        return true;
    }
}
```

On voit alors que la méthode `isExecutable()` est exportée pour l'InteropLibrarie. Cela nous indique alors qu'il est possible de rendre les fonctions d'un langage utilisables à l'extérieur de celui-ci, via un appel polyglotte par exemple.

7. GraalVM Team. *TruffleBoundary JavaDoc*. URL : `https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/CompilerDirectives.TruffleBoundary.html`.

La classe `SLFunction` exporte également le mécanisme d'exécution, mais est assez complexe et non-pertinant dans ce contexte, donc enlevé du code montré.

4.1.3.2 SLAddNode

Analysons maintenant la classe `SLAddNode`, les commentaires importants sont laissés pour ajouter des informations supplémentaires et intéressantes pour ProloGraal et sont propriétaires de l'équipe GraalVM :

```
@NodeInfo(shortName = "+")
public abstract class SLAddNode extends SLBinaryNode {
    /**Specialization for primitive {@code long} values. This is
       the fast path of the
     * arbitrary-precision arithmetic. We need to check for
        overflows of the addition, and switch to
     * the {@link #add(SLBigNumber, SLBigNumber) slow path}.
        Therefore, we use an
     * {@link Math#addExact(long, long) addition method that
        throws an exception on overflow}. The
     * {@code rewriteOn} attribute on the {@link Specialization}
        annotation automatically triggers
     * the node rewriting on the exception.
     * In compiled code, {@link Math#addExact(long, long)
        addExact} is compiled to efficient machine
     * code that uses the processor's overflow flag. Therefore,
        this method is compiled to only two
     * machine code instructions on the fast path.
     * This specialization is automatically selected by the
        Truffle DSL if both the left and right
     * operand are {@code long} values. */
    @Specialization(rewriteOn = ArithmeticException.class)
    protected long add(long left, long right) {
        return Math.addExact(left, right);
    }

    @Specialization
    @TruffleBoundary
    protected SLBigNumber add(SLBigNumber left, SLBigNumber
       right) {
        return new
           SLBigNumber(left.getValue().add(right.getValue()));
    }
```

```
    /**Specialization for String concatenation. The SL
       specification says that String concatenation
     * works if either the left or the right operand is a
        String. The non-string operand is
     * converted then automatically converted to a String.
     * To implement these semantics, we tell the Truffle DSL to
        use a custom guard. The guard
     * function is defined in {@link #isString this class}, but
        could also be in any superclass. */
    @Specialization(guards = "isString(left, right)")
    @TruffleBoundary
    protected String add(Object left, Object right) {
        return left.toString() + right.toString();
    }

    protected boolean isString(Object a, Object b) {
        return a instanceof String || b instanceof String;
    }
}
```

On peut alors observer deux nouvelles annotations. Premièrement, l'annotation `NodeInfo()` permettet d'ajouter des informations supplémentaires utiles au debbugging au noeud, ici le noeud `SLAddNode`. Elle est ici utilisée pour ajouter un nom au noeud, nom utile uniquement pour faciliter le debugging du langage. Cela nous permet de voir qu'il est possible de spécifier des valeurs pour certaines des annotations offertes par Truffle.
La seconde est l'annotation `@Specialization()`. Cette annotation permet de spécifier au compilateur quelle méthode exécuter lorsque plusieurs méthodes possèdent le même nom (surcharge Java). Cela permet notamment d'optmiser l'exécution du noeud si les bonnes spécialisations sont utilisées.
Regardons de plus près la première des trois méthodes `add()`. Elle prend en paramètre deux longs. La VM Graal décidera donc d'utiliser cette méthode si deux longs sont croisés autour d'un opérateur '+' pour exécuter l'addition. On voit également l'annotation `@Specialization()` qui contient un attribut `rewriteOn` très intéressant : si lors de l'exécution de la méthode une `ArithmeticException` est levée (causée par un overflow), la VM Graal va alors décider de remplacer le noeud `SLAddNode` courrant avec un nouveau noeud `SLAddNode` implmentant alors la méthode `add()` utilisant des `SLBigNumbers` plutôt que des longs. les `SLBigNumbers` sont implémentés en `BigNumbers` Java, leur attribuant une exécution très lente comparé à des opérations appliquées sur des types primitifs Java. Comme mentionné dans les commentaires de GraalVM, l'évaluation de `Math.addExact()` constitue en l'exécution d'uniquement deux opérations machines.

La troisième méthode `add()` est une spécialisation utilisant un autre attribut de l'annotation `@Specialization()`. Grâce à l'attribut `guards`, on peut alors spécifier à la VM Graal de n'utiliser cette spécialisation que sous certaines conditions. Ici, on s'assure que les deux objets reçus en paramètre sont des strings (boolean retourné par la méthode protected `isString()`).
On voit aussi dans ce code que des annotations `@TruffleBoundaries` sont utilisées. Elles semblent alors êtres utilisées pour les méthodes ayant un long compute time.

4.1.3.3 Conclusion

En conclusion, voici en bref les informations récoltées lors de cette analyse :

- Les fonctions propres à un langage sont exportables et exécutables a l'extérieur du langage
- `@NodeInfo` permet d'ajouter des informations aux noeuds du langage utiles pour le debugging
- `@Specialization` permet de spécifier plusieurs manières d'effectuer une même action, en donnant des hints au compilateur pour choisir la méthode la plus optimisée à exécuter.
- `@TruffleBoundary` semble être utilisé pour les méthodes possédant un long computation time.

Ces informations ne sont pas utilisées dans ProloGraal par soucis de temps d'implémentation. Elles sont néanmoins décrites pour garder une trace de ces potentielles futures améliorations de ProloGraal.

4.2 Restructuration de ProloGraal

Une restructuration du fonctionnement de ProloGraal a été nécessaire pour rendre le langage interopérable. Bien que la restructuration ait demandé la modification de nombreuses lignes, seuls les ajouts majeurs ont ici été documentés.

4.2.1 Exécution des requêtes

L'objectif du projet est alors de pouvoir exécuter du code Prolog au sein d'un langage étranger. Voici comment on exécute ce genre de requête en Java, utilisant la classe GraalVM `Context` [8] :

```
private static final String PROLOGRAAL = "pl";
context.eval(PROLOGRAAL, "requestprolograal(test).");
```

De par l'utilisation du `String PROLOGRAAL`, GraalVM sait explicitement que le langage à évaluer est ProloGraal. Pour exécuter la requête, il va donc demander à la classe `ProloGraalLanguage`, classe clée du code source de ProloGraal, de parser le contenu de la requête (et ainsi, transformer la chaîne de charactère `"requestprolograal(test)."` en objets ProloGraal utilisables pour la résolution de la requête) :

8. GraalVM Team. *Context JavaDoc*. URL : `https://www.graalvm.org/sdk/javadoc/org/graalvm/polyglot/Context.html`.

```
public class ProloGraalLanguage extends
   TruffleLanguage<ProloGraalContext> {
   /* lot of code */
   @Override
   protected CallTarget parse(ParsingRequest request) {
      Source source = request.getSource();
      List<ProloGraalClause> requestedClauses = new
         ArrayList<>();
      //simplified code compared to real impl
      requestedClauses =
         ProloGraalParserImpl.parseProloGraal(source);

      ProloGraalRuntime runtime = new
         ProloGraalRuntime(this.getContextReference().get());

      RootNode eval = new ProloGraalEvalRootNode(this,
         runtime,requestedClauses);

      return Truffle.getRuntime().createCallTarget(eval);
   }
   /* lot of code */
}
```

Les éléments importants sont qu'un parser est utilisé pour transformer l'entrée de `String` à `List<ProloGraalClause>`, un runtime est créé (il contiendra les futures règles admises par le langage) et un `RootNode` (instantié en tant que `ProloGraalEvalRootNode`) est créé pour exécuter la requête (devenue liste de règles ProloGraal). Finalement, ce `RootNode` sera retourné. Le `RootNode` sera par la suite évalué. Cela passe par l'exécution de la méthode `execute()` du noeud, qui est alors instantié en tant que `ProloGraalEvalRootNode`. Regardons alors le contenu de la méthode `execute()` de cette classe :

```
public final class ProloGraalEvalRootNode extends RootNode {
   @Override
   public Object execute(VirtualFrame frame) {
      ArrayList<ProloGraalBoolean> answers = new ArrayList<>();

      for(ProloGraalClause request : requests){
         Deque<Integer> currentBranches = new ArrayDeque<>();
         ProloGraalClause goal = /* goal initialisation */;
         ProloGraalRuntime runtime;
         runtime = new
            ProloGraalRuntime(language.getContextReference().get());
```

```
            runtime.addProloGraalClause(goal);
            ProloGraalBoolean callResult =
               (ProloGraalBoolean) Truffle.getRuntime()
                  .createCallTarget(reference.get().getResolverNode())
                  .call(runtime, currentBranches);
            answers.add(callResult);
         }
         if(answers.size()==1)return answers.get(0);

         ProloGraalBooleanList answersList = new
            ProloGraalBooleanList(answers);
         return answersList;
         //return Truffle.getRuntime().createCallTarget(
            context.getInterpreterNode() ).call();
      }
}
```

L'ancien mécanisme était alors de charger le fichier parsé en tant que nouvelles règles puis d'appeler le dernier `return` commenté, ce qui lançait alors un interpréteur interactif que l'on peut par la suite questionner. Cette précédente méthode présentait alors deux soucis :

- Le retour ce cette méthode (et donc le retour du `Context.eval()` offert par ProloGraal) était le résultat de l'évaluation du noeud interpréteur, ce qui signifie qu'il faudrait drastiquement changer l'interpréteur pour qu'il réagisse de manière voulue (le noeud interpréteur retourne un succès ProloGraal lorsqu'on décide de quitter cet interpréteur, c'est alors ce qui est retourné par ProloGraal lorsqu'on décide d'évaluer des requêtes à l'interpréteur : un succès sans variables assignées).
- Cela impliquerait que l'utilisateur devrait interréagir avec l'interpréteur à chaque appel vers ProloGraal, ce qui n'est pas désiré pour évaluer du code dynamiquement.

Une meilleure alternative est alors de directement exécuter les requêtes de l'utilisateur. Dans le code ci-dessus, on voit la variable `ProloGraalBoolean callResult` être initialisée avec l'exécution Truffle d'un `ResolverNode`, et ce pour chaque requête demandée par l'utilisateur. Ces `callResults` sont alors stockés et retournés en cas de résultats.
Que l'on passe par l'interpréteur interactif ou non, c'est finalement ce `ResolverNode` que l'on exécute pour résoudre notre question Prolog. Ce `ResolverNode` est précédemment initialisé en tant que `ProloGraalResolverNode`, implémentation de ce noeud propre à ProloGraal. Ce noeud retournera alors la réponse à notre question Prolog, un succès ou échec, accompagné de variables unifiées en cas de succès. Bien que ce noeud ainsi que d'autre mécanismes sous-jacents on également été modifiés, il n'est pas pertinant de détailler ces modifications.
Grâce à ces modifications, il est maintenant possible pour ProloGraal de retourner les succès et échecs obtenus après exécution des requêtes fournises par un appel de la méthode `Context.eval()`. À noter que si une seule requête est posée, un objet `ProloGraalBoolean`

sera retourné. Sinon un objet spécial `ProloGraalBooleanList` sera retourné, consistant essentiellement en une liste de `ProloGraalBoolean`. Cela aura un impacte sur la manière dont on va gérer le retour d'un appel à `Context.eval()`, discuté dans la séction `Résultats` de ce chapitre.

4.2.2 Modification du parser

Deux modifications importantes ont été ajoutées au niveau du parser :

- Ajout du charactère '~' comme nouveau délimiteur de chaîne de caractère, seule alternative trouvée pour permettre l'ajout de chaînes de caractère imbriquées. L'utilité de cet ajout est détaillé dans la section `consult/1 et consultstring/1`.
- Accepte maintenant une liste de buts Prolog comme une clause. Nécessaire pour permettre aux utilisateurs de ProloGraal de demander plusieurs buts à résoudre en une seule requête.

Bien que nécessaire, la deuxième modification n'est actuellement pas implémentée de manière convenable. En effet, en Prolog officiel, une liste de but constituant une requête Prolog n'est pas associée à une règle qui ne possède pas de tête de règle. Cela poserait également des problèmes de clarté de code si la suite de ProloGraal était implémentée sans modifier cette spécificité temporaire du projet.
Pour mettre en place ces changements, il suffit de modifier le fichier `ProloGraal.g4`, fichier de grammaire ANTLR[9] interne au projet Java ProloGraal, avec les apports suivants :

```
ATOM : (LOWERCASE (LOWERCASE | UPPERCASE | DIGITS |
   UNDERSCORE)*) |
('~' .*? '~') | //added line
('\'' .*? '\'');

clause :
fact |
goalList | //TODO clause can temporary be a goalList
composedClause
;
```

Il suffira ensuite de run les exécutable `envsetup` et ensuite `generate_parser` disponibles sous `./code/`. Il est conseillé de les exécuter via linux (wsl dans la cmd windows pour lancer un sous-système linux si installé). Si les scripts contiennent des erreurs de syntaxe comportant des /r, il est conseillé d'utiliser la commande linux `dos2unix` pour reformater ces scritps pour linux. Cela générera alors à nouveau les fichiers .Java utiles pour le parsing de ProloGraal, tel les classes `ProloGraalLexer` et `ProloGraalParser`.

9. Terence Parr. *ANTLR Website*. URL : `https://www.antlr.org/index.html`.

4.2.3 Ajouts de règles

Il n'était précédemment pas possible d'ajouter des règles au `ProloGraalRuntime` durant le cours d'exécution de requêtes Prolog. Cela est nécessaire pour l'implémentation des prédicats consult/1 et consultstring/1, mais également pour de futurs ajouts tel le prédicat Prolog assert/1 retract/1. Regardons les méthodes intéressantes pour le management des règles dans `ProloGraalRuntime.java` :

```
public final class ProloGraalRuntime {
    // the clauses. A map is used to allow fast filtering using
       the head of a clause.
    // each map entry contains the list of clauses sharing the
       same head.
    private final Map<ProloGraalTerm<?>, List<ProloGraalClause>>
       clauses;

    public final Map<ProloGraalTerm<?>, List<ProloGraalClause>>
       getClauses() {
       return clauses;
    }

    //edited the two methods to correct Runtime
    public final void addProloGraalClause(ProloGraalClause
       clause){
       clauses.putIfAbsent(clause.getHead(), new ArrayList<>());
       List<ProloGraalClause> clauses1 =
          clauses.get(clause.getHead());
       clauses1.add(clause);
    }

    public final void addProloGraalClauses(List<ProloGraalClause>
       clauseList){
       // put every clauses into the map
       for (ProloGraalClause clause : clauseList) {
          addProloGraalClause(clause);
       }
    }
}
```

Ces ajouts très simple permettent alors d'ajouter des règles au Runtime à la volée. Les méthodes permettant la suppressions de règles, pour le prédicat retract/1 par exemple, devront également être implémentés dans cette classe.

4.3 Export des ProloGraalObjects

Pour que les objets ProloGraal soient utilisables en dehors du langage ProloGraal, ils doivent répondre à des méthodes aux noms prédéfinis. Ces méthodes représentent alors des messages Truffle, méthodes appelables sur n'importe quel objet Truffle retourné par l'évaluation d'un code provenant d'un langage étranger. En Java typiquement, un appel polyglotte s'effectue via l'appel de la méthode `Context.eval()`, `Context` [10] étant une classe GraalVM représentant un contexte polyglotte. Chaque objet doit alors exporter certains messages spécifiques, et implémenter le comportement spécifique à chacun de ces messages pour cet objet. Dans les chapitres qui suivent, les mécanismes d'opérabilité spécifiques à chaque ProloGraalObject exporté sont étayés.

4.3.1 Utilisation de l'InteropLibrary

L'InteropLibrary [11] est une classe Java présente dans la librairie Truffle. Grâce aux annotations Truffle sélectionnées dans le chapitre précédent `Export d'objets SimpleLanguage`, l'InteropLibrary permet de rendre les objets spécifiques à un langage interopérables. Nous allons donc utiliser les annotations suivantes :

- `@ExportLibrary(InteropLibrary.class)` : permet d'exporter notre objet pour l'InteropLibrary. Cette annotation sera simplement inscrite dans le code au dessus de la déclaration de chaque classe exportée. Cela concerne donc les classes présentées dans les chapitres suivants.
- `@ExportMessage` : cette annotation de l'InteropLibrary permet d'exporter une classe en tant que message Truffle. Grâce à cela, notre objet exporté devient utilisable à l'extérieur de son langage source grâce à des méthodes appelables sur celui-ci.

L'annotation `@ExportMessage` exige pour certains groupes de méthode qu'elles soient toutes exportées ou non. Par exemple, la méthode `isBoolean()` est une méthode exportable offerte par l'InteropLibrary, mais il sera également nécessaire d'exporter la méthode `asBoolean()` pour que la compilation du langage soit possible.

4.3.2 ProloGraalBoolean

La classe `ProloGraalBoolean` représente un succès ou un échec Prolog. Cette abstraite est la classe mères de `ProloGraalSuccess` et `ProloGraalFailure`. La classe `ProloGraalBoolean` en elle même ne possède pas de comportement interopérable, mais ses deux classes filles oui.
Penchons nous sur le plus complexe des deux, le `ProloGraalSuccess`. Dans ProloGraal, lorsque nous obtenons un succès, les valeurs attribuées aux variables durant la résolution de la requête sont enregistrées dans l'objet `ProloGraalSuccess` retourné.

10. GraalVM Team. *Context JavaDoc.* URL : `https://www.graalvm.org/sdk/javadoc/org/graalvm/polyglot/Context.html`.
11. GraalVM Team. *InteropLibrary JavaDoc.* URL : `https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/interop/InteropLibrary.html`.

```
@ExportLibrary(InteropLibrary.class)
public class ProloGraalSuccess extends ProloGraalBoolean {
   private Map<ProloGraalVariable, ProloGraalVariable> variables;
   @Override
   public String toString() { return "yes"; }
   @ExportMessage
   public boolean isBoolean(){ return true; }
   @Override
   @ExportMessage
   public boolean asBoolean() {return true;}

   private ProloGraalVariable getVarByName(String name){
   /* get var from map by name or null */ }
   @ExportMessage
   public boolean isMemberReadable(String member)
   { return getVarByName(member)!=null?true:false; }
   @ExportMessage
   public Object readMember(String member) throws /*excps*/ {
      if(isMemberReadable(member)){
         return variables.get(getVarByName(member));
      } else { throw UnknownIdentifierException.create(member); }
   }
   @ExportAllMessages
   public boolean hasMembers() {return variables.size()>0;}
   public Object getMembers() { /* get list of var names */ }
   public void removeMember(String member) throws /* excps */ {
   /* remove member */ }
   public void writeMember(String member, Object value) {
   /* write member */ }
   public boolean isMemberModifiable(String member) {
   /* is member modifiable? */ }
   /* ... */
}
```

On constate alors que la classe `ProloGraalSuccess` exporte deux ensembles de messages :

- `isBoolean()` et `asBoolean()` : permettant de déterminer si notre requête Prolog a mené un échec ou un succès. En cas de succès, il sera possible de lire les variables stockées dans le succès.
- `isMemberReadable()`, `readMember()`, ... : permet d'accéder au contenu des variables, les modifier, même ajouter de nouvelles variables à la liste des succès. Ces méthodes étant assez nombreuses, une certaine partie d'entre-elles ont été tronqués.

Il suffit alors d'implémenter correctement la méthode readMember pour pouvoir retourner les variables par leur nom, étant le `String` member passé en paramètre. On s'assure donc que la variable est présente dans la `Map variables`. En cas de succès, on retourne la variable demandée, sinon, on throw une exception [12] (provenant de la librairie Truffle).
L'utilisateur pourra alors recevoir la variable de son choix, variable ayant également exporté des messages Truffle.
Le code de la classe `ProloGraalFailure` est similaire pour l'export des messages booléens, mais n'exporte pas les messages permettant à l'objet d'être interprétée comme un dictionnaire.

4.3.3 ProloGraalBooleanList

Cet objet ne représentant rien de concret par rapport au réel langage Prolog est utilisé lorsque plusieurs requêtes sont envoyées en un l'évaluation d'un seul `Context.eval()`. En effet, plusieurs `ProloGraalBooleans` vont être générés et il est nécessaire de rendre la totalité des réponses accessibles. Voici uniquement le code essentiel de cette classe, car en effet, cette classe exporte également un grand nombre de méthodes du aux méthodes à exporter pour que l'objet interréagisse comme un tableau générique Truffle :

```
@ExportLibrary(InteropLibrary.class)
public class ProloGraalBooleanList extends ProloGraalObject {
   private final List<ProloGraalBoolean> booleanList;
   public ProloGraalBooleanList(List<ProloGraalBoolean>
      booleanList){
      this.booleanList = new ArrayList<>(booleanList);
   }
   @ExportMessage
   public boolean hasArrayElements(){ return
      !booleanList.isEmpty(); }
      @ExportMessage
   public long getArraySize(){ return booleanList.size(); }
   @ExportMessage
   public Object readArrayElement(long index) throws /* excps */{
      if( hasArrayElements() && isArrayElementReadable(index) ){
         return booleanList.get((int)index);
      }else{ /* throw excps */ }
}
```

L'objet est alors fonctionnel, et retourne des excpetions si on le requête avec des index invalides.

12. GraalVM Team. *UnknownIdentifierException JavaDoc.* URL : https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/interop/UnknownIdentifierException.html.

4.3.4 ProloGraalVariable

Les `ProloGraalVariables` ont la propriété de se bind à un terme ProloGraal générique. Il peut alors prendre la forme d'un nombre, d'un texte ou même encore de prédicats. pour l'instant, seul les nombres et les atomes (représentant des chaines de caractère) sont exportés. Voyons premièrement l'export de la variable pour les atomes :

```
@ExportLibrary(InteropLibrary.class)
public final class ProloGraalVariable extends
   ProloGraalTerm<ProloGraalVariable> {
    private final String name;
    private boolean isBound = false;
    private ProloGraalTerm<?> boundValue;

    @ExportMessage
    public boolean isString(){ return isBound && boundValue
       instanceof ProloGraalAtom && ((ProloGraalAtom)
       boundValue).isString(); }

    @ExportMessage
    public String asString() throws UnsupportedMessageException{
       if(isString()){
          return ((ProloGraalAtom) boundValue).asString();
       }else{ throw UnsupportedMessageException.create(); }
    }
}
```

Pour savoir si la variable peut actuellement retourner un `String` Java, on s'assure que la variable est déjà bindée, que le bound term est bien un atome, et que l'atome lui même réponde true à `isString()` (l'atome exporte également `isString()` et `asString()` mais ne nécessite pas un chapitre dans le rapport). Si tout cela est vérifié, on extrait alors le nom de l'atome et retournons le résultat.
Le code pour l'export en nombre est similaire, à l'exception que les nombres doivent exporter bien plus de méthodes que les chaînes de caractère.

4.3.5 ProloGraalNumber

Les nombres ProloGraal ont également été exportés pour pouvoir obtenir leurs valeurs en dehors du langage ProloGraal. Deux classes héritent de la classe abstraite `ProloGraalNumber`, étant `ProloGraalIntegerNumber` et `ProloGraalDoubleNumber`. Voici une version épurée de la classe `ProloGraalDoubleNumber` :

```
@ExportLibrary(InteropLibrary.class)
```

```
public final class ProloGraalDoubleNumber extends
   ProloGraalNumber<ProloGraalDoubleNumber> {
    private final double value;
    @ExportMessage
    public boolean isNumber() { return true; }
    @ExportMessage
    public double asDouble(){ return value; }
    @ExportMessage
    public int asInt() throws /* excps */ {
       if (fitsInInt()) {
          return (int) value;
       } else { throw UnsupportedMessageException.create(); }
    }
    @ExportMessage
    public boolean fitsInDouble() { return true; }
    @ExportMessage
    public boolean fitsInInt() { return value<=Integer.MAX_VALUE
       && value>=Integer.MIN_VALUE; }
}
```

On voit que les `ProloGraalNumbers` doivent alors s'assurer que la valeur qu'ils possèdent puissent être downcast sans perte de données. Lors d'un cast d'un `double` vers un `int` en Java, le nombres après la virgule se retrouvent tronqués. On s'assure alors simplement que la valeur du nombre est inférieure à la valeur maximale qu'un `Integer` Java peut posséder.
Le nombre exporte alors correctement ses valeurs. Le code présent dans `ProloGraalIntegerNumber` est quasi semblable, à l'exception qu'il ne dois pas tester si sa valeur peut être plus grande qu'un `Integer` (le type primitif de sa variable `value` est `int`).

4.4 Prédicats complémentaires

Quelques prédicats supplémentaires ont été intégrés pour permettre au langage ProloGraal de pouvoir consulter des règles en cours d'exécution, ou bien encore de permettre à nouveau l'utilisation de l'interpréteur interactif.

4.4.1 consult/1 et consultstring/1

Ces deux prédicats ont été implémentés pour consulter des règles Prolog à la volée. Le prédicat consult/1 permet de consulter un fichier contenant des règles Prolog, alors que consultstring/1 consulte les règles contenues dans l'atome passé en paramètre. Plongeons dans la méthode `execute()` du consult/1, le constructeur étant assez trivial (semblable à celui de is/2 mais avec un seul argument) :

```
public ProloGraalBoolean execute() {
```

```
    ProloGraalTerm consultContentTerm =  arg.getRootValue();
    if(consultContentTerm instanceof ProloGraalAtom){
       String filename = ((ProloGraalAtom)
          consultContentTerm).getName();
       //consult atoms should be encapsuled in string delimiter
          to support path with spaces
       filename = filename.substring(1,filename.length()-1);
       File loadedFile = new File(filename);
       if(loadedFile.exists()) {
          String fileContent;
          /* read fileContent or return Failure */
          Source source = Source.newBuilder(
             ProloGraalLanguage.ID, fileContent, null ).build();
          //we filter clauses to remove goal lists from returned
             clauses
          List<ProloGraalClause> clauses =
             ProloGraalParserImpl.parseProloGraal( source
             ).stream().filter( proloGraalClause ->
             proloGraalClause.getHead() != null ).collect(
             Collectors.toList() );

          context.getRuntime().addProloGraalClauses(clauses);
          return new ProloGraalSuccess(getVariables());
       }else{ /* print file does not exist and throw failure */ }
    }else{ /* consult/1 predicate takes an atom as argument and
       failure */ }
}
```

Les clauses sont chargées via le parser, ce qui nous retourne une liste de clauses. On la transforme alors en stream pour pouvoir filtrer les clauses qui n'ont pas de tête. Ces fameuses clauses sans tête sont en fait les requêtes étant composés de plusieurs buts. Ce filtre permet de ne pas ajouter au Runtime des règles qui provoqueront par la suite des erreurs. Ajouter une requête composée de plusieurs but dans un fichier Prolog composé de règles n'a pas de sens et ne devrait pas arriver, mais ce filtre ajoute une robustesse supplémentaire à la lecture des inputs. Ce filtre ne serait plus nécessaire quand les requêtes composées de plusieurs buts seront correctement implémentées.
Lorsque les clauses ont été filtrées, elles sont simplement ajoutées au Runtime et retourne ensuite un succès. Le prédicat consultstring/1 possède le même comportement mais ne charge pas de fichier et parse directement le contenu de l'atome (il doit également supprimer le premier et dernier caractère, comme expliqué en commentaire dans le code ci-dessus).
L'utilisation du caractère ' ' en tant que second délimiteur de chaîne de caractère est justifié

par le fonctionnement du prédicat consultstring/1. Une variable ne peut pas s'unifier avec une liste de clauses, il est alors nécessaire d'intégrer la liste de clauses dans un atome. Les clauses peuvent également posséder des atomes, et les délimiteurs de ceux-ci entreraient en conflit avec ceux de l'atome contenant les clauses.

4.4.2 useinterpreter/0

Ce prédicat très simple permet de lancer l'interpréteur interactif. Voyons directement la méthode `execute()` :

```
@Override
public ProloGraalBoolean execute() {
    //Launch interpreter
    Truffle.getRuntime().createCallTarget(
       context.getInterpreterNode() ).call();
    return new ProloGraalSuccess(getVariables());
}
```

Il n'est pas nécessaire d'ajouter un Runtime car le noeud interpréteur possède une référence vers lui. Si l'on sort alors de l'interpréteur intéractif avec la commande 'exit.', le programme va simplement continuer après avoir retourné le succès.

4.5 Résultats

Voici un exemple d'appel à du code ProloGraal depuis de Java, puis utilisation de ces éléments pour effectuer diverses actions :

```
System.out.println("###First test###");
Value prologBoolean = context.eval( languageId: "pl", source: "consultstring('test(10).'), test(A), is(B,'**'(2,A)).");
System.out.println("Did request success? "+prologBoolean.asBoolean());
if(prologBoolean.asBoolean()){
   System.out.println("A value: "+prologBoolean.getMember("A").asInt());
   System.out.println("B value: "+prologBoolean.getMember("B").asInt());
}

System.out.println("###Second test###");
Value prologBooleanArray = context.eval( languageId: "pl", source: "consultstring(~test(a). test2(b).~). test(A). test2(c),test2(B).");
System.out.println("did 2nd request success? " + prologBooleanArray.getArrayElement(1).asBoolean());
System.out.println("A value: "+prologBooleanArray.getArrayElement(1).getMember("A").asString());
System.out.println("did 3nd request success? " + prologBooleanArray.getArrayElement(2).asBoolean());


###First test###
Did request success? true
A value: 10
B value: 1024
###Second test###
did 2nd request success? true
A value: a
did 3nd request success? false
```

On voit alors que l'on peut récupérer les variables des booléens Prolog en cas de succès. À

noter que la classe de l'objet retourné par `Context.eval()` est `Value`[13], classe présente dans la librairie Truffle. ProloGraal retourne effectivement un tableau de booléens Prolog dans le cas d'un appel composé de plusieurs requêtes.
Avec les modifications apportées, du code ProloGraal est également exécutable sur les autres langage implémentant correctement l'interopérabilité, comme par exemple JavaScript. Voici un exemple de code ProloGraal appelé depuis du JS, puis quelques opérations sont effectuées également en JS avec le résultat :

```
context.eval( languageId: "js",  source: "var success = Polyglot.eval('pl'," +
                                         "\"consultstring(~power(5).~), " +
                                         "power(A), " +
                                         "is(B,'**'(2,A)), " +
                                         "is(C,'/'(B,A)).\");\n" +
                                       "console.log('prolog called from js result: '+success);\n" +
                                       "console.log('prolog B var: '+success.B);\n" +
                                       "console.log('prolog C var: '+success.C);");
prolog called from js result: true
prolog B var: 32
prolog C var: 6
```

13. GraalVM Team. *Value JavaDoc*. URL : `https://www.graalvm.org/truffle/javadoc/org/graalvm/polyglot/Value.html`.

5 Améliorations possibles

Alors que ProloGraal possède maintenant des caractéristiques interopérables, il y a beaucoup d'améliorations envisageables :

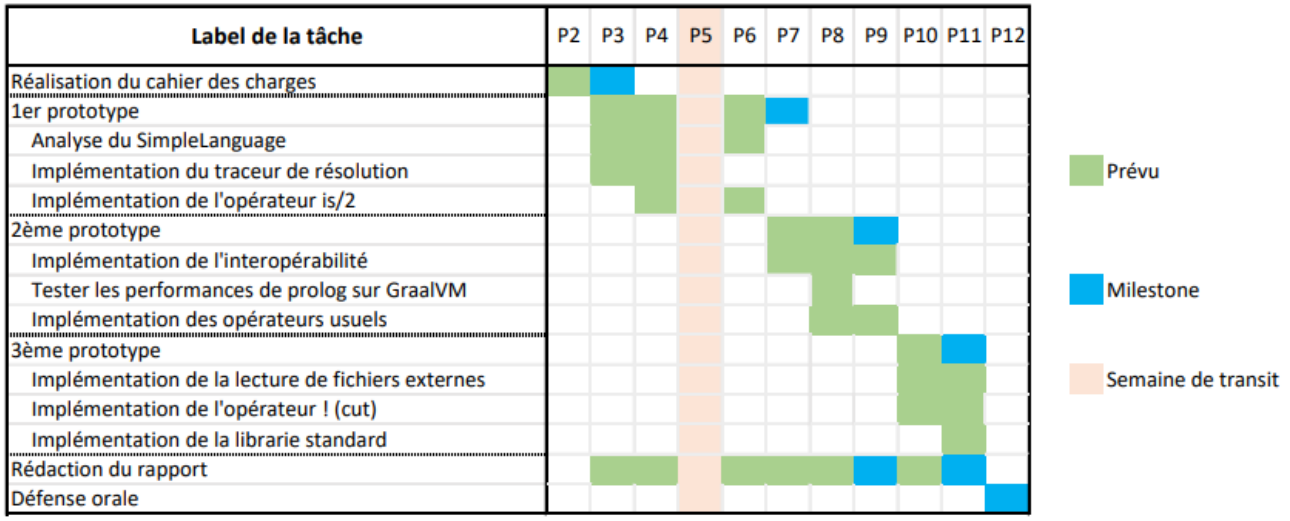
- Corriger le système actuel pour traiter les requêtes à plusieurs buts (actuellement parsés en tant que règles ProloGraal)
- Utiliser les mécanismes d'optimisations offerts par Truffle analyés, comme par exemple les annotations `@TruffleBoundary` ou `@Specialization`
- Ajouter des prédicats built-in Prolog très courrants mais pourtant absents de ProloGraal, tel que le cut (!) ou assert et retract par exemple
- Exporter les prédicats built-in pour les rendre exécutables en dehors de ProloGraal
- Modifier le langage pour que les syntaxes simplifiées de Prolog soient utilisables
- Modifier le fonctionnement actuel de l'arbre de résolution pour ne pas utiliser la pile d'appels Java à chaque nouveau noeud. Actuellement, la profondeur de l'arbre de preuve ne peut pas excéder un certain nombre sous peine de provoquer une `StackOverflowException`

6 Conclusion

Ce chapitre consiste en la conclusion du rapport. Une critique sur l'atteinte des objectifs est décrite, suivi d'une conclusion personnelle.

6.1 Atteinte des objectifs

Alors que les objectifs présents dans le cahier des charges pour le premier prototype ont été tous réalisés, seul l'implémentation de l'interopérabilité et l'implémentation de la lecture de fichiers externes ont été réalisé pour le reste des objectifs.
Cela est principalement du au temps nécessaire à la compréhension du fonctionnement du projet, mais également du temps nécessaire pour reconstruire certaines infrastructures déjà présentes pour les rendre plus permissives ou modifier leurs comportements. Il eut souvent arrivé lors de l'implémentation qu'un obstacle se présentait par rapport à l'implémentation actuelle, et qu'un contournement provisoire est temporairement implémenté pour en discuter avec Monsieur Bapst.
En addition de cela, une semaine a été perdue par l'étudiant durant la semaine de transit liée aux conditions de confinements. Le travail était possible durant cette semaine mais une mécompréhension de l'élève sur la situation a men à une semaine sans travail. À part ces quelques points relevés, l'objectif principal du projet, étant de rendre ProloGraal interopérable, est accompli. Comme il a été constaté dans la section `Résultats` du chapitre `Interopérabilité`. Quelques tests montrent alors ProloGraal répondre à des requêtes provenant de Java, ou encore JavaScript.



6.2 Conclusion personnelle

En conclusion, j'ai trouvé ce projet très intéressant. Alors que j'avais quelques difficultés en début de projet pour bien intégrer le fonctionnement général de ProloGraal, ainsi que l'utilité exacte de chaque composants, j'ai tout de même eu plaisir à intégrer chaque composant. La

réflexion demandée par la restructuration de ProloGraal fut également très intéressante et enrichissante. Avoir refait du Prolog est également très sympatique, j'ai grâce au projet réappris certains concepts oubliés de Prolog et même mieux compris certains. J'ai également trouvé l'idée de pouvoir rendre ce langage interopérable très intéressante, surtout pour la simplicité de syntaxe qu'il offre pour résoudre des problèmes complexes. Je suis tout de même un peu déçu de ne pas avoir travaillé durant la semaine de transit maintenant que le projet touche à sa fin. Je pense que le projet aurait pu être un peu plus étoffé et mieux terminé avec cela.

7 Déclaration d'honneur

Je, soussigné, Tony Licata, déclare sur l'honneur que le travail rendu est le fruit d'un travail personnel.
Je certifie ne pas avoir eu recours au plagiat ou à toute autre forme de fraude. Toutes les sources d'information utilisées et les citations d'auteur ont été clairement mentionnées.
Belfaux, le 8 mai 2020

8 Remerciements

Je remercie le Professeur Frédéric Bapst pour les aides apportées tout au long du projet, dans les différentes problématiques rencontrées, ainsi que pour l'aide administrative fournie.

9 Bibliographie

Références

[1] JetBrains. *Download IntelliJ*. URL : https://www.jetbrains.com/idea/download/other.html.

[2] Martin Spoto & Tony Licata. *ProloGraal Repository*. URL : https://gitlab.forge.hefr.ch/tony.licata/prolog-truffle/.

[3] Terence Parr. *ANTLR Website*. URL : https://www.antlr.org/index.html.

[4] GraalVM Team. *Context JavaDoc*. URL : https://www.graalvm.org/sdk/javadoc/org/graalvm/polyglot/Context.html.

[5] GraalVM Team. *Getting started with GraalVM*. URL : https://www.graalvm.org/getting-started/.

[6] GraalVM Team. *Github release of JVM 19.2.0.1*. URL : https://github.com/oracle/graal/releases/tag/vm-19.2.0.1/.

[7] GraalVM Team. *InteropLibrary JavaDoc*. URL : https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/interop/InteropLibrary.html.

[8] GraalVM Team. *Simple Language Repository*. URL : https://github.com/graalvm/simplelanguage.

[9] GraalVM Team. *TruffleBoundary JavaDoc*. URL : https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/CompilerDirectives.TruffleBoundary.html.

[10] GraalVM Team. *UnknownIdentifierException JavaDoc*. URL : https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/interop/UnknownIdentifierException.html.

[11] GraalVM Team. *Value JavaDoc*. URL : https://www.graalvm.org/truffle/javadoc/org/graalvm/polyglot/Value.html.