



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

---

# Programmation logique sur Truffle et GraalVM

## Rapport de projet

---

Projet de Bachelor  
I-3  
Printemps 2020

Tony Licata  
tony.licata@edu.hefr.ch

Superviseurs:

Frédéric Bapst  
frederic.bapst@hefr.ch

Julien Bégard  
julien.begard@protonmail.com

17 juillet 2020

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Contexte . . . . .	4
1.2	Objectifs . . . . .	5
1.3	Déroulement . . . . .	6
1.4	Structure du rapport . . . . .	6
1.5	Gestion du projet . . . . .	7
<b>2</b>	<b>Technologies du projet</b>	<b>8</b>
2.1	GraalVM . . . . .	8
2.2	Truffle . . . . .	8
2.3	Prolog . . . . .	8
<b>3</b>	<b>Fonctionnalités</b>	<b>9</b>
3.1	Corrections . . . . .	9
3.2	Ajouts . . . . .	12
3.3	Conclusion . . . . .	14
<b>4</b>	<b>Optimisations</b>	<b>16</b>
4.1	Profiling . . . . .	16
4.2	Migration des prédicats builtins . . . . .	18
4.3	Spécialisation des noeuds Truffle . . . . .	20
4.4	Autres mécanismes offerts par Truffle . . . . .	24
4.5	Conclusion . . . . .	26
<b>5</b>	<b>Tests de performance</b>	<b>27</b>
<b>6</b>	<b>Tests unitaires</b>	<b>30</b>
6.1	Problème amené par l'upgrade du JDK GraalVM . . . . .	30
6.2	Nouveaux tests unitaires . . . . .	30
<b>7</b>	<b>Améliorations possibles</b>	<b>33</b>
<b>8</b>	<b>Conclusion</b>	<b>34</b>
8.1	Atteinte des objectifs . . . . .	34
8.2	Conclusion personnelle . . . . .	34
<b>9</b>	<b>Déclaration d'honneur</b>	<b>35</b>
<b>10</b>	<b>Remerciements</b>	<b>35</b>



# 1 Introduction

Cette section introduit le document en situant le contexte, en présentant les différents objectifs établis et en décrivant le déroulement prévu pour la réalisation du projet. Les détails sur la gestion du projet sont également introduits dans cette section.

## 1.1 Contexte

GraalVM est une machine virtuelle basée sur la Java Virtual Machine, qui possède la spécificité de pouvoir exécuter des programmes polyglottes. Pour autant qu'un langage soit implémenté pour GraalVM, il est possible, au cours de la même exécution, d'exécuter du code dans l'un ou l'autre de ces langages. Pour pouvoir faciliter l'implémentation d'un tel langage, l'équipe GraalVM a mis au point une API nommée Truffle, ainsi qu'un langage fictif "tutoriel" nommé SimpleLanguage qui est implémenté via Truffle. L'équipe GraalVM annonce également des gains de performances par rapport à une exécution classique du langage désiré, motivé par leur slogan "Run Programs Faster Anywhere".

Lors de précédents projets de semestre réalisés à l'HEIA-FR, un langage représentant le langage de programmation logique Prolog, nommé ProloGraal, a été implémenté pour GraalVM avec Truffle. Ces deux projets se sont déroulés durant l'année scolaire 2019-2020, le premier étant réalisé par M. Spoto Martin, et le second par moi-même. Ce projet de Bachelor est donc une continuation de projet.

ProloGraal reproduit essentiellement les fonctionnalités fondamentales de Prolog, et est sujet à de nombreuses améliorations, autant pour l'ajout de nouvelles "features" que pour des corrections de concepts actuellement en place.

Une des faiblesses notables de ProloGraal est sa vitesse d'exécution lente qui apparaît 1000 fois plus lent que d'autres environnements Prolog classiques tel SWI-Prolog ou GNU Prolog par exemple. Au vu des promesses de l'équipe GraalVM à propos des performances de leur machine virtuelle, on peut espérer décemment accélérer la vitesse d'exécution actuelle de ProloGraal.

Pour optimiser un langage, Truffle offre un panel d'outils intéressants. On peut par exemple citer la spécialisation de noeuds instrumentables, permettant au langage d'emprunter des chemins d'exécution optimisés selon les paramètres de la méthode courante. ProloGraal ne profitant pas de ces optimisations, il semble une bonne opportunité de les intégrer pour permettre de l'accélérer via des mécanismes Truffle, et également d'en découvrir plus sur les possibilités que nous offre cette API.

## 1.2 Objectifs

Un des objectifs principaux de ce projet consiste à optimiser ProloGraal. Pour se faire, il sera nécessaire de le profiler afin d'obtenir des informations claires sur les portions du code ralentissant l'exécution. En plus de cela, des mécanismes d'optimisation mis à disposition par Truffle seront utilisés pour accélérer ProloGraal.

Lors de la résolution d'un but Prolog dans ProloGraal, cas d'utilisation principal d'un interpréteur Prolog, seuls deux comportements imposent à la machine d'appeler de nouvelles routines : une unification avec une règle possédant un ou plusieurs buts, car il sera ensuite nécessaire de résoudre ce ou ces nouveaux buts ; ou une unification avec un prédicat builtin, prédicat représentant les fonctions de la librairie standard Prolog. Le système d'unification des buts aux règles étant assez complexe, il a été décidé d'implémenter les mécanismes de spécialisation de noeuds pour les prédicats builtins.

Des benchmarks seront effectués pour mesurer les performances de ProloGraal après qu'une optimisation majeur a été implémentée pour mesurer les gains potentiels apportés.

Mis à part les optimisations à apporter, il est également important d'enrichir les fonctionnalités actuelles de ProloGraal. Les opérateurs traditionnels de Prolog, tels que `=/2`, `=</2` ou `>/2` par exemple, seront implémentés en tant que prédicats builtins, offrant ainsi de nouvelles possibilités de programmes pour les utilisateurs de ProloGraal. D'autres prédicats builtins seront implémentés selon les besoins d'implémentation, tels le prédicat `real_time/1` qui sera utile pour calculer les temps d'exécution lors de benchmarks.

GraalVM offre la possibilité pour un langage d'exécuter du code dans un langage étranger, pour autant que le langage source implémente correctement certains mécanismes. Actuellement, ProloGraal ne permet pas aux utilisateurs d'évaluer du code dans un langage étranger, et il est également envisagé d'ajouter cette fonctionnalité.

ProloGraal possède également quelques lacunes amenées du précédent projet. Pour ainsi apporter de nouvelles fonctionnalités dans un environnement sain, une restructuration du code sera nécessaire pour mieux représenter le langage Prolog. Par exemple, ProloGraal gère actuellement les requêtes Prolog comme des règles Prolog, mais est implémenté de sorte à gérer ce souci. Chacune des lacunes désignées à être corrigées est présente sur le planning, dans la section "Gestion du projet". En plus de ces corrections, le JDK GraalVM actuellement utilisé par ProloGraal sera mis à jour, et correspondra au dernier JDK stable mis à disposition par l'équipe GraalVM<sup>1</sup>.

En résumé, quatre objectifs généraux ressortent des objectifs suscités, et la liste des activités dans le planning seront groupés selon ces quatre groupes :

- Correction des défauts de l'implémentation actuelle ;
- Ajout de features importantes de Prolog ;
- Amplification de l'interopérabilité de ProloGraal ;
- Optimisation de l'exécution de code ProloGraal.

---

1. GraalVM TEAM. *Github release of JVM 20.1.0*. URL : <https://github.com/graalvm/graalvm-ce-builds/releases/tag/vm-20.1.0>.

### 1.3 Déroulement

Ce projet est défini sur deux phases principales :

- Une première phase permettant de corriger les lacunes restées dans ProloGraal du précédent projet, pour assurer une structure de code saine pour les futures modifications ;
- Une seconde phase consistant à ajouter les différentes nouvelles fonctionnalités évoquées ainsi que l'optimisation de ProloGraal.

### 1.4 Structure du rapport

Bien que la section "Déroulement" nous indique que le projet est séparé en deux phases, le rapport lui ne les distinguera pas spécialement.

Quelques fragments de code présents dans la version finale du projet sont documentés dans le rapport lorsque cela est pertinent. Le code présent dans le rapport est une version épurée du vrai code, seuls les éléments pertinents sont gardés.

Le rapport est structuré comme suit :

- Description des technologies utilisées ;
- Fonctionnalités ajoutées à ProloGraal ou corrigées ;
- Optimisations apportées et potentielles optimisations pour le futur ;
- Tests de performances de ProloGraal ;
- "Correction" des précédents tests unitaires ainsi que les nouveaux tests apportés ;

## 1.5 Gestion du projet

Pour assurer la gestion du projet, une séance hebdomadaire sera organisée chaque mercredi matin avec le Superviseur Frédéric Bapst. Ces séances ont pour but de réviser les décisions de l'étudiant si besoin, et également d'apporter des conseils d'implémentation.

Un dépôt git public est présent sur le gitlab de l'HEIA-FR, contenant le code de ProloGaal mais également les fichiers administratifs. Le dépôt est utilisé tout au long du projet pour rendre accessibles les documents administratifs. Le lien<sup>2</sup> menant vers le dépôt git est présent en note de bas de page.

Un procès verbal est rédigé pour chaque séance hebdomadaire dans le but de garder une trace des différentes décisions établies lors de celles-ci. Ces procès verbaux sont accessibles sur le dépôt git sous `./docs/pv`.

Un cahier des charges et un planning ont été réalisés pour clairement poser les objectifs du projet. Les deux documents sont disponibles sur le dépôt git sous `./docs/specs`, et le planning est également présent dans cette section sous la figure 1.

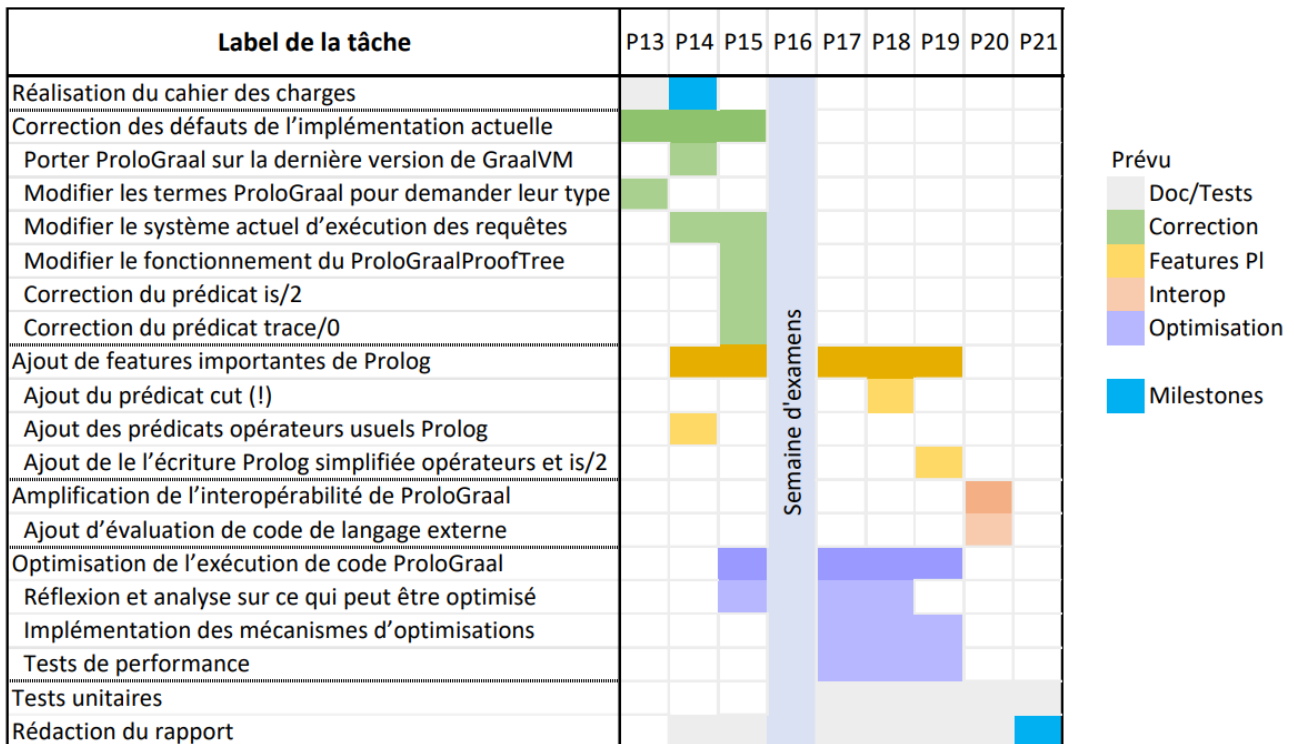


FIGURE 1 – Planning du projet

2. Martin Spoto & Tony LICATA. *ProloGaal Repository*. URL : <https://gitlab.forge.hefr.ch/tony.licata/prolog-truffle/>.

## 2 Technologies du projet

Cette section décrit les différentes technologies utilisées dans le cadre du projet.

### 2.1 GraalVM

GraalVM, comme annoncé dans le contexte, est une machine virtuelle permettant d'exécuter, au sein d'un même fil d'exécution, du code dans plusieurs langages différents. GraalVM offre divers moyens d'utiliser leur machine virtuelle, comme par exemple l'outil GraalVM Native Image<sup>3</sup> permettant de transformer du code Java en standalone exécutable, mais nous allons uniquement utiliser le JDK GraalVM 20.1.0 pour Java 11<sup>4</sup> pour ce projet.

### 2.2 Truffle

Truffle est une API nous permettant d'intégrer un nouveau langage pour GraalVM. Pour ce projet, Truffle sera chargé en tant que librairie Maven via le pom.xml du projet.

Truffle est l'API nous permettant d'implémenter les différents mécanismes d'optimisation offerts par GraalVM. En effet, différentes annotations Java nous permettent d'instrumenter les classes du langage désiré pour les transformer en noeuds instrumentables Truffle. Des précisions sur ces noeuds sont apportées dans le chapitre "Optimisations" du rapport.

### 2.3 Prolog

Prolog<sup>5</sup> est un langage de programmation logique. À la différence d'un langage de programmation impératif, Prolog consiste essentiellement en la résolution de différentes requêtes passées via un interpréteur, résolution basée sur les données précédemment récoltées par l'interpréteur (règles chargées). Pour une meilleure compréhension du rapport, il est conseillé de connaître les éléments de base du langage Prolog, tels que les requêtes, les règles, les succès ou les termes.

---

3. GraalVM TEAM. *Native Image*. URL : <https://www.graalvm.org/docs/reference-manual/native-image/>.

4. GraalVM TEAM. *Github release of JVM 20.1.0*. URL : <https://github.com/graalvm/graalvm-ce-builds/releases/tag/vm-20.1.0>.

5. WIKIPEDIA. *Prolog*. URL : <https://fr.wikipedia.org/wiki/Prolog>.



## 3 Fonctionnalités

Ce chapitre décrit les différentes fonctionnalités qui ont été modifiées ou ajoutées à ProloGaal durant ce projet.

### 3.1 Corrections

Ce chapitre expose les corrections appliquées sur les fonctionnalités provenant des précédents projets de semestre impliquant ProloGaal.

#### 3.1.1 Représentation d'une requête Prolog

Lors de la précédente version de ProloGaal, les requêtes Prolog (Prolog queries, représentant les questions que l'on pose à un interpréteur usuel Prolog) étaient contenues dans des instances de la classe `ProloGaalClause`. Cette erreur de conception liait ainsi des objets Prolog à des objets Java ne les représentant pas réellement, mais le code possédait des contournements pour gérer ces règles Prolog particulières en tant que requêtes Prolog.

Ce projet a mené à une correction de ce problème en ajoutant une nouvelle classe `ProloGaalQuery`. Cette classe représente simplement une suite de buts contenus dans une liste, et offre quelques méthodes pour manipuler cette liste :

```
public class ProloGaalQuery {
    // the goals of this query
    private List<ProloGaalTerm<?>> goals;

    public ProloGaalQuery() {
        goals = new ArrayList<>();
        variables = new HashMap<>();
    }

    public void addGoal(ProloGaalTerm<?> goal) {
        goals.add(goal);
    }

    public List<ProloGaalTerm<?>> getGoals() {
        return goals;
    }
}
```

Les termes Prolog composant la requête sont stockés dans la liste java `goals`. Les buts seront par la suite accessibles via la méthode `getGoals`. La classe gère également les variables Prolog présentes dans les buts, ce mécanisme a été repris de la classe `ProloGaalClause`.

En plus de l'ajout de cette classe, des modifications ont été nécessaires dans d'autres parties du code, notamment le système de parsing de fichiers Prolog qui doit maintenant parser les requêtes

Prolog en tant que `ProloGaalQuery`. Ces modifications étant éparpillées dans plusieurs fichiers, il n'est pas pertinent de discuter de celles-ci.

### 3.1.2 Prédicat builtin `is/2`

Le prédicat builtin `is/2` Prolog permet d'effectuer des calculs, et d'assigner le résultat de ce calcul à une variable. Par exemple, pour calculer  $3 + 4$  en Prolog, on pourra écrire `is(A, '+'(3,4))`, ce qui assignera la valeur 7 à la variable A.

Le prédicat `is/2` possède une Java `Map` d'opérateurs prédéfinis<sup>6</sup>, opérateurs possédant un nombre arbitraire d'arguments. L'opérateur binaire `+/2` par exemple possède deux arguments, mais il existe également des opérateurs unaires tels `exp/1` représentant  $e^x$ , voire même simplement `e/0` pour représenter la constante  $e$ .

La précédente version de ProloGaal ne permettait l'utilisation que d'opérateurs binaires pour le prédicat builtin `is/2`. Cela signifie qu'il était par exemple impossible d'ajouter un opérateur unaire à la `Map` des opérateurs valides.

Il était précédemment impossible d'ajouter un nouvel opérateur unaire car la classe Java `ProloGaalIsBuiltin` définissait les valeurs de sa `Map` d'opérations en tant qu'objets implémentant l'interface Java `BiFunction`<sup>7</sup>. On pouvait ainsi alimenter la `Map` d'expressions lambda par exemple, mais celles-ci devaient forcément posséder deux paramètres.

Pour corriger cela, il a été décidé d'utiliser la Java `Function`<sup>8</sup> interface à la place, qui elle ne prend qu'un paramètre. Ce paramètre représente une liste d'arguments, de taille arbitraire. Après avoir changé l'implémentation des différentes opérations de `is/2` pour correctement traiter ce nouveau paramètre, il est maintenant possible de simplement ajouter un nouvel opérateur pour le prédicat `is/2` avec un nombre arbitraire d'argument.

Pour maintenir une certaine clarté au sein de la classe Java `ProloGaalIsBuiltin`, il a été décidé de migrer la `Map` d'opérations vers une classe `ProloGaalIsOperators`, qui contient uniquement des méthodes et attributs statiques. Voici un extrait de cette classe nous montrant l'ajout de l'opérateur `exp/1` :

```
public final class ProloGaalIsOperators {
    /* is/2 operators (only exp/1 for this example)*/

    public static ProloGaalNumber exp(List<ProloGaalNumber>
        args) {
        if (args.size() != 1) return null;
        return new ProloGaalDoubleNumber(new HashMap<>(),
            Math.exp(args.get(0).asDouble()));
    }
}
```

---

6. Tau PROLOG. *is/2 - Manual*. URL : <http://tau-prolog.org/documentation/prolog/builtin/is/2>.

7. ORACLE. *BiFunction JavaDoc*. URL : <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/BiFunction.html>.

8. ORACLE. *Function JavaDoc*. URL : <https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html>.

```

}

public static
Map<ProloGaalAtom,
    Function<List<ProloGaalNumber>,ProloGaalNumber>>
getOperationsMap(){
    Map<ProloGaalAtom,
        Function<List<ProloGaalNumber>,ProloGaalNumber>>
        operationsMap = new HashMap<>();
    //...
    addOpperation(operationsMap, "exp",
        ProloGaalIsOperators::exp);
    return operationsMap;
}

private static void addOpperation( Map<ProloGaalAtom,
    Function<List<ProloGaalNumber>,ProloGaalNumber>>
    operations,
    String atomName,
    Function<List<ProloGaalNumber>,ProloGaalNumber>
    function){
    operations.put(new ProloGaalAtom(new
        HashMap<>(),atomName), function);
}
}

```

Tous les opérateurs de `is/2` ayant été implémentés dans `ProloGaal` étaient bien fonctionnels après application de ce changement. Par la suite, après que le fonctionnement des prédicats builtins a changé, les opérateurs de `is/2` représentés par des atoms (`e/0`, `pi/0`, `tau/0`) ne furent plus utilisables, et cette lacune est toujours présente. Ce problème est apparu car les opérateurs du prédicat `is/2` sont maintenant contenus dans des `ProloGaalIsOpStructure`. Cette classe héritant de `ProloGaalStructure` impose donc à l'opérateur d'être une structure, et cela cause des problème lors du parsing d'un fichier par `ProloGaal` si celui-ci possède un opérateur atome.

### 3.1.3 Noeud d'arbre de preuve `ProloGaal`

Prolog résout les questions qu'on lui adresse en manipulant un arbre de preuve jusqu'à ce que celui-ci mène à un succès pour chaque but de la question d'origine. Cet arbre est constitué de noeuds, appelés les noeuds d'arbre de preuve.

Ces noeuds représentent les buts restants à résoudre pour déterminer si la question initiale mène vers un succès ou non. Ainsi, tant que la totalité des buts du noeud courant ne sont pas

résolus, un nouveau noeud sera généré dans l'arbre de preuve.

ProloGaal implémente la résolution d'une requête Prolog via la classe `ProloGaalProofTreeNode`. Précédemment, celle-ci créait une nouvelle instance de `ProloGaalProofTreeNode` pour avancer dans la résolution des buts du noeud courant, puis appelait la méthode `execute()` pour résoudre les buts du nouveau noeud. Cela menait alors à un ajout d'appel de méthode dans la pile d'appels de méthode Java, et liait ainsi la profondeur possible de l'arbre de preuve avec la taille de la pile d'appels Java, comme pour tout code récursif.

ProloGaal se passe désormais de la pile d'appel java pour résoudre un but. Pour ce faire, la méthode `ProloGaalProofTreeNode.execute()` va simuler l'ancien système d'appel récursif à l'intérieur d'une boucle while. La condition de sortie de cette boucle est que la liste de buts courants soit vide, et donc que tous les buts aient été résolus. Des attributs de classe ont été ajoutés pour correctement préserver les comportements du précédent code récursif, comme par exemple la gestion des variables locales Java via une pile, gestion des états de la fonction `execute()` via des enums Java, etc...

Cette modification a permis à ProloGaal de découpler la profondeur possible d'arbre de preuve. Pour un benchmark Prolog simple, consistant simplement à compter jusqu'à  $N$ , ProloGaal a pu passer d'un  $N$  maximal de 5'000, à un  $N$  maximal avoisinant les 200'000.

Le code relatif à ces changements est un peu trop long pour être documenté dans ce rapport. Néanmoins, cette pratique consistant à transformer un code récursif en un autre ne possédant pas de récursivité est un principe connu et documenté<sup>9</sup>, et l'application de celui-ci n'a pas soulevé de grosses difficultés pour ProloGaal.

## 3.2 Ajouts

Ce chapitre documente les nouvelles fonctionnalités intégrées à Prolog durant ce projet.

### 3.2.1 Opérateurs usuels Prolog

Il est possible dans Prolog de comparer des termes entre eux via les opérateurs usuels. On peut par exemple comparer si un nombre est plus grand qu'un autre via `>(LeftNum,RightNum)`. Pour ce projet, seuls les opérateurs `=/2`, `>/2`, `>=/2`, `</2` et `=</2` ont été implémentés.

Mis à part le prédicat `=/2`, ces prédicats builtins opérateurs sont implémentés de manière très similaire, observant simplement si la condition induite par cet opérateur et ses deux arguments est vraie ou non, et retourne un succès ou un échec Prolog en conséquence. Le prédicat `=/2`, lui, va simplement unifier ses deux arguments et retournera un succès si l'unification s'est bien déroulée.

---

9. Boğaziçi UNIVERSITY. *HOW TO CONVERT A RECURSIVE ALGORITHM TO A NON-RECURSIVE ONE*. URL : <https://www.cmpe.boun.edu.tr/~akin/cmpe160/recursion.html>.

### 3.2.2 Évaluation de code étranger

GraalVM permet d'exécuter du code provenant d'un langage étranger au langage hôte. Pour cela, il est nécessaire que le langage hôte offre une interface à l'utilisateur pour pouvoir exécuter le langage étranger désiré.

Pour ajouter cette nouvelle fonctionnalité dans ProloGraal, il a été décidé de créer un nouveau prédicat builtin `polyglot_eval/3`, initialement non présent dans la librairie standard Prolog. Ce prédicat, comme indiqué par son arité<sup>10</sup>, accepte trois arguments. Voici une utilisation typique de ce prédicat :

```
polyglot_eval(A, 'js', '3 + 4').
```

Le premier argument, ici `A`, est un terme Prolog. Nous utilisons ici une variable Prolog pour stocker le résultat de l'exécution, mais il aurait tout autant été possible d'écrire le nombre 7, et le prédicat aurait également donné un succès.

Le second argument est un atome Prolog. Celui-ci doit représenter le languageID du langage désiré. Chaque langage intégré à GraalVM se doit de posséder un id, celui de ProloGraal est `pl` par exemple. Ici, `js` est entouré d'apostrophes pour indiquer au parser que le texte est bien un atome Prolog, mais ce texte serait également "parsé" en tant qu'atome Prolog sans les apostrophes, contrairement au troisième argument, car lui possède des espaces.

Le troisième argument est un atome Prolog, représentant le code étranger à exécuter. Ce code se devra d'être syntaxiquement juste, car ProloGraal n'intègre actuellement pas le mécanisme d'exception Prolog. Si un code exécuté est syntaxiquement incorrect, l'exception arrêtera le programme et sera retournée.

Voici l'implémentation Java de l'exécution de ce builtin :

```
public ProloGraalBoolean returnValue(ProloGraalTerm left,
    ProloGraalTerm center, ProloGraalTerm right) {
    Context context =
        Context.newBuilder().allowAllAccess(true).build();
    Value res = context.eval(center.asAtom().getName(),
        right.asAtom().getName());
    ProloGraalTerm resAsTerm = genericValueToProloGraalTerm(res);
    if(resAsTerm != null && left.unify(resAsTerm))
        return new ProloGraalSuccess();
    return new ProloGraalFailure();
}
```

On voit alors que l'instance de la classe `Context`<sup>11</sup> créée nous permet d'évaluer le code étranger de la même manière que le prédicat `polyglot_eval/3`. la valeur de retour sera stockée dans

---

10. UNSW COMPUTING. *Prolog arity*. URL : <http://www.cse.unsw.edu.au/~billw/dictionaries/prolog/arity.html>.

11. GraalVM TEAM. *Context JavaDoc*. URL : <https://www.graalvm.org/sdk/javadoc/org/graalvm/polyglot/Context.html>.

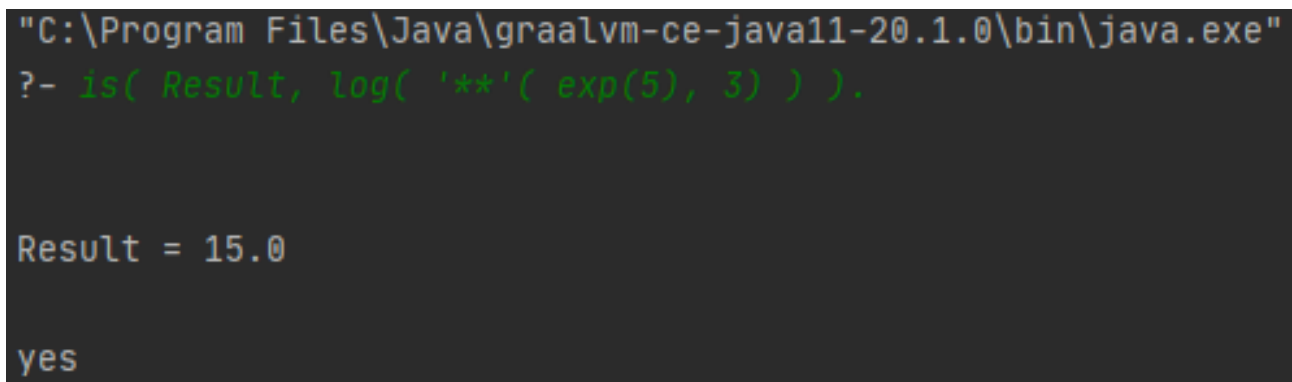
une instance de la classe `Value`<sup>12</sup>, qui elle représente une valeur générique Truffle, que l'on peut par la suite questionner pour connaître sa nature (chaîne de caractère, nombre, booléen, etc...). Cette valeur est directement transformée en terme Prolog, puis unifiée avec le premier argument du prédicat, le `ProloGaalTerm left`. Selon le résultat de l'unification avec `left`, `polyglot_eval/3` retourne un succès ou un échec Prolog.

### 3.3 Conclusion

Ce chapitre montre les résultats les plus pertinents sur les différents ajouts et modifications apportés aux fonctionnalités de ProloGaal, et s'ensuit une conclusion générale sur les fonctionnalités.

#### 3.3.1 Résultat du builtin `is/2`

Voici un exemple d'utilisation du prédicat builtin `is/2` contenant des opérateurs d'arités<sup>13</sup> différentes, comme visible sur la figure 2. On peut voir que `is/2` gère sans soucis l'expression  $\ln(e^{5^3})$ , qui est évalué à 15 et unifié à la variable `Result`.



```
"C:\Program Files\Java\graalvm-ce-java11-20.1.0\bin\java.exe"
?- is( Result, log( '**'( exp(5), 3) ) ).

Result = 15.0

yes
```

FIGURE 2 – Résultat de l'exécution du prédicat builtin `is/2`

#### 3.3.2 Résultat du builtin `polyglot_eval/3`

Voici un exemple d'utilisation du prédicat builtin `polyglot_eval/3` évaluant du code JavaScript, comme visible sur la figure 3. On peut sans autre calculer une boucle `for` par exemple, opération usuellement fastidieuse à reproduire en syntaxe Prolog. On peut voir que la dernière déclaration du code JavaScript est simplement `a`; car dans l'implémentation JavaScript pour GraalVM, la valeur de la dernière déclaration est retournée lorsque le code évalué est de type script.

---

12. GraalVM TEAM. *Value JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/org/graalvm/polyglot/Value.html>.

13. UNSW COMPUTING. *Prolog arity*. URL : <http://www.cse.unsw.edu.au/~billw/dictionaries/prolog/arity.html>.

```
"C:\Program Files\Java\graalvm-ce-java11-20.1.0\bin\java.exe" -Xmx2048m -Xms2048m -Dtruff
?- polyglot_eval( Result, 'js', 'var a = 0; for(let i = 0; i <= 10; i++){ a+=i; } a;' ).

Result = 55

yes
```

FIGURE 3 – Résultat de l'exécution du prédicat builtin `polyglot_eval/3`

### 3.3.3 Conclusion des fonctionnalités

Les corrections apportées à ProloGaal ont sensiblement renforcé sa représentation interne de Prolog via l'ajout des `ProloGaalQueries`. Elles ont également offert la possibilité de tester les performances de ProloGaal, lui permettant de résoudre des requêtes Prolog engendrant des arbres de preuve bien plus profonds ( $\sim$ facteur  $\times 40$ ).

Les nouvelles fonctionnalités implémentées à ProloGaal offrent un panel de possibilités plus riche pour l'écriture un programme Prolog. En effet, les opérateurs usuels sont très utiles pour engendrer certains comportements, et très utilisés par les développeurs Prolog. Le prédicat `polyglot_eval/3`, lui, met à disposition une fonctionnalité particulière sortant du cadre classique de Prolog, permettant l'évaluation de codes au langage étranger, comme démontré dans la section "Résultat du builtin `polyglot_eval/3`" du rapport.

Il reste cependant un grand nombre de fonctionnalités manquantes à ProloGaal, tels certains prédicats builtins utiles et populaires comme par exemple le prédicat `!/0`, `assert/1` et `retract/1`. Les différentes fonctionnalités qu'il serait judicieux d'ajouter à ProloGaal sont documentées dans le chapitre "Améliorations possibles".

## 4 Optimisations

Ce chapitre décrit les différentes optimisations apportées à ProloGraal durant ce projet. La section "Autres mécanismes offerts par Truffle", quant à elle, décrit les différents mécanismes que Truffle met à disposition et qui n'ont pas pu être implémentés dans GraalVM par manque de temps ou d'opportunité.

### 4.1 Profiling

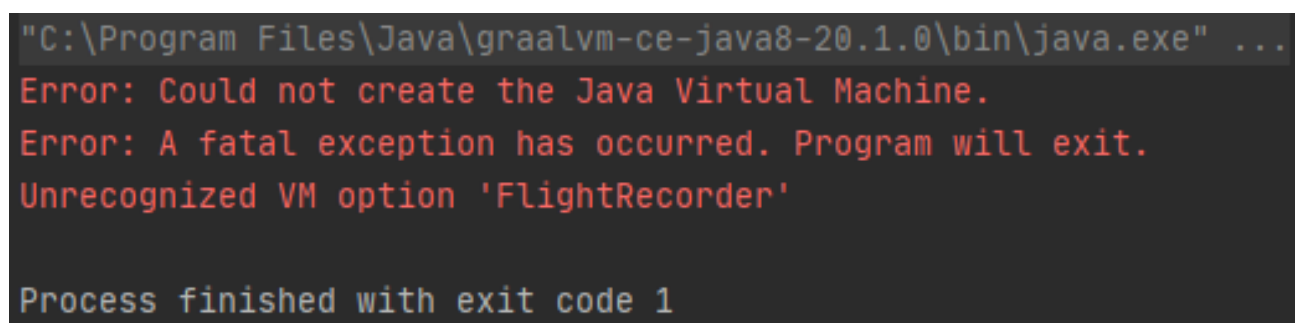
Cette section décrit les différents éléments relatifs au profiling de ProloGraal. Des explications sur les pré-requis nécessaires pour pouvoir profiler un programme s'exécutant sous une machine virtuelle GraalVM sont détaillées, suivi des résultats obtenus via profiling et optimisations apportées.

#### 4.1.1 Pré-requis GraalVM

Pour pouvoir profiler l'exécution d'un programme sous une machine GraalVM, il est nécessaire que la version du JDK GraalVM utilisé soit un build Java 11. Il existe en effet deux différents builds d'une même version de GraalVM, un build pour Java 8 et un autre pour Java 11, comme observable sur la page de téléchargement du build GraalVM 20.1.0<sup>14</sup>.

Lors des précédentes versions de ProloGraal, le JDK GraalVM utilisé était le build 19.2.0.1 pour Java 8. Il a premièrement été décidé de passer sur le build 20.1.0 pour Java 8, puis de passer sur le build pour Java 11 par la suite.

De nombreuses tentatives de profiling ont été effectuées entre ces deux décisions, mais menaient à des échecs du fait que le build Java 8 ne supporte pas certaines fonctionnalités pour le profiling. Des recherches ont été effectuées sur le message d'erreur retourné, visible sur la figure 4, mais ne furent pas fructueuses et menèrent alors à une perte de temps de travail notable. De l'aide apportée par le Professeur Frédéric Bapst permit alors de cibler la source du problème, car les options de profiling fonctionnaient pour le build GraalVM Java 11 que M. Bapst a utilisé.



```
"C:\Program Files\Java\graalvm-ce-java8-20.1.0\bin\java.exe" ...  
Error: Could not create the Java Virtual Machine.  
Error: A fatal exception has occurred. Program will exit.  
Unrecognized VM option 'FlightRecorder'  
  
Process finished with exit code 1
```

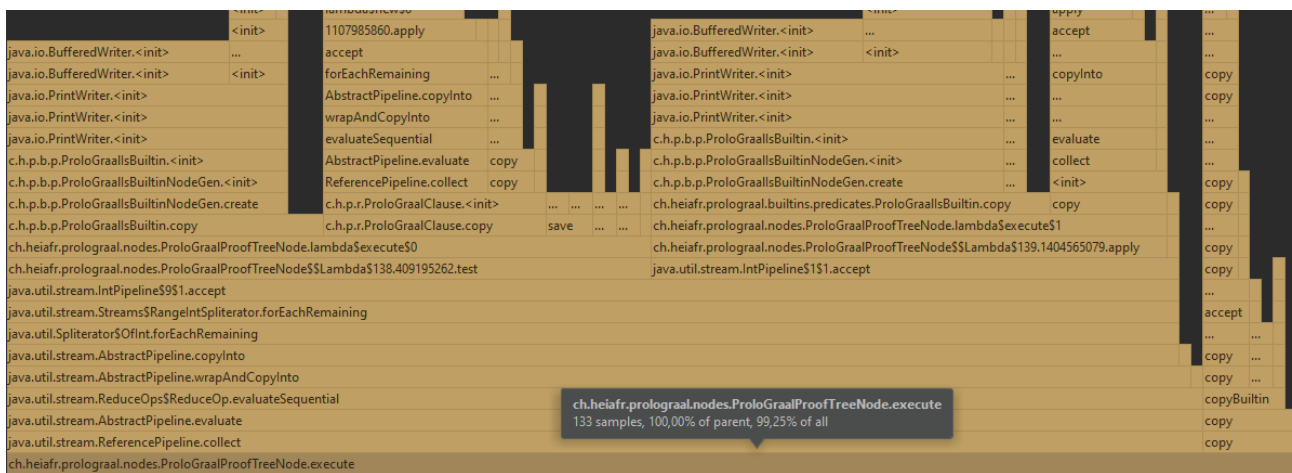
FIGURE 4 – Message d'erreur retourné lors du profiling d'un programme sous GraalVM Java 8

14. GraalVM TEAM. *Github release of JVM 20.1.0*. URL : <https://github.com/graalvm/graalvm-ce-builds/releases/tag/vm-20.1.0>.



### 4.1.2 Optimisation apportée

Il a été décidé de profiler le benchmark présenté au chapitre "Tests de performance" pour observer les portions du code les plus visitées lors de son exécution. Sur la figure 5, on peut observer un profiling effectué avec l'IDE IntelliJ IDEA version 2020.1.2<sup>15</sup>. La largeur des barres oranges présentes sur la figure 5 représente la portion du temps dans laquelle ProloGaal se trouvait dans la méthode inscrite sur cette même barre lors de l'exécution du benchmark. On peut par exemple observer les informations relatives à la barre la plus basse de l'image, représentant la méthode `ProloGaalProofTreeNode.execute()`, indiquant que cette méthode est parcourue 99.25% du temps lors de l'exécution du benchmark. Toutes les barres présentes au dessus de celle-ci sont des méthodes appelées lors de l'exécution de la méthode `ProloGaalProofTreeNode.execute()`.

FIGURE 5 – Profiling du benchmark utilisant le builtin `is/2`

Précédemment, le prédicat builtin `is/2`, massivement utilisé lors de ce benchmark, initialisait un `PrintWriter` pour pouvoir communiquer à l'utilisateur via l'interpréteur les erreurs de syntaxe lors de l'utilisation du prédicat `is/2`. Grâce au profiling présent sur la figure 5, l'initialisation gourmande du `PrintWriter` a été mise en évidence. Ces messages informatifs quant à l'utilisation sur prédicat `is/2` ne sont initialement pas présent lors de son utilisation dans un interpréteur classique Prolog. Pour optimiser la vitesse d'exécution du prédicat `is/2`, il a été décidé de supprimer l'utilisation du `PrintWriter`.

Après suppression du `PrintWriter`, l'exécution du benchmark s'est vue grandement accélérée. Des informations supplémentaires à propos des gains mesurés sont disponibles au chapitre "Tests de performance" du rapport.

Mis à part l'utilisation du `PrintWriter`, la figure 5 met également l'utilisation massive de méthodes `copy()` tout au long de l'exécution de la méthode `ProloGaalProofTreeNode.execute()`.

15. JETBRAINS. *Download IntelliJ*. URL : <https://www.jetbrains.com/idea/download/other.html>.

Ces copies sont là pour préserver l'état non-unifié des variables utilisées dans les prédicats builtins. Cela permet alors d'utiliser plusieurs fois un même prédicat builtin, car l'unification Prolog est utilisée lors de la résolution des buts au sein de la méthode `ProloGaalProofTreeNode.execute()`. Cela signifie que si l'on ne copie pas le prédicat builtin désiré, les variables du prédicat builtin seraient alors unifiées, et il ne serait plus possible de les unifier avec d'autres valeurs. Ces détails techniques sur la résolution d'un but Prolog par ProloGaal mettent en avant que ce mécanisme de copie de règles est actuellement très utilisé pour éviter les effets non désirés de l'unification Prolog dans ce contexte. Ce mécanisme de copie crée une nouvelle instance Java de `ProloGaalClause`, ce qui nécessite du temps d'exécution conséquent, ainsi qu'une utilisation notable de la mémoire de la heap. Une optimisation possiblement meilleure aurait été de modifier le mécanisme de résolution des buts de ProloGaal plutôt que de supprimer l'utilisation du `PrintWriter`, mais cela représente une tâche bien plus importante et n'était pas prévue sur le planning initial.

## 4.2 Migration des prédicats builtins

Cette section décrit la transformation des prédicats builtins implémentés dans ProloGaal, ceux-ci passant de classes Java traditionnelles en classes représentant des noeuds Truffle instrumentables.

### 4.2.1 Définition du `TypeSystem`

Les noeuds instrumentables Truffle, introduits dans le chapitre suivant, définissent les noeuds enfants qu'ils peuvent posséder via un `TypeSystem`<sup>16</sup>, interface présente dans la librairie Truffle. Des tentatives ont été effectuées pour "caster" implicitement les termes implémentés en tant que `ProloGaalTerm`, pour que les méthodes présentes dans les noeuds Truffle acceptant des `ProloGaalTerm` puissent également accepter les classes enfants de `ProloGaalTerm`. Ce mécanisme, pourtant présent dans le `TypeSystem` de `SimpleLanguage`<sup>17</sup>, n'a été finalement pas implémenté, car les différentes tentatives ne furent pas fructueuses.

Bien que les casts implicites ne furent pas implémentés, le `TypeSystem` de ProloGaal, `ProloGaalTypes`, Permet bien l'utilisation des différents éléments de ProloGaal au sein des noeuds instrumentables Truffle. Voici la version actuelle de la classe `ProloGaalTypes` :

```
@TypeSystem({int.class, double.class, boolean.class})
public abstract class ProloGaalTypes {
    @TypeCheck(ProloGaalTerm.class)
    public static boolean isProloGaalTerm(Object value) {
        return value instanceof ProloGaalTerm;
    }
}
```

16. GraalVM TEAM. *@TypeSystem JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/dsl/TypeSystem.html>.

17. GraalVM TEAM. *SLTypes - Simple Language Repository*. URL : <https://github.com/graalvm/simplelanguage/blob/master/language/src/main/java/com/oracle/truffle/sl/nodes/SLTypes.java>.

```

    // same for ProloGaalBoolean and ProloGaalObject
}

```

#### 4.2.2 Noeuds instrumentables Truffle

Un noeud instrumentable Truffle est une classe Java étendant la classe `Node`<sup>18</sup>, utilisant un `TypeSystem`<sup>19</sup> valide, implémentant la classe `InstrumentableNode`<sup>20</sup> ainsi que les méthodes `isInstrumentable()` et `createWrapper()`, et finalement utilisant l'annotation `@GenerateWrapper`<sup>21</sup>. Si la méthode `isInstrumentable()` retourne vrai, le noeud peut alors être instrumenté par Truffle.

Lorsque ces conditions sont remplies, il est possible de spécifier une méthode abstraite représentant l'opération de ce noeud instrumentable. Il est cependant nécessaire que cette méthode abstraite suive un certain format, décrit plus loin dans cette section. Il est par la suite possible d'implémenter cette opération via l'annotation Truffle `@Specialization`<sup>22</sup> si celle-ci est placée au dessus de la méthode désignée à implémenter spécialiser l'opération de ce noeud instrumentable.

Un noeud instrumentable Truffle possède les mécanismes nécessaires pour laisser Truffle accéder aux informations de ce noeud, et de prendre des décisions en conséquence pour optimiser à la volée le code exécuté pour résoudre ce noeud.

Pour implémenter ce principe de noeuds instrumentables Truffle dans ProloGaal, il a été décidé de transformer les prédicats builtins. Ceux-ci implémentaient simplement une méthode abstraite héritée qui représentait l'action de ce builtin, et après modification, implémentent leurs comportements via ce système de noeuds instrumentables.

Les règles builtin Prolog sont des règles qui ne possèdent pas de liste de buts, de la même manière que les faits Prolog. Cela signifie que la tête de règle, atome ou structure Prolog, est la seule composante de la règle builtin. Pour implémenter ce système de noeuds instrumentables, il a été décidé que la tête de règle du prédicat builtin soit un noeud instrumentable. Une classe abstraite `ProloGaalBuiltinHeadNode`, respectant les conditions nécessaires pour être un noeud instrumentable Truffle, a été mise en place pour représenter la classe mère de toutes les têtes de règle builtin :

```
@TypeSystemReference(ProloGaalTypes.class)
```

18. GraalVM TEAM. *Node JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/nodes/Node.html>.

19. GraalVM TEAM. *@TypeSystem JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/dsl/TypeSystem.html>.

20. GraalVM TEAM. *InstrumentableNode JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/instrumentation/InstrumentableNode.html>.

21. GraalVM TEAM. *GenerateWrapper JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/instrumentation/GenerateWrapper.html>.

22. GraalVM TEAM. *Specialization JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/dsl/Specialization.html>.

```

@NodeInfo(shortName = "ProloGaalBuiltinHeadNode")
@GenerateWrapper
public abstract class ProloGaalBuiltinHeadNode extends
    ProloGaalGenericNode implements InstrumentableNode {
    protected final ProloGaalTerm<?> value;

    public ProloGaalBuiltinHeadNode(ProloGaalBuiltinHeadNode
        other){
        this.value=other.value;
    }

    public abstract ProloGaalBoolean
        executeBuiltin(VirtualFrame frame);

    public boolean isInstrumentable() { return true; }

    public WrapperNode createWrapper(ProbeNode probe) {
        // ASTNodeWrapper is generated by @GenerateWrapper
        return new ProloGaalBuiltinHeadNodeWrapper(this, this,
            probe);
    }
}

```

Un détail important est que les noeuds instrumentables Truffle doivent également fournir un constructeur par défaut ou un constructeur de copie.

On peut voir sur le fragment de code que le `TypeSystem` de `ProloGaal` est défini en tant que `ProloGaalTypes` de par l'utilisation de l'annotation `@TypeSystemReference`, et le `TypeSystem` sera également délégué aux classes enfants. La ou les méthodes qui seront spécialisées par les classes enfants seront déterminées par Truffle en tant qu'opérations du noeud, pour autant que cette ou ces méthodes respectent la syntaxe des méthodes spécialisables Truffle :

```
public abstract {Type} execute[Text](VirtualFrame);
```

Dans le cas de la classe `ProloGaalBuiltinHeadNode`, la méthode spécialisable est `executeBuiltin()`, car elle respecte la syntaxe requise.

Le mécanisme de spécialisation de noeuds instrumentables Truffle est décrit lors du chapitre suivant, expliquant également comment intégrer les classes enfants de `ProloGaalBuiltinHeadNode`.

### 4.3 Spécialisation des noeuds Truffle

Cette section décrit le fonctionnement du mécanisme de spécialisation offert par Truffle via l'annotation `@Specialization`<sup>23</sup>. Ce mécanisme étant implémenté dans les classes enfants de

---

23. GraalVM TEAM. *Specialization JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/dsl/Specialization.html>.

`ProloGraalBuiltinHeadNode`, une documentation de l'application de ce mécanisme est également fournie

### 4.3.1 Mécanisme de spécialisation

Lors de la spécialisation d'une opération, un élément très important est le nombre d'arguments de la méthode spécialisée. En effet, le nombre d'arguments de la méthode devra être égal au nombre d'enfants au sens noeud Truffle que cette classe possède. En effet, la bonne pratique Truffle pour gérer les différents arguments d'entrée des opérations définies est de les annoncer via l'annotation Truffle `@NodeChild`<sup>24</sup>. Voyons un exemple d'utilisation de cette annotation, avec également une méthode spécialisée :

```
@NodeInfo(shortName = "LanguageExecutableNode")
@NodeChild(value = "left", type = LanguageExecutableNode.class)
@NodeChild(value = "right", type = LanguageExecutableNode.class)
public abstract LanguageExecutableNode extends
    LanguageGenericNode{
    //...
    @Specialization
    public LanguageObject specializeOp( LanguageObject left,
        LanguageObject right ) { /*...*/ }
    //... }
```

Il y a alors deux noeuds déclarés pour cette classe fictive `LanguageExecutableNode`, étant de même type. On voit alors que le nombre d'argument nécessaire pour les méthode spécialisées est de deux.

Le type de la valeur de retour des méthodes spécialisées doit être de même type que celui de la valeur de retour de l'opération que l'on désire spécialiser. Le type peut également hériter du type de l'opération. On voit que le type des arguments dans cet exemple est `LanguageObject`, le même type que celui de la valeur de retour. Cela signifie que les noeuds enfants au sens Truffle seront déjà exécutés lorsque cette méthode spécialisée sera initiée, et que les arguments que nous recevons sont les valeurs de retour des noeuds enfants exécutés.

Ce système de noeuds enfants est utilisé dans ProloGraal pour représenter les arguments des prédicats builtins. Les prédicats sont eux-même des noeuds, mais ceux-ci possèdent des noeuds enfants de type `ProloGraalTermNode`, noeuds instrumentables Truffle représentant les termes Prolog en tant que noeuds. des explications sur l'application de l'annotation `@NodeChild` sont présentes dans le chapitre "Implémentation des builtins en tant que noeuds instrumentables Truffle".

---

24. GraalVM TEAM. *@NodeChild JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/dsl/NodeChild.html>.

### 4.3.2 Mécanisme de spécialisation

Le mécanisme de spécialisation Truffle permet aux noeuds instrumentables Truffle d'emprunter des chemins d'exécution différents selon les conditions d'exécution de celui-ci. Pour cela, il est nécessaire qu'au moins une méthode abstraite, désignée comme l'opération du noeud, soit déclarée. Celle-ci doit suivre le format indiqué au chapitre "Noeuds instrumentables Truffle" pour être considérée par Truffle comme opération de ce noeud instrumentable.

Lorsque nous déclarons nos spécialisations, il est important de respecter un certain ordre de déclaration. En effet, imaginons une classe B qui hérite d'une classe A, le seul ordre possible pour la déclaration de ces trois méthodes est le suivant :

```
@Specialization(rewriteOn = LanguageException.class)
public A specOne(B left, B right){ /* ... */ }

@Specialization(guards = {"left.isA()", "right.isA()"})
public A specTwo(B left, A right){ /* ... */ }

@Specialization
public A specThree(A left, A right){ /* ... */ }
```

En effet, si l'argument `right` de `specThree()` était de type B, une erreur de compilation serait survenue, car B est une classe héritant de A, et que `right` avait déjà été "généralisé" à la classe A avec la méthode `specTwo()`.

L'annotation `@Specialization` permet également d'appliquer des restrictions supplémentaires sur les spécialisations, comme présenté dans le fragment de code. L'argument `rewriteOn` permet ainsi à la méthode `specOne()` d'automatiquement exécuter la prochaine spécialisation si l'exécution de `specOne()` venait à lancer l'exception fictive `LanguageException`. L'argument `guards`, lui, permet de spécifier une liste d'expressions retournant des booléens, et cette méthode est sélectionnée uniquement si toutes les expressions sont évaluées à "vrai".

### 4.3.3 Implémentation des builtins en tant que noeuds instrumentables Truffle

Il est important de noter que les classes enfants, les implémentations des builtins en tant que builtin head nodes, n'héritent pas directement de la classe `ProloGaalBuiltinHeadNode`. Sur la figure 6, on peut observer les liens d'héritage entre les différentes classes relatives aux noeuds builtin ProloGaal.

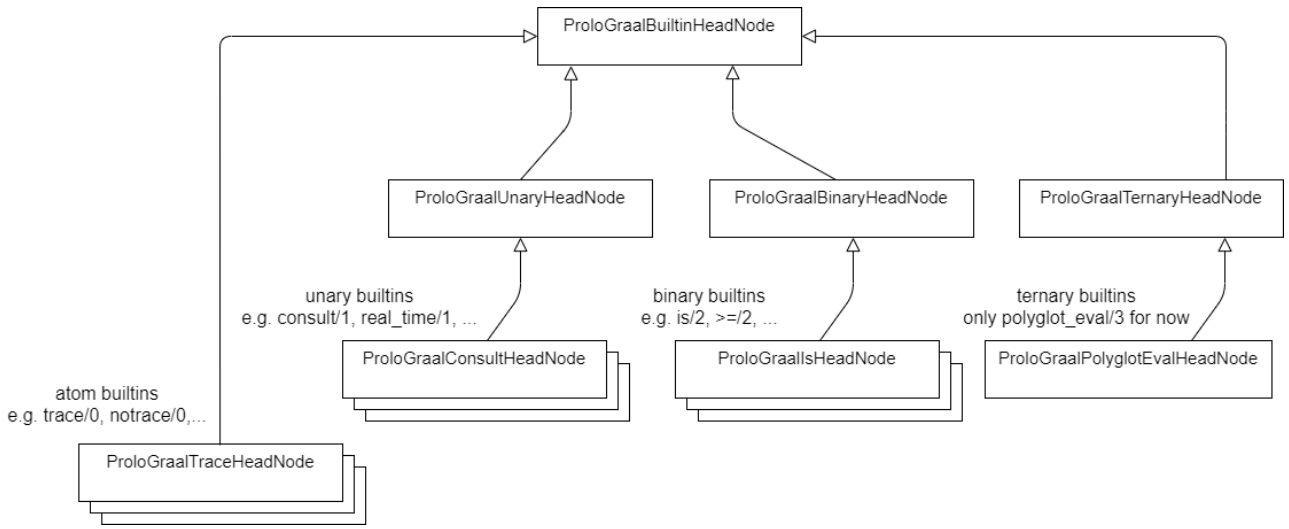


FIGURE 6 – Diagramme représentant les héritages entre les différents noeuds builtins

Observons par exemple le contenu de la classe `ProloGaalTernaryHeadNode` pour découvrir les propriétés transmises par cette classe intermédiaire :

```

@NodeChild(value = "left", type = ProloGaalTermNode.class)
@NodeChild(value = "center", type = ProloGaalTermNode.class)
@NodeChild(value = "right", type = ProloGaalTermNode.class)
public abstract class ProloGaalTernaryHeadNode extends
    ProloGaalBuiltinHeadNode {
    public ProloGaalTernaryHeadNode(ProloGaalTerm<?> value) {
        /* ... */
    }
}

```

La classe `ProloGaalTernaryHeadNode` permet de définir une seule fois les trois arguments de tous les prédicats builtins héritant cette classe. Il existe une alternative Truffle pour simplement déclarer les noeuds enfants Truffle d'une classe en tant qu'une liste de noeuds Truffle<sup>25</sup>, mais l'implémentation de cette alternative amenait à des erreurs et c'est pourquoi il a été décidé d'utiliser ces classes intermédiaires.

Regardons maintenant le contenu de la classe `ProloGaalPolyglotEvalHeadNode`, héritant de `ProloGaalTernaryHeadNode` :

```

@NodeInfo(shortName = "ProloGaalPolyglotEvalHeadNode")
public abstract class ProloGaalPolyglotEvalHeadNode extends
    ProloGaalTernaryHeadNode {

```

25. GraalVM TEAM. *SLBuiltinNode*, line 61 for `@NodeChild` array - *Simple Language Repository*. URL : <https://github.com/graalvm/simplelanguage/blob/master/language/src/main/java/com/oracle/truffle/sl/builtins/SLBuiltinNode.java>.

```

@Specialization(guards =
    {"center.isAtom()", "right.isAtom()"})
public ProloGaalBoolean returnValue(ProloGaalTerm left,
    ProloGaalTerm center, ProloGaalTerm right ) { /* ... */
}

@Specialization
public ProloGaalBoolean wrongParams(ProloGaalTerm left,
    ProloGaalTerm center, ProloGaalTerm right ) { /* ... */
}
}

```

Cette classe exécute la spécialisation `returnValue()` si les arguments `center` et `right` sont de type atome. Nous n'utilisons pas le cast direct de ces arguments en exigeant un `ProloGaalAtom` `center` par exemple, car on pourrait très bien recevoir une variable dont la valeur est un atom. Si la `guards` de `returnValue()` n'est pas respectée, la méthode `wrongParams()` s'exécutera pour annoncer que les paramètres de l'opération sont incorrects.

On exécutera par la suite ces noeuds instrumentables durant la résolution des buts Prolog dans la méthode `ProloGaalProofTreeNode.execute()`. Un grand nombre de changements ont été effectués dans `ProloGaal` pour gérer cette nouvelle manière d'exécuter les prédicats builtins, mais ne sont pas présentés dans ce rapport car ils touchent essentiellement au parsing et à la résolution de buts par `ProloGaal`.

Peu de prédicats builtins transformés en noeuds instrumentables spécialisent plusieurs fois la même opération. Cela est principalement dû à la simplicité de la classe `ProloGaalTypes`, qui ne nous permet actuellement pas de caster implicitement certaines valeurs en d'autres. Il aurait par exemple été possible de préciser l'utilisation de `ProloGaalAtoms` pour les arguments `center` et `right` du prédicat `polyglot_eval/3` si le `TypeSystem` castait automatiquement les variables étant unifiées à un atome en tant que l'atome en question.

## 4.4 Autres mécanismes offerts par Truffle

Truffle offre d'autres mécanismes pour optimiser automatiquement le code d'un langage. Les sections suivantes décrivent certains de ces mécanismes. Ces mécanismes ne sont pas présents dans `ProloGaal`, principalement par manque de temps pour pouvoir d'avantage les explorer, mais également parfois par manque d'opportunité (comme par exemple pour l'annotation `@ExplodeLoop`).

### 4.4.1 Annotation `@Cached`

Cette annotation permet de déclarer des paramètres supplémentaires aux méthodes spécialisées. Observons l'exemple suivant utilisant l'annotation `@Cached`<sup>26</sup>, grandement inspiré d'un code

---

26. GraalVM TEAM. *@Cached JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/dsl/Cached.html>.



provenant du projet GraalVM SimpleLanguage<sup>27</sup> :

```
public abstract class LanguageEvalBuiltin extends
    LanguageBuiltinNode {
    @Specialization(guards = {"stringsEqual(cachedId, id)",
        "stringsEqual(cachedCode, code)"})
    public Object evalCached(String id, String code,
        @Cached("id") String cachedId,
        @Cached("code") String cachedCode,
        @Cached("create(parse(id, code, context))")
            DirectCallNode callNode) { /* ... */ }

    @Specialization
    public Object evalUncached(String id, String code) { /* ...
        */ }
}
```

Grâce à l'annotation `@Cached`, le paramètre `cachedId` est initialisé qu'une seule fois à l'entrée de la méthode `evalCached()`, puis est considéré comme final et n'est plus modifié. Si ce noeud instrumentable est à nouveau exécuté, et que les arguments passés en paramètre sont les mêmes que les arguments "cached", le noeud caché `DirectCallNode callNode` est alors appelé, car non nécessaire de l'instancier à nouveau. Nous économisons donc l'initialisation d'un nouveau `DirectCallNode` lors de l'appel répété de cette spécialisation avec les mêmes paramètres.

#### 4.4.2 Annotation `@ExplodeLoop`

L'annotation `@ExplodeLoop`<sup>28</sup> permet de dérouler automatiquement des boucles `for` contenues dans des méthodes. Il est cependant nécessaire que le nombre d'itération de la boucle soit une constante.

#### 4.4.3 Classe `LoopNode`

La classe `LoopNode`<sup>29</sup> permet d'optimiser automatiquement des boucles `while`. Il est nécessaire pour cela d'utiliser d'autres éléments, comme par exemple créer une classe implémentant l'interface `Truffle RepeatingNode`.

---

27. GraalVM TEAM. *SLEvalBuiltin - Simple Language Repository*. URL : <https://github.com/graalvm/simplelanguage/blob/master/language/src/main/java/com/oracle/truffle/sl/builtins/SLEvalBuiltin.java>.

28. GraalVM TEAM. *@ExplodeLoop JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/nodes/ExplodeLoop.html>.

29. GraalVM TEAM. *LoopNode JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/nodes/LoopNode.html>.

Cette classe pourrait par exemple être utilisée pour optimiser la nouvelle façon de résoudre un but par ProloGraal, car la suppression de la récursivité a introduit l'utilisation d'une boucle `while`. Ce mécanisme n'étant pas présent dans ProloGraal, il ne semble pas pertinent d'expliquer dans les détails l'implémentation d'un tel mécanisme.

## 4.5 Conclusion

En conclusion sur les optimisations, le profiling a permis d'apporter des informations claires sur ce qui ralentit actuellement ProloGaal et également corriger un des problèmes découverts, tandis que les migrations des prédicats builtins en noeuds instrumentables Truffle ont rendu ProloGraal disponible pour les différentes optimisations manuelles proposées par Truffle, comme présenté à la section "Autres mécanismes offerts par Truffle".

## 5 Tests de performance

Trois tests de performance, ou benchmarks, ont été réalisés durant le projet. Voici le programme Prolog utilisé pour effectuer ces benchmarks, nommé "18\_benchmark.pl.excluded" :

```
benchmark_text('time for N equal ').
benchmark_one:-
    do_benchmark(100).

% benchmark_...:-

benchmark_five:-
    do_benchmark(50000).

do_benchmark(N):-
    benchmark_text(BenchText),
    benchmark(N,Time),
    write(BenchText), write(N), write(': '), write(Time).

benchmark(N, Time):-
    real_time(A),
    benchmark(N),
    real_time(B),
    is(C, '-'(B,A)),
    '='(Time,C).

benchmark(0).

benchmark(N):-
    is(N1, '-'(N,1)),
    benchmark(N1).
```

Pour pouvoir comparer les temps obtenus via ProloGraal avec une évaluation plus classique de Prolog, ce benchmark a également été testé sur les interpréteurs SWI-Prolog et GNU Prolog, comme visible sur les figures 7 et 8.

```

?- benchmark_one.
time for N equal 100: 0.0
true .

?- benchmark_two.
time for N equal 1000: 0.0
true .

?- benchmark_three.
time for N equal 10000: 0.0
true .

?- benchmark_four.
time for N equal 30000: 0.0010190010070800781
true .

?- benchmark_five.
time for N equal 50000: 0.0010218620300292969
true .

```

FIGURE 7 – Résultat du test de performance  
(en secondes) sur SWI-Prolog

```

| ?- benchmark_one.
time for N equal 100: 0

true ?

yes
| ?- benchmark_two.
time for N equal 1000: 0

true ?

yes
| ?- benchmark_three.
time for N equal 10000: 1

true ?

yes
| ?- benchmark_four.
time for N equal 30000: 2

true ?

(15 ms) yes
| ?- benchmark_five.
time for N equal 50000: 3

true ?

(16 ms) yes

```

FIGURE 8 – Résultat du test de performance  
(en millisecondes) sur GNU Prolog

Les trois tests effectués sur ProloGaal sont également présents sur les figures 9, 10 et 11.

On peut observer les temps du test effectué le 22 juin, qui pour  $N = 50'000$  semble s'exécuter en moyenne  $\sim 1000x$  plus lentement que les interpréteurs traditionnels pour Prolog. Ce test a été effectué peu de temps après la suppression de la récursivité lors de la résolution d'un but, alors que le prédicat `is/2` initialisait encore un `PrintWriter` dans son constructeur, et c'est ce qui provoquait des temps si lents pour des  $N$  élevés.

Le deuxième test, effectué le 26 juin, a été planifié après avoir supprimé l'initialisation du `PrintWriter` par le prédicat `is/2`. Il démontre un net gain de performance après cette minime modification.

Le troisième test a été effectué après la migration des prédicats builtins en noeuds instrumentables Truffle. La différence de temps observable entre le deuxième et troisième test est dû à la fluctuation du temps d'exécution plutôt qu'à une réelle amélioration. Il est maintenant su grâce au profiling que la meilleure manière d'optimiser ProloGaal actuellement est de modifier son implémentation naïve de la résolution d'un but Prolog.

```

?- benchmark_one.

time for N equal 100: 14

yes
?- benchmark_two.

time for N equal 1000: 50

yes
?- benchmark_three.

time for N equal 10000: 343

yes
?- benchmark_four.

time for N equal 30000: 648

yes
?- benchmark_five.

time for N equal 50000: 1714

yes

```

FIGURE 9 – Test du 22.06

```

?- benchmark_one.

time for N equal 100: 12

yes
?- benchmark_two.

time for N equal 1000: 53

yes
?- benchmark_three.

time for N equal 10000: 141

yes
?- benchmark_four.

time for N equal 30000: 388

yes
?- benchmark_five.

time for N equal 50000: 584

yes

```

FIGURE 10 – Test du 26.06

```

?- benchmark_one.

time for N equal 100: 19

yes
?- benchmark_two.

time for N equal 1000: 44

yes
?- benchmark_three.

time for N equal 10000: 224

yes
?- benchmark_four.

time for N equal 30000: 331

yes
?- benchmark_five.

time for N equal 50000: 473

yes

```

FIGURE 11 – Test du 14.07

## 6 Tests unitaires

Cette section discute des tests unitaires effectués durant le projet, ainsi qu'une marche à suivre nécessaire pour pouvoir tester ProloGaal sous le JDK GraalVM 20.1.0 (pour Java 8 et 11).

### 6.1 Problème amené par l'upgrade du JDK GraalVM

Des tests unitaires ont été repris depuis les précédents projets de semestre, et ceux-ci pouvaient être normalement exécutés via Maven sous GraalVM 19.2.0.1. Avec l'upgrade du JDK GraalVM, le portant à la version 20.1.0, le précédent mécanisme pour injecter ProloGaal aux langages GraalVM connus ne fonctionne plus lorsque cela implique d'exécuter des tests unitaires. Cette précédente technique consiste à utiliser une VM option permettant d'effectuer ce travail. Cette méthode fonctionne toujours lorsque cela implique de charger des langage pour une exécution classique (sans effectuer de tests unitaires), mais le langage ne sera pas détecté si l'on décide d'exécuter les tests unitaires.

Disque local (C:) > Programmes > Java > graalvm-ce-java11-20.1.0 > languages






Nom	Modifié le	Type
 antlr	17.07.2020 00:29	Dossier de fichiers
 js	17.06.2020 00:56	Dossier de fichiers
 nfi	17.06.2020 00:56	Dossier de fichiers
 pl	17.07.2020 00:29	Dossier de fichiers
 regex	17.06.2020 00:56	Dossier de fichiers

FIGURE 12 – Contenu du dossier "languages" du build GraalVM pour gérer les tests unitaires

Pour contourner ce problème sous GraalVM 20.1.0 (pour Java 8 ou 11), il est nécessaire d'avoir le jar généré du langage ProloGaal ainsi que celui d'ANTLR dans le dossier "[jre/]languages" du JDK GraalVM, comme montré sur la figure 12 (le dossier "jre" est présent uniquement dans le build Java 8). Le dossier "pl" possède simplement le jar du langage généré par ProloGaal à chaque exécution. Il est cependant nécessaire de déplacer ces dossiers dans un dossier supplémentaire "temp" si l'on veut réellement exécuter ProloGaal, car ce mécanisme d'injection de langage prime sur celui décrit plus haut, et les modifications alors apportées à ProloGaal sembleront ne pas s'effectuer, car le langage "pl" présent dans "[jre/]languages" prime sur celui ajouté par la VM option.

### 6.2 Nouveaux tests unitaires

De nouveaux tests unitaires ont été ajoutés pour tester les différents builtins ajoutés durant le précédent projet et celui-ci, étant au nombre de 4. Ceux-ci testent un prédicat builtin particulier

si celui-ci possède un comportement complexe, comme le prédicat `is/2` par exemple, ou un groupe de builtins, tels les opérateurs usuels. Regardons par exemple le contenu du fichier "09\_is.output", fichier contenant les résultats attendus après exécution des requêtes présentes en commentaire :

```
% '=' (A, log(exp(10))), is(10, A).
A = log/1(exp/1(10))
yes
% is(3, '+'(2,1)).
yes
% is(1, '-'(2,1)).
yes
% is(8, '*'(4,2)).
yes
% is(2, '/'(4,2)).
yes
% is(1024, '**'(2,10)).
yes
% is(27, '^'(3,3)).
yes
% is(1, mod(21,2)).
yes
% is(2, abs('-'(0,2))).
yes
% is(2, sqrt(4)).
yes
% is(A, '-'(0,1)), is(A, sign('-'(0,271))).
A = -1
yes
% is(A, sqrt('*'('+('-(log(exp(10)),5),13),2))), '='(B, 6),
    '='(B, A).
A = 6.0
B = 6
yes
```

Voici le type des nouveaux tests apportés. Ils testent les différents comportements possibles du prédicat builtin et s'assurent également que celui-ci ne réponde pas systématiquement yes, comme visible à la dernière ligne, où l'on s'assure du résultat stocké dans la variable A étant égale à 6 via l'utilisation du prédicat builtin `=/2`.

Les résultats après exécution des nouveaux tests unitaires sont visibles sur la figure 13. Les lignes suivant les commentaires **-EXPECTED-** sont les résultats stockés dans les fichiers ".output" des tests unitaires. Les lignes suivant les commentaires **-ACTUAL-** sont les résultats retournés

par l'interpréteur après exécution des requêtes. Le dernier message annonce que tous les tests unitaires sont validés.

Certains prédicat builtins n'ont pas été testés. On peut par exemple citer le prédicat builtin `trace/0`, dont le fonctionnement n'est pas encore totalement fidèle à celui de Prolog, et qu'il a donc été jugé non-nécessaire de le tester pour l'instant. Le prédicat builtin `real_time/1` n'est également pas testé, car son comportement exact est imprévisible et il est difficile d'établir un test pertinent pour lui.

```
-----07_consult(ch.heiafr.prolograal.ProloGaalSimpleTestSuite)-----
----- EXPECTED -----
[[yes], [Clauses = test(10). test(12). string(abc)., yes], [yes], [A = test, yes], [yes], [no], [yes], [yes], [yes]]
----- ACTUAL -----
[[yes], [Clauses = test(10). test(12). string(abc)., yes], [yes], [A = test, yes], [yes], [no], [yes], [yes], [yes]]
-----08_operators(ch.heiafr.prolograal.ProloGaalSimpleTestSuite)-----
----- EXPECTED -----
[[A = 10, B = 5, yes], [yes], [no], [no], [no], [yes], [no], [yes], [no], [yes], [no], [yes], [yes], [yes], [yes]]
----- ACTUAL -----
[[A = 10, B = 5, yes], [yes], [no], [no], [no], [yes], [no], [yes], [no], [yes], [no], [yes], [yes], [yes], [yes]]
-----09_is(ch.heiafr.prolograal.ProloGaalSimpleTestSuite)-----
----- EXPECTED -----
[[A = log(1/exp(1(10))), yes], [yes], [yes], [yes], [yes], [yes], [yes], [yes], [yes], [yes], [yes], [yes], [A = -1, yes], [A = 6.0, B = 6, yes], [yes]]
----- ACTUAL -----
[[A = log(1/exp(1(10))), yes], [yes], [yes], [yes], [yes], [yes], [yes], [yes], [yes], [yes], [yes], [yes], [A = -1, yes], [A = 6.0, B = 6, yes], [yes]]
-----10_polyglot_eval(ch.heiafr.prolograal.ProloGaalSimpleTestSuite)-----
----- EXPECTED -----
[[A = 55, yes], [A = hello world, yes], [A = false, yes], [yes]]
----- ACTUAL -----
[[A = 55, yes], [A = hello world, yes], [A = false, yes], [yes]]
Tests run: 10, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.793 sec
```

FIGURE 13 – Exécution des tests unitaires sur la dernière version de ProloGaal



## 7 Améliorations possibles

Ce projet ayant apporté de nombreuses contributions à ProloGaal, il reste néanmoins beaucoup d'améliorations possibles pour ProloGaal :

- Utilisation d'un caractère d'échappement pour pouvoir utiliser n'importe quel caractère pour un nom d'atome ;
- Extend le ParserListener actuel pour le spécialiser dans le parsing des requêtes ou règles Prolog ;
- Actuellement, les prédicats noeuds builtins étendent de `binaryHeadNode`, `ternaryHeadNode`, etc... pour gérer les arguments du prédicat. Il serait possible d'utiliser une notation Truffle pour déclarer un tableau d'argument plutôt que d'étendre `binaryHeadNode` pour pouvoir posséder deux arguments. Cette notation est présente dans le fichier `SLBuiltinNode.java`<sup>30</sup> présent dans le projet GraalVM SimpleLanguage ;
- Ajout du prédicat `cut/0 (!)` ;
- Ajout des prédicats `assert/1` et `retract/1` ;
- Ajout de l'écriture simplifiée pour les opérateurs usuels Prolog et pour le prédicat `is/2` ;
- Implémenter les opérateurs usuels restants (`=../2`, `:=/2`, etc...) ;
- Il semble que l'implémentation actuelle des structures builtin, les `ProloGaalBuiltinStructure` ne soit pas la meilleure à préconiser, et qu'il serait préférable de trouver une manière d'étendre ce concept aux termes Prolog plutôt qu'aux structures ;
- Modifier le système actuelle de résolution d'une question Prolog pour ne plus copier constamment les règles du `ProloGaalRuntime` ;
- Rendre générique au sens Truffle les prédicats builtins ProloGaal, listes, etc... et les rendre ainsi utilisables dans d'autres langages GraalVM ;
- Corriger `is/2` pour qu'il puisse à nouveau utiliser les constantes  $e$ ,  $\pi$ ,  $\tau$  ;
- Implémenter le système d'exception de Prolog.

---

30. GraalVM TEAM. *SLBuiltinNode*, line 61 for *@NodeChild* array - Simple Language Repository. URL : <https://github.com/graalvm/simplelanguage/blob/master/language/src/main/java/com/oracle/truffle/sl/builtins/SLBuiltinNode.java>.

## 8 Conclusion

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

### 8.1 Atteinte des objectifs

Si l'on se réfère à la section "Objectifs" dans l'introduction du rapport, il semble bien que les objectifs principaux du projet aient été réalisés. On notera tout de même que la vitesse d'exécution de ProloGraal, bien qu'accélérée par rapport au premier test de performance effectué dans ce projet, reste très lent comparé aux interpréteurs classiques Prolog. Ce souci est lié à un défaut d'implémentation qu'il n'était initialement pas prévu de corriger pour ce projet, et c'est pourquoi il a été décidé de ne pas se pencher dessus. Si l'on se réfère plutôt au planning du projet, également présenté dans la section "Gestion du projet" de l'introduction, on s'aperçoit que les objectifs "Ajout du prédicat cut (!)" et "Ajout de l'écriture Prolog simplifiée pour les opérateurs et is/2" n'ont pas été documentés dans le rapport ni implémentés. En effet, de par la difficulté à implémenter des deux objectifs, mais également par manque de temps, il a été décidé de ne pas les implémenter.

### 8.2 Conclusion personnelle

En conclusion, j'ai trouvé ce travail de Bachelor formateur et très intéressant d'un point de vue gestion du projet. Les différentes séances, généralement, permettaient de confirmer ou corriger les objectifs établis avant la séance, et le reste de la semaine permettait d'appliquer les différentes décisions entreprises mais également de planifier les prochaines selon les informations récoltées tout au long.

Il a quelques fois été difficile de devoir prendre des décisions d'implémentation, car celle-ci devaient non-seulement refléter fidèlement les concepts du langage Prolog, mais également être valide d'un point de vue Truffle et respecter la philosophie de programmation Java. Bien que cela rendait les décisions d'implémentation difficiles, il était cependant très intéressant de devoir parfois mélanger les trois concepts. Les différents outils offerts par Truffle sont également très riches et puissants, et j'aurais personnellement apprécié pouvoir mieux les intégrer à ProloGraal. Je pense qu'il aurait été plus sage de d'avantage s'attaquer aux tests unitaires et à la rédaction du rapport durant la réalisation du projet, il m'arrive souvent de trop me focaliser sur l'implémentation lorsque celle-ci soulève des problèmes.

## **9 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 16 juillet 2020

## **10 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. Je remercie également l'Expert Julien Bégard pour son intérêt porté sur le projet.

# 11 Bibliographie

## Références

- [1] UNSW COMPUTING. *Prolog arity*. URL : <http://www.cse.unsw.edu.au/~billw/dictionaries/prolog/arity.html>.
- [2] JETBRAINS. *Download IntelliJ*. URL : <https://www.jetbrains.com/idea/download/other.html>.
- [3] Martin Spoto & Tony LICATA. *ProloGaal Repository*. URL : <https://gitlab.forge.hefr.ch/tony.licata/prolog-truffle/>.
- [4] ORACLE. *BiFunction JavaDoc*. URL : <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/BiFunction.html>.
- [5] ORACLE. *Function JavaDoc*. URL : <https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html>.
- [6] Tau PROLOG. *is/2 - Manual*. URL : <http://tau-prolog.org/documentation/prolog/builtin/is/2>.
- [7] GraalVM TEAM. *@Cached JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/dsl/Cached.html>.
- [8] GraalVM TEAM. *@ExplodeLoop JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/nodes/ExplodeLoop.html>.
- [9] GraalVM TEAM. *@NodeChild JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/dsl/NodeChild.html>.
- [10] GraalVM TEAM. *@TypeSystem JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/dsl/TypeSystem.html>.
- [11] GraalVM TEAM. *Context JavaDoc*. URL : <https://www.graalvm.org/sdk/javadoc/org/graalvm/polyglot/Context.html>.
- [12] GraalVM TEAM. *GenerateWrapper JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/instrumentation/GenerateWrapper.html>.
- [13] GraalVM TEAM. *Github release of JVM 20.1.0*. URL : <https://github.com/graalvm/graalvm-ce-builds/releases/tag/vm-20.1.0>.
- [14] GraalVM TEAM. *InstrumentableNode JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/instrumentation/InstrumentableNode.html>.
- [15] GraalVM TEAM. *LoopNode JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/nodes/LoopNode.html>.

- [16] GraalVM TEAM. *Native Image*. URL : <https://www.graalvm.org/docs/reference-manual/native-image/>.
- [17] GraalVM TEAM. *Node JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/nodes/Node.html>.
- [18] GraalVM TEAM. *SLBuiltinNode*, line 61 for @NodeChild array - *Simple Language Repository*. URL : <https://github.com/graalvm/simplelanguage/blob/master/language/src/main/java/com/oracle/truffle/sl/builtins/SLBuiltinNode.java>.
- [19] GraalVM TEAM. *SLEvalBuiltin* - *Simple Language Repository*. URL : <https://github.com/graalvm/simplelanguage/blob/master/language/src/main/java/com/oracle/truffle/sl/builtins/SLEvalBuiltin.java>.
- [20] GraalVM TEAM. *SLTypes* - *Simple Language Repository*. URL : <https://github.com/graalvm/simplelanguage/blob/master/language/src/main/java/com/oracle/truffle/sl/nodes/SLTypes.java>.
- [21] GraalVM TEAM. *Specialization JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/dsl/Specialization.html>.
- [22] GraalVM TEAM. *Value JavaDoc*. URL : <https://www.graalvm.org/truffle/javadoc/org/graalvm/polyglot/Value.html>.
- [23] Boğaziçi UNIVERSITY. *HOW TO CONVERT A RECURSIVE ALGORITHM TO A NON-RECURSIVE ONE*. URL : <https://www.cmpe.boun.edu.tr/~akin/cmpe160/recursion.html>.
- [24] WIKIPEDIA. *Prolog*. URL : <https://fr.wikipedia.org/wiki/Prolog>.